CS4404

Mission Debriefing Report of Mission 3

Matthew Malone, Nour Elmaliki

Table of Contents

Introduction:

This paper will attempt to explore single sign on (SSO) utilities and their vulnerabilities. In our experiments we will accomplish the following:

1. Set up a SSO infrastructure and misconfigure it so it is vulnerable to attacks.
2. Use a replay attack to steal login strings and then use them to gain unauthorized access to a user's account.
3. Implement a defense based on cryptographic nonce values to prevent replay attacks.

Reconnaissance

Identity Providers

- Authenticity
    - User identity must be correct, otherwise relying parties cannot trust the IP
    - Without any controls, an attacker could impersonate someone with sufficient access to the target relying party
    - Kerberos uses a password or key unique to the user is used to decrypt the Auth response
- Availability
    - Whole infrastructure is bottlenecked around the IP, if it goes down then nobody will be able to authenticate
    - Kerberos is vulnerable to an availability attack as an attacker with a valid username could request lots of tickets, forcing the IP to generate many responses without verifying the requesting party

Relying Parties

- Authenticity
    - Reliance on the IP
    - An attacker can replay a previous ticket or a ticket to a different relying party
        - https://sso-attacks.org/Token_Recipient_Confusion
        - Access Token misuse:
            - This attack is common in OAuth 2 when RP's use the *access_token* to authenticate users(*access_token* is generic(non-rp specific) token for any user)
        - Signed_Request Misuse:
            - OAuth2 - *Signed_request* is decoded but signature is not checked w/ the *App_secret*
    - Relying party should use unique identifiers for each ticket, and verify that a ticket is for the correct RP
- Integrity
    - The ticket from the Identity provider must remain the same
    - Attackers could change permissions or identity for the relying party
    - Kerberos solution: Tickets are encrypted with a key unknown to a user, preventing users from viewing or changing the ticket contents

Users

- Confidentiality
    - Users have the expectation that their account data is private and other people can't see it
    - Attackers can steal tokens(usually by intercepting traffic) and impersonate users when RP's don't track the number of uses for each ticket.

OpenID vs Kerberos Comparison

Kerberos

Kerberos is meant for controlled environment setups. In this environment, there are known and trusted Kerberos servers. These servers authenticate the service provider to the client, and vice versa. The Kerberos server must know of all clients and all service providers. Kerberos is transparent to the user. Think WPI's login system which uses Kerberos, but is entirely transparent to the end user. One difference between Kerberos and OpenID is that Kerberos can authenticate service providers. Kerberos is completely useless for the most part on the open internet as it only works for a single domain or resource. Kerberos issues tickets to clients as a form of authentication.

OpenID

OpenID is meant for an open environment. Any service provider can work with an OpenID server to authenticate clients. Unlike Kerberos, with OpenID you can go to any website that supports it, and select your own provider to use. Unlike Kerberos, OpenID cannot authenticate service providers. OpenID is good for the open internet as it can be used with multiple domains and services instead of Kerberos which can only be used on one. OpenID issues tokens from an API to authenticate users.

Section 1: Infrastructure Building

This was our first attempt at infrastructure building using PHP with an OpenID 2.0 system. After trying to configure a client to connect to the OpenID provider, we realized that OpenID was too outdated to use for this assignment. It was published in December 2007, and many of the resources on configuring it or setting it up no longer exist. We decided to scrap the infrastructure and redo it with a more modern and up-to-date tool called Keycloak which uses OpenID Connect.

First Attempt

As outlined above, we went with an OpenID implementation for this mission. We looked at the official websites for OpenID support and decided to use a tool called SimpleID found here: https://simpleid.org/. This tool is written in PHP to act as a standalone personal OpenID provider. After zipping and uploading the tool to VM1, it was a matter of installing php and configuring it. See the config.php inside the attached openServer/simpleid/ directory for a full configuration. To create the necessary users, the included default user was modified to include

a new user called admin with a password of password. For our implementation, we decided to just use simple MD5 hashes by running the following command:
echo -n 'password'' | md5sum
Very secure. We know. The goal was to make this as simple as possible and misconfigure the settings. We disabled HTTP to make attacks easier and disabled the internal system it had for verifying traffic. To see the configured user, look at admin.identity under the identities folder in openServer. The entire openID server was configured to use 10.21.8.1:8000 as it's IP address.

