Holden Stefan
09/28/2021

Demonstrating Porous Defenses

Summary:

Porous Defenses, in general, deal with insecure trust in applications and their usage. Defenses in this context could deal with permissions and authorization, authentication, encryption, or dangerous functions within the software. To quickly summarize this class of weaknesses, it could be said that porous defenses are the result of uninhibited trust in users and lack of understanding of possible consequences that result from that trust.

The first weakness demonstrated here is CWE-807: Reliance on Untrusted Inputs in a Security Decision. In its essence, this weakness is the idea that certain inputs such as cookies, variables or hidden fields, that may not be apparent to a user, are safe to use when making security decisions, especially when those decisions deal with trust. The reality is that many inputs, especially those that are stored client-side and relayed to the server via HTTP headers can be modified, and affect applications in dangerous ways. Whether these modifications are a result of javascript, intercept proxies, or another customized attack-vector, anything sent from the client should not be taken for granted as trustworthy even if there are client-side controls in place.

The second weakness to be demonstrated is CWE-798: Use of Hard-Coded Credentials. This weakness is simple, and maybe even obvious, in theory, but in practice can end up being much more nuanced and difficult to both detect and mitigate once it is in place. The idea behind this weakness is that software is shipped with a default, hard-coded set of credentials that grants access to some part of the application. It can be split into two variants: inbound and outbound, which can lead to different results, but are both yielded from the same underlying vulnerability. This demonstration is of the ‘inbound’ variant, where ‘HTTP Basic Authorization’ (using a 401 response and a ‘WWW-Authenticate’ header) has hardcoded credentials that grant anyone full access to the server.

In mitigating the first weakness (Untrusted Inputs), I was able to successfully secure the target functionality by storing the cookie on the server, so that modifications on the client-side had no effect when decisions were made on the server-side.

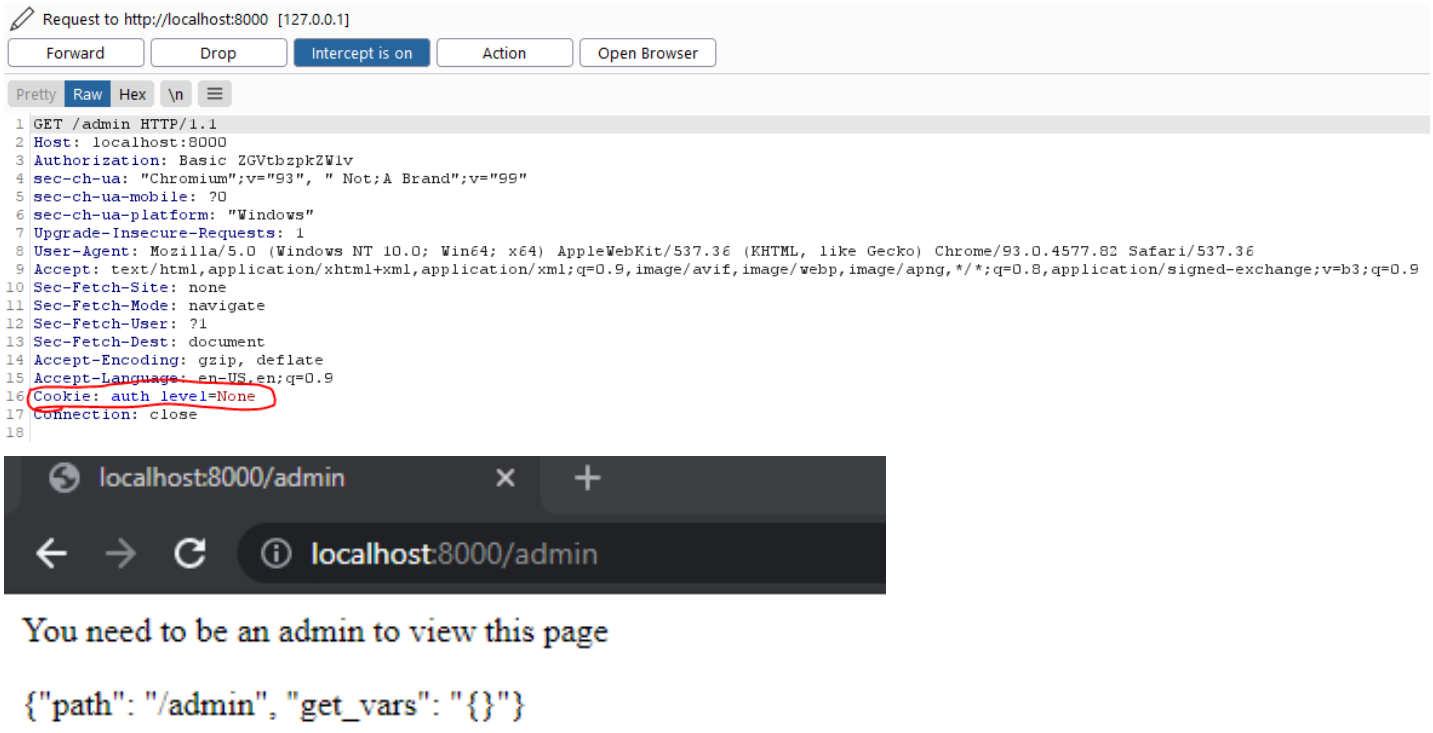
When it comes to the second weakness (Hard-Coded Credentials), I was successful in this very basic application, however, it would still rely on a rather active management team to ensure that the server remained secure. My solution involved employing a separate json file to store the credentials, allowing them to be changed, as well as utilizing a login-attempt counter, to limit the number of login attempts, and locking the user out after the limit is reached. I found, that the login-attempt counter worked in basic usage of the application (sending a 403 response after the limit was reached and blocking any further attempts to gain access to the server), however, it could still be bypassed using any number of brute-forcing tools, due to the fact that I did not implement a way to track a session in any meaningful way. For example, using Burp Suite Intruder, a brute force attack was not locked out after a certain number of attempts, and could successfully yield the correct encoded key, which could easily be decoded and used again. In the current session, even with the correct key the server would lock the user out, however, simply starting another session could easily bypass this. My thinking was that using

the json file to proactively change the credentials, as well as increase the difficulty of brute-force attacks, would limit the exploitability, but this would require fairly active management, and likely need additional functionality to work effectively.

*Just as a note: since most of the weaknesses for this week's assignment dealt with back-end programming, I decided to try to learn more and write my own Python HTTP server. So there are probably quite a few mistakes, and additional weaknesses beyond the scope of these two. Additionally, while the servers run successfully on AWS-Cloud9, I was not able to analyze them properly, and so I ran them on my local machine which allowed me to use Burp Suite to do more in-depth analysis.

Example 1 [CWE-807: Reliance on Untrusted Inputs in a Security Decision]:
Overview:

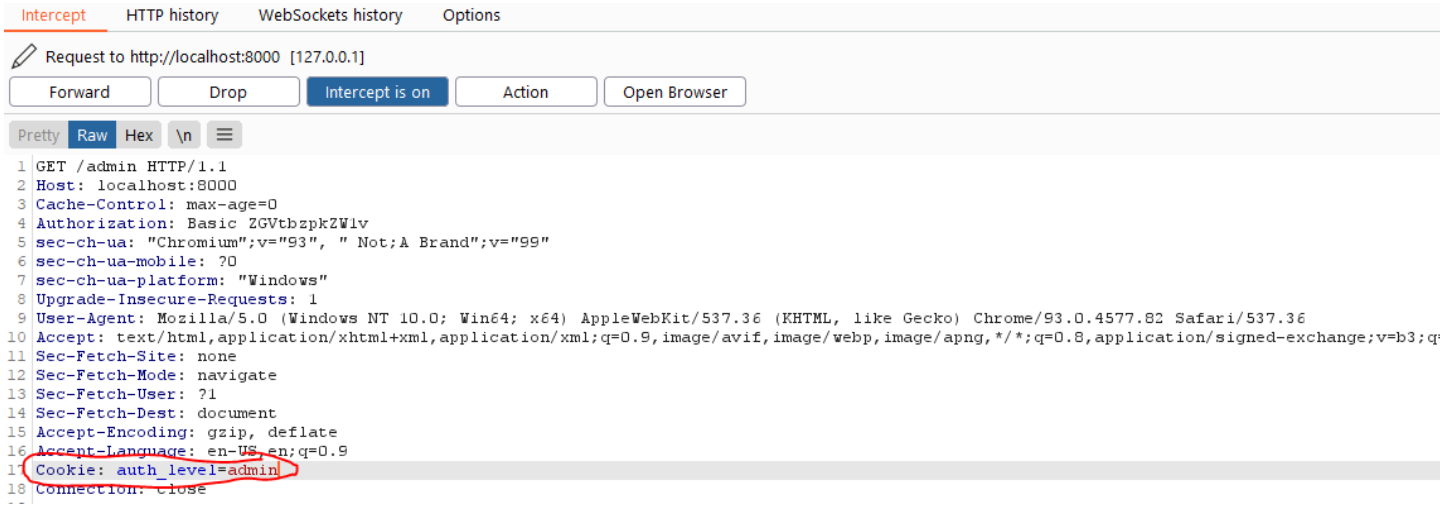
For the demonstration of this weakness, I made use of cookies for defining a level of authority for a user. As this was a very stripped down application, the basic premise was that there was an admin page served by an HTTP server written in python. There was a cookie implemented, which if set to 'None', the admin page would return 'Must be an admin to view this page', but if that cookie was set to 'admin' the page would return 'Welcome, admin!'. There was no actual functionality to set that cookie to 'admin' and so it would always be set to 'None', but could be intercepted and changed, and the server would trust whatever this cookie was even if it came from the client.