Next was a matter of making a simple website server that a client can access to sign on with their openID. PHP was once again our tool of choice with a tool called LightOpenID serving as a connecting bridge between the webserver and the openID server https://github.com/iignatov/LightOpenID. With this light implementation of openID, it was simple to write a short script to start a web server that when queried, redirected the user to sign in using their openID identification. From there, the user is given the choice to cancel authentication resulting in them being directed to a failed login page, or they can sign in and thus be redirected to a success page. Note that the server CAN directly serve the pages themselves due to the simplicity of the setup, the general goal still remains. The purpose of this assignment is the configuration of a SSO system and the exploitation of said SSO implementation. Not the configuration of a web server and the exploitation of it.

How to Replicate the Setup

Begin by transferring the contents of the zip folder onto VM1 and unzipping them. Note you will need to install zip (apt install zip). Next install PHP with apt install php. Then navigate to the webServer and openServer/simpleid/ directories and run the following command:
php -S 10.21.8.1:8000 //For the simpleID server
php -S 10.21.8.1:8080 //For the webServer
This will start the required openID server and the web server for accessing the website sign on page. Note that you can either run these with an & trailing the command in order to background the process or use your own desktop setup with multiple windows. For our purposes, iceWM was used. Run it on any of the team 8 VMs by running the "startx" command.

Second Attempt

Starter code didn't have any room to change or add new things.
For this implementation, we needed an online VM to configure a tool known as Wildfly which uses maven. Our online VM was configured to have the same local IP as the online one. This was simply done by adding an extra host-only network adapter in VirtualBox and then configuring it by following the instructions here:
https://ernieleseberg.com/virtualbox-with-static-ip-on-ubuntu-server-guest/. Note that this was done on the same Ubuntu 20.04.1 LTS version as seen on the isolated network. To begin, on our online VM we downloaded the latest Keycloak installation and unzipped it and navigated to the "/keycloak-11.0.3/bin/" directory and ran: "./standalone.sh -b 10.21.8.1 -Djboss.socket.binding.port-offset=100"
Note the -b command telling the application to listen on the machine's local IP instead of just the internal localhost. The port offset is set to prevent conflicts with WildFly as described below.

WildFly 20.0.0 was downloaded from here: https://www.wildfly.org/downloads/ and unzipped. It was then necessary to install the required client adapter found at: https://www.keycloak.org/downloads for OpenID Connect. The adapter was extracted to the WildFly directory.
From there the following commands were run:

```
cd wildfly-20.0.0.Final
./jboss-cli.sh --file=adapter-install-offline.cli
bin/standalone.sh -b 10.21.8.1
```

From there we navigated to 10.21.8.1:8080 on firefox to connect to wirefly to verify that it was running. Now we had to create a client in our Keycloak installation so we navigated to http://10.21.8.1:8180/auth/admin on and went to the clients section and created a new client with the following information:

Client ID: vanilla
Client Protocol: openid-connect
Root URL: http://10.21.8.1:8080/vanilla

After creating the client we navigated to the installation tab in Keycloak and selected Keycloak OIDC JSON and then downloaded the file to use with the WildFly configuration by following the following steps:

Navigate to the /standalone/configuration/ directory in the wildfly folder.
Modify the following line at line number 408 in the standalone.xml file to this:

```
<subsystem xmlns="urn:jboss:domain:keycloak:1.1">
  <secure-deployment name="vanilla.war">
    <realm>demo</realm>
    <auth-server-url>http://localhost:8180/auth</auth-server-url>
    <public-client>true</public-client>
    <ssl-required>EXTERNAL</ssl-required>
    <resource>vanilla</resource>
  </secure-deployment>
</subsystem>
```

Now we just needed some more example code from https://github.com/keycloak/keycloak-quickstarts to get the WildFly server up and running. This repository was downloaded and Maven was installed with “apt install maven”. Next the following commands were executed once the repository was unzipped:

```
cd keycloak-quickstarts/app-profile-jee-vanilla/config
cp /home/student/keycloak.json .
cd ..
mvn clean wildfly:deploy
```

Now finally we were able to run the required infrastructure.