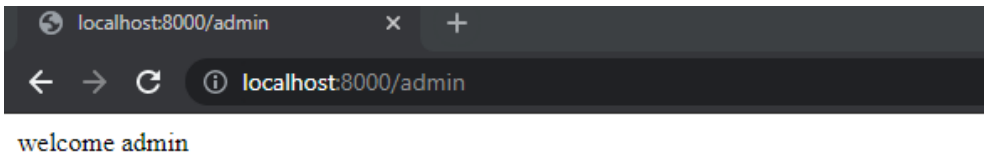
Cookie working properly: blocking access when set to 'None

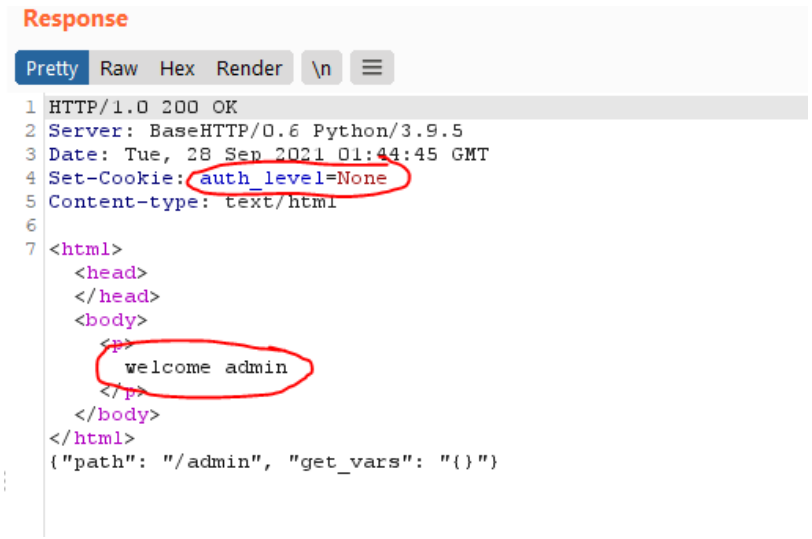
Analysis:

Exploiting this vulnerability was very straightforward, as the cookie in question was stored on the client-side, and could be modified to any value which would allow it to be interpreted by the server at face-value.



Using Burp Suite to intercept the request, I could simply change the value of the cookie from ‘None’ to ‘admin’ and forward that to the server. The server, even though there was no way to set that cookie to ‘admin’, simply used a conditional to check the value, and if that value was ‘admin’ would return the admin page.