Final Iteration

As outlined above, we went with an OpenID Connect implementation for this mission. We looked at the official websites for OpenID support and decided to use a tool called Keycloak found here: https://www.keycloak.org/. This tool is written in Java to act as a standalone personal OpenID Connect provider. Due to the online nature of part of the setup, an online VM

was used to configure the tool, then it was uploaded to the isolated network. Due to the large size of the files, we needed to zip it into multiple parts then extract them that way. Next we needed to setup the necessary routing information for this attack by running the following:

On VM3 10.21.8.3
route add default gw 10.21.8.2
route add 10.21.8.1 gw 10.21.8.2

On VM2 10.21.8.2
route add default gw 10.21.8.2
route add 10.21.8.1 gw 10.21.8.2
route add 10.21.8.3 gw 10.21.8.2
sysctl net.ipv4.ip_forward=1

On VM1 10.21.8.1
route add default gw 10.21.8.2
route add 10.21.8.3 gw 10.21.8.2

To begin, we downloaded the latest Keycloak installation and then split it into parts and uploaded it to the VM and unzipped it. Next we navigated to "/keycloak-11.0.3/bin/" directory and ran: "./standalone.sh -b 10.21.8.1 -Djboss.socket.binding.port-offset=100"
Note the -b command telling the application to listen on the machine's local IP instead of just the internal localhost. The port offset is set to prevent conflicts with other programs we were running at the time.

Next we navigated to http://10.21.8.1:8180/auth/admin to set up the server as described here: https://www.keycloak.org/docs/latest/getting_started/.

To begin, an admin user was set up with the credentials of "admin" and "password" as the corresponding username and password. Very secure. Next, a new realm was created called *Nour*, and then a user was made for testing logins in this realm with the username "test" and password as "password". Keycloak actually allows for multiple OpenID Connect servers or "realms", however for our purpose we only needed one. Keycloak generates hashes of the passwords automatically using SHA256 with 30 passes so no work to generate the hash was necessary on our front. Next we had to set up the client server.

Next we added our own client in Keycloak. A client is basically an authentication server that other relying parties can use to connect to the keycloak instance. To create it, we did the following:

Navigate to http://10.21.8.1:8180/auth/admin and go to the client section awnd create a new client with the following information:

Client ID: nour
Client Protocol: openid-connect
Root URL: http://10.21.8.1:8001

After several failed attempts at using premade SSO clients, we decided to create our own from scratch. We used a python http server to serve html files with some javascript to

interact with the IDP. Since our client is minimalist, we decided to use OpenId's implicit flow for authentication , described here: https://openid.net/specs/openid-connect-core-1_0.html#ImplicitFlowAuth. This is the only flow that doesn't require server side session management and suits a mostly web scripting based server. Implicit flow is essentially just using Oauth2, however OpenID provides some tools to make it a bit more secure for authentication. Real world applications should use https, encrypt the associated tokens and such, and verify responses from the IDP. We mostly forwent those steps in favor of simplicity, and knowing that some of these flaws will be rectified in our defense. For proof that this ran successfully, view screenshot 5 to see a successful login with the print of the user's name.

How to Replicate the Setup

First follow the setup as described above and zip the wildfly and keycloak directories from the online VM and transfer them to the isolated network VM1 in parts. For our configuration 9 zip parts were required and were put together with the following commands:

```
cat output.zip.00* > output.zip
unzip output.zip
cd  keycloak-11.0.3
bin/standalone.sh -b 10.21.8.1 -Djboss.socket.binding.port-offset=100
```

Next configure the keycloak server by adding the realm and users as described in the earlier section. You can access the admin panel by going to the server and logging in with *Admin* and *password.*

Once the IDP is ready, setup the client server. Navigate to the clientServer folder and run:
sudo python3 server.py

This will create a http server available on port 8001. Index.html contains the javascript logic that makes the authentication request to the IDP, index2.html parses the jwt token containing the user's information and just prints their name.

Section 2: Attack

Due to the fact that our attacks would be trivially easy if we only used HTTP as the protocol for on-path adversary attacks (we could simply edit the plaintext nonce segments), we decided to ignore this route, and pursue another route of attack. We decided to focus on a passive on-path adversary attack. Specifically a replay attack.

Replay Attack

A replay attack requires an on-path adversary setup. In this form of attack, the attacker grabs a secure network message, intercepts it, or delays and resends it in order to make the server do what the attacker wants. In the real world, this is surprisingly easy to do, and doesn't require any advanced knowledge. This is because the attacker is simply taking existing data, and redirecting it as necessary or stealing it as necessary. In our implementation, the attacker borrows a login request and uses it later to login under the user. This can be done even with encryption, since we're just copying the entire request. Without HTTPS, we could change the nonce values, but that would be too easy.

As per the Oauth specification found here: https://tools.ietf.org/html/rfc6819#section-4.6.1, access tokens and login information should not be sent over an insecure channel to prevent replay attacks. Well thankfully enough, this SSO server was misconfigured to not use TLS to transport any information.

How to Replicate the Attack[1]

1. Start up the VMs with the infrastructure as found in the Infrastructure section above. Make sure route settings are set.
2. Use Wireshark or a packet sniffer of your choice to intercept HTTP data. See screenshot 1 for proof that this works. You have to type in http into the filter box, and it will display the requisite information. Note that you can go to Edit /Preferences/Protocols/HTTP to add port number 8001 to the HTTP standard so Wireshark knows to decrypt these as HTTP messages. You could also just right click the individual packets and select decode as HTTP. Note that 8001 is the port chosen for the relying party server. See screenshot 1.
3. Capture traffic on the ens3 device with promiscuous mode off as you login as the test user on VM3. See screenshot 2.
4. Capture the full URI from the http request. (Note that realistically you could capture the session key, but we need the entire URL to verify the defense section later). See screenshots 2 as proof for interception of the request, and then 3 for how we ripped the URI off of the packet. Simply right click the full request URI value in the packet, and then select copy then value. See screenshot 3.
5. After stealing the URI, it's a matter of simply copy-pasting the string into a web browser to login as the user. See screenshot 4.

[1] You may notice the url in the attack screenshots includes a nonce, however the Client is not configured to verify the nonce until the defense section.

Section 3: Defense

Our infrastructure uses OpenID Connect instead of Oauth2. This provides additional security options over Oauth2, specifically the ability to use a nonce token. A nonce token is a randomly generated cryptographic token that a server uses to prevent replay attacks. Each login provides a unique nonce token. If a client includes a nonce parameter during a user authentication, the IDP response will include the unaltered nonce. This allows the client or relying party to uniquely identify users authenticated by the identity provider.

How to Replicate the Defense

- On VM1 navigate to the client_w_server folder.
- Run the server with “sudo python3 server.py”. Note as proof, important changes to the code can be seen in screenshots 6 through 9.