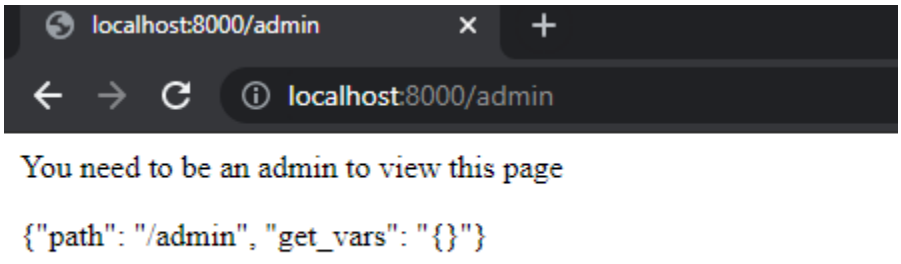


Another interesting issue that I didn't expect, was that since the cookie on the server would always be set to 'None', the response would have the cookie set to 'None', yet still display the admin page.
The vulnerable function in question:

```
if self.path == '/admin':
    cookies = self.parse_cookies(self.headers['Cookie'])
    if cookies['auth_level'] == 'admin':
    #todo
        self.wfile.write(bytes('<html><head></head><body><p>welcome admin</p></body></html>', 'utf-8'))
    else:
        self.wfile.write(bytes('<html><head></head><body><p>You need to be an admin to view this page</p></body></html>', 'utf-8'))
```

Mitigation:

Mitigating this weakness was likewise easy to do. The cookie was stored on the server, and simply displayed on the client-side via its header. This way the cookie's value could not be changed by the client, and even if the attacker intercepted the request and modified the value, it would still make use of the variable stored on the server to confirm the level of authority.



Truthfully, this sort of negated the need for the cookie at all in this specific case, yet still demonstrates the use of server-side storage for trusted inputs.

The corrected cookie implementation and conditional:

```
c_val = None
self.send_response(200)
self.send_header('Set-Cookie', f'auth_level={c_val}')
```

```
if self.path == '/admin':
    cookies = self.parse_cookies(self.headers['Cookie'])
    if cookies['auth_level'] == 'admin' and c_val == 'admin':
    #todo
        self.wfile.write(bytes('<html><head></head><body><p>welcome admin</p></body></html>', 'utf-8'))
    else:
        self.wfile.write(bytes('<html><head></head><body><p>You need to be an admin to view this page</p></body></html>', 'utf-8'))
```

Example 2 [CWE-798: Use of Hard-Coded Credentials]:

Overview

To demonstrate the weakness of hard-coded credentials, I made use of the HTTP Basic Authentication protocol, which essentially, upon getting a request from an unknown session, will respond with a 401 and 'WWW-Authenticate' response and header respectively, along with a challenge, which in this case was a username/password form. If the username/password were entered incorrectly, this process would continue until it was correctly entered --granting access to the server. Furthermore, since the credentials for the challenge were hard-coded, it would make changing them very difficult, and so post-compromise, resolving the issue would be a real challenge.

Response

Pretty Raw Hex Render \n

```
1 HTTP/1.0 401 Unauthorized
2 Server: BaseHTTP/0.6 Python/3.9.5
3 Date: Tue, 28 Sep 2021 02:41:56 GMT
4 WWW-Authenticate: Basic realm="Test
5 Content-type: text/html
6
7 {"success": false, "error": "Auth Header not received"}
```

Response to an incorrect/incomplete HTTP Basic Auth request

Analysis:

While the hard-coded credentials could theoretically be well protected and kept secret, in its current implementation however, this server is vulnerable to brute-force attacks, and essentially incapable of changing credentials even if an attack is detected. In order to demonstrate a brute-force attack, Burp Suite Intruder was used, where the HTTP Basic Auth request was able to be repeatedly modified and sent, until a 200 response was returned. This was just a basic list brute-force, and in reality would take much longer, but for demonstration purposes, a very short list was generated, which would be Base64 encoded and inserted into the appropriate header before a request was sent.

Target | Positions | Payloads | Resource Pool | Options

Payload Positions

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions - see help for full details.

Attack type: Sniper

```
GET / HTTP/1.1
Host: localhost:8000
sec-ch-ua: "Chromium";v="93", " Not;A Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
Authorization: Basic §§
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.82 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: auth_level=None
Connection: close
```

Above: The target request in Burp Suite, the highlighted ‘S’ symbols are the location for the payload.

Attack | Save | Columns | 5. Intruder attack of localhost - Temporary attack - N

Results | Target | Positions | Payloads | Resource Pool | Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length
0		401	☐	☐	213
1	YWRtaW46YWRtaW4=	401	☐	☐	213
2	cm9vdDpyb290	401	☐	☐	213
3	dGVzdDp0ZXN0	401	☐	☐	213
4	ZGVtbzpkZW1v	200	☐	☐	176

Request | Response

Pretty | Raw | Hex | \n | ≡

```
GET / HTTP/1.1
Host: localhost:8000
sec-ch-ua: "Chromium";v="93", " Not;A Brand";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
Authorization: Basic ZGVtbzpkZW1v
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/5
Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,im
-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
```

Above: The list of responses from each payload delivered, the highlighted response with the 200 status code is displayed, where the red circle depicts the correct, base64 encoded key to gain access.