How the Defense Works

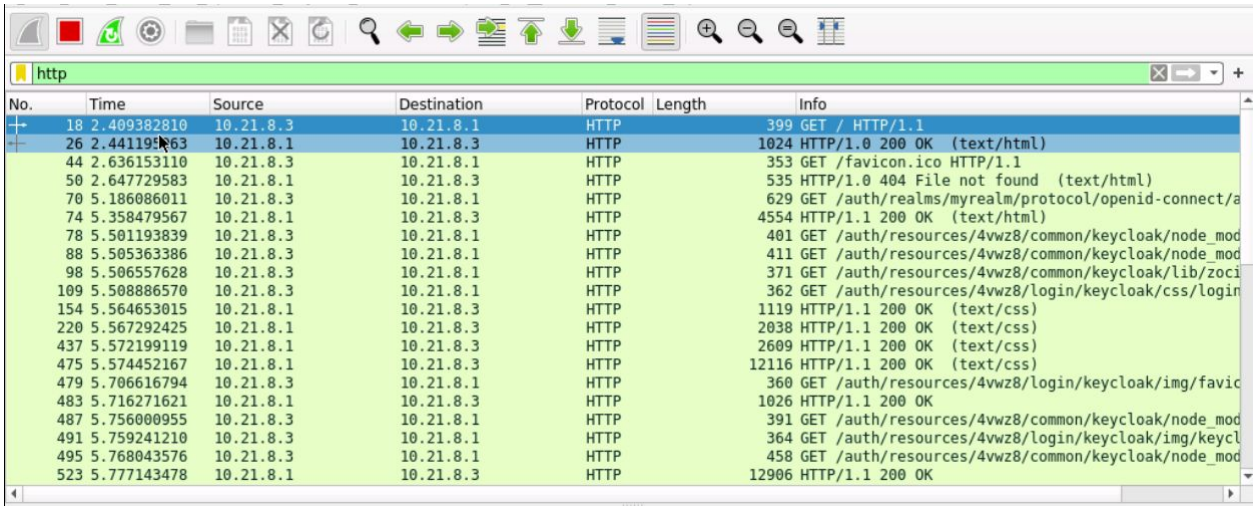
The defense is a rewriting of the SSO client server. When the user is redirected to the identity provider (keycloak) server, they are issued a nonce that is associated with their IP address. When the user is redirected back to the SSO client server from the identity provider (the profile.html page), the identity provider will send their encrypted token with the information. This token contains the cryptographic nonce value from above. The change on this iteration of the client server is that this one will verify the nonce and IP address to ensure that the data has not been modified. It will then allow the user access to secure information. In our case, it’s just their name. The server is also configured to only allow one login per nonce. This prevents replay attacks with spoofed IP addresses using the same nonce. For proof of the defense, see screenshot 10 where a replay attack is attempted after the defense has been implemented. The user is now redirected to a page telling them to stop stealing traffic.

Note that this defense isn’t entirely perfect as anyone can register nonces. However it is a simple implementation of a realistic defense using nonces. Nonces were added to OpenID Connect for this very purpose. The randomly generated nonces that the Keycloak server uses are also not necessarily cryptographically sound and their ID tokens are only encrypted with Base64URL, however this is not what the defense is about.