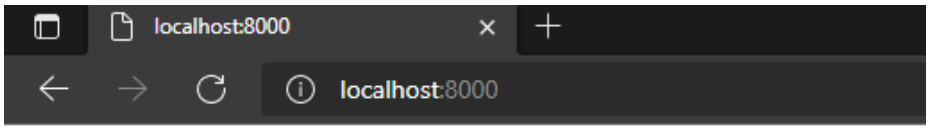
It is difficult to show a single code-snippet for this vulnerability, because essentially, the whole server is vulnerable. Each request, first checks for the correct key, and if it is not present, will begin the HTTP Basic Auth request/response process.

Mitigation:

In order to mitigate this weakness, 2 approaches were used. The first was a login-attempt tracker, which honestly was not easy to implement with this HTTP Basic protocol. Essentially I had to increment the login_attempt variable each time there was a request. So it is not actually tracking the number of attempted logins, but every single request made. Then I had to use a conditional in the HTTP Auth header function, which would only execute the authentication protocol if the number of requests was less than 5, otherwise, it would send back a 403 status, and from there the user was no longer able to attempt to login, and could not gain access to the server. The second approach was to use an external json file to store the credentials. This enables the credentials to easily be changed in the event of an attack. In practice, this should be done using a database, in conjunction with encryption to add another layer of security.

The result of these approaches was that after 5 requests were made without the proper authentication key, the user was locked out of the server.

```
PS C:\Users\Holden Stefan\Desktop\SDEV325\week6> python py_server3.py
running at http://localhost:8000
127.0.0.1 - - [28/Sep/2021 02:36:58] "GET / HTTP/1.1" 401 -
Number of login attempts: 2
127.0.0.1 - - [28/Sep/2021 02:37:56] "GET / HTTP/1.1" 401 -
Number of login attempts: 3
127.0.0.1 - - [28/Sep/2021 02:37:57] "GET / HTTP/1.1" 401 -
Number of login attempts: 4
127.0.0.1 - - [28/Sep/2021 02:37:58] "GET / HTTP/1.1" 403 -
Exceeded allowable login attempts: authorization denied
127.0.0.1 - - [28/Sep/2021 02:37:58] "GET /favicon.ico HTTP/1.1" 403 -
Exceeded allowable login attempts: authorization denied
```



{"success": false, "error": "Too many attempts[5]"}

In the old function for sending the HTTP Basic Authentication header, there were zero checks, which enabled the authentication process to continue indefinitely. This was the main culprit in the server's brute-force vulnerability.

```
def do_AUTHHEAD(self):
    self.send_response(401)
    self.send_header('WWW-Authenticate', 'Basic realm="Test"')
    self.send_header('Content-type', 'text/html')
    self.end_headers()
```

In the new function for sending the HTTP Basic Authentication header, a conditional check was implemented where the number of login-attempts was used to determine whether or not the response would be a 401/WWW-Authenticate or a 403/forbidden.

```
def do_AUTHHEAD(self):
    if self.login_attempts <5:
        self.send_response(401)
        self.send_header('WWW-Authenticate', 'Basic realm="Test')
        self.send_header('Content-type', 'text/html')
        self.end_headers()
    else:
        response = {
        'success': False,
        'error': f'Too many attempts[{self.login_attempts}]'
        }
        self.send_response(403)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(bytes(json.dumps(response), 'utf-8'))
```

The main issue was that no effective session management was implemented, making it still very possible to brute-force and see the auth-key. While in the current session, you may still be blocked from accessing the server, there is nothing stopping an attacker from starting a new session and simply using what they gained from the previous attack to gain access on the first try.

Sources

MITRE. (2010, January 15). *Common weakness enumeration*. CWE. Retrieved September 28, 2021, from http://cwe.mitre.org/data/definitions/798.html.

MITRE. (2010, January 18). *Common weakness enumeration*. CWE. Retrieved September 28, 2021, from http://cwe.mitre.org/data/definitions/807.html.

Mozilla. (n.d.). *WWW-authenticate - http: MDN*. HTTP | MDN. Retrieved September 29, 2021, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/WWW-Authenticate.