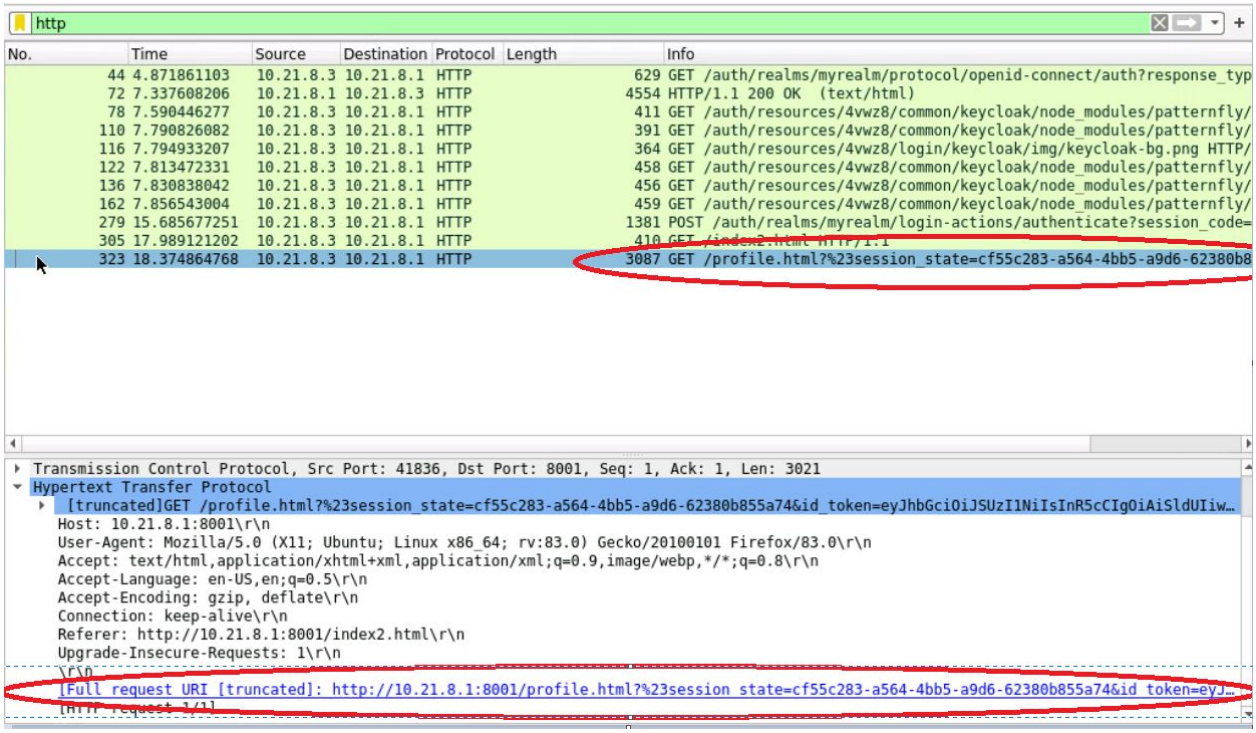
Conclusion

SSO sign ons are rapidly becoming commonplace alternatives to registering a unique account on all websites. These systems have shown many vulnerabilities that we don’t see with traditional manual login systems such as replay attacks among others. Cryptographic nonces are one form of security that these systems use to thwart replay attacks, however they are usually used in combination with other forms of protection such as SSL to encrypt the traffic. Hopefully in the future we will see secure SSO implementations that can be used universally for many commonplace websites.

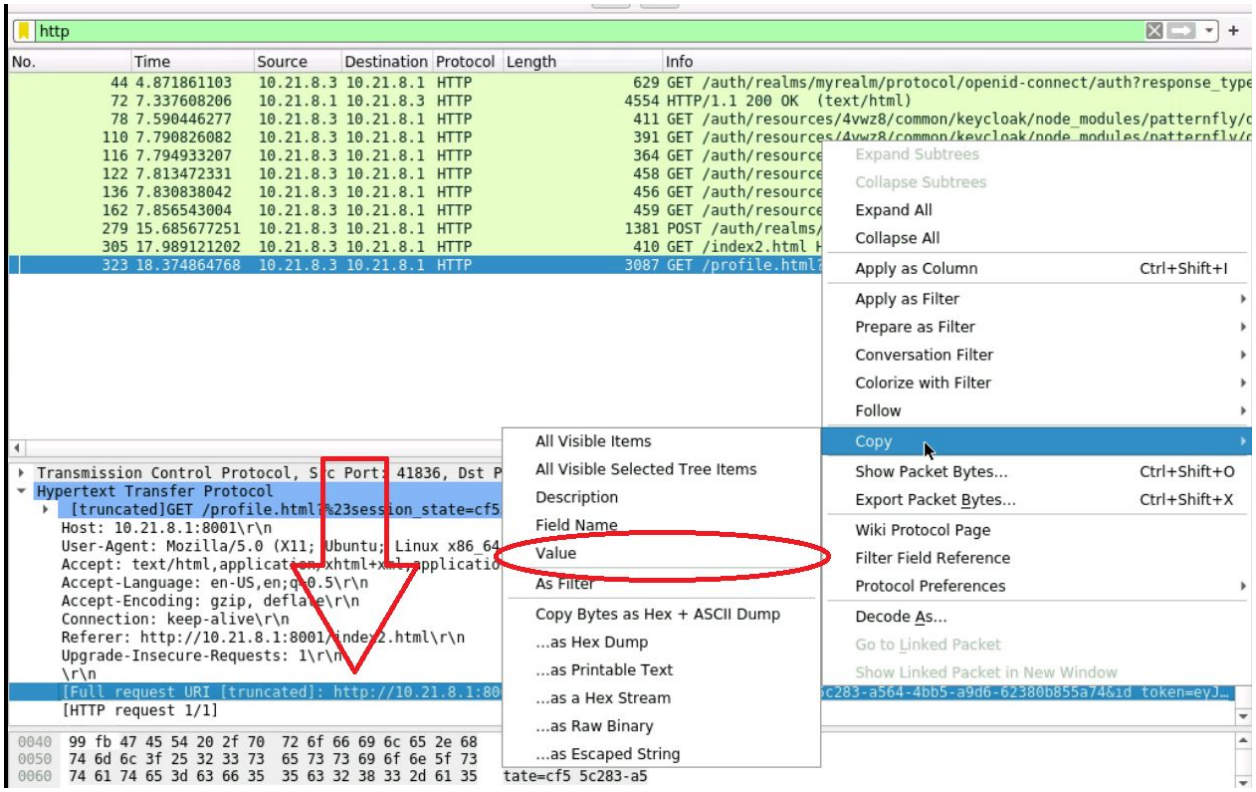
Appendix



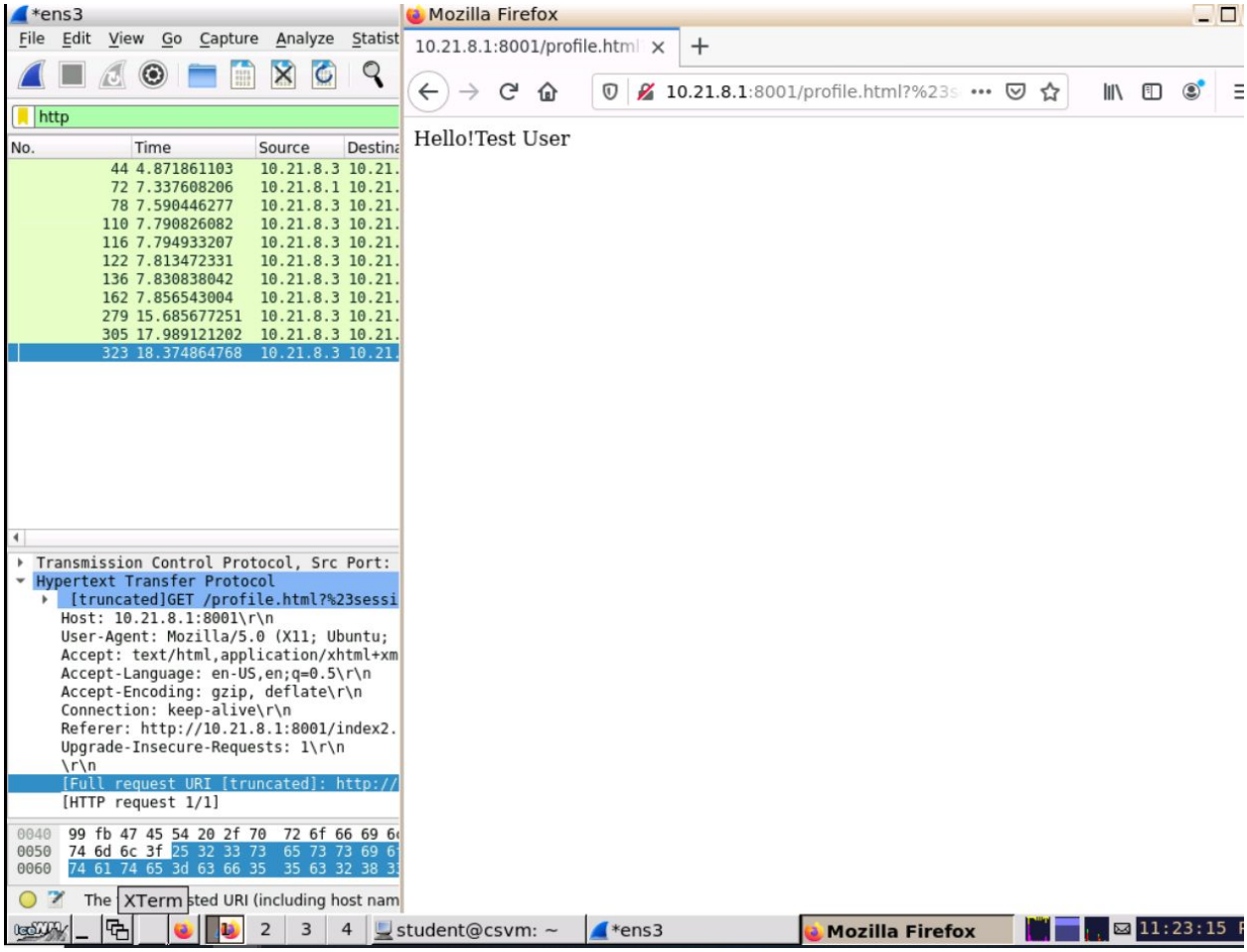
Screenshot 1: Observing Client go through authentication process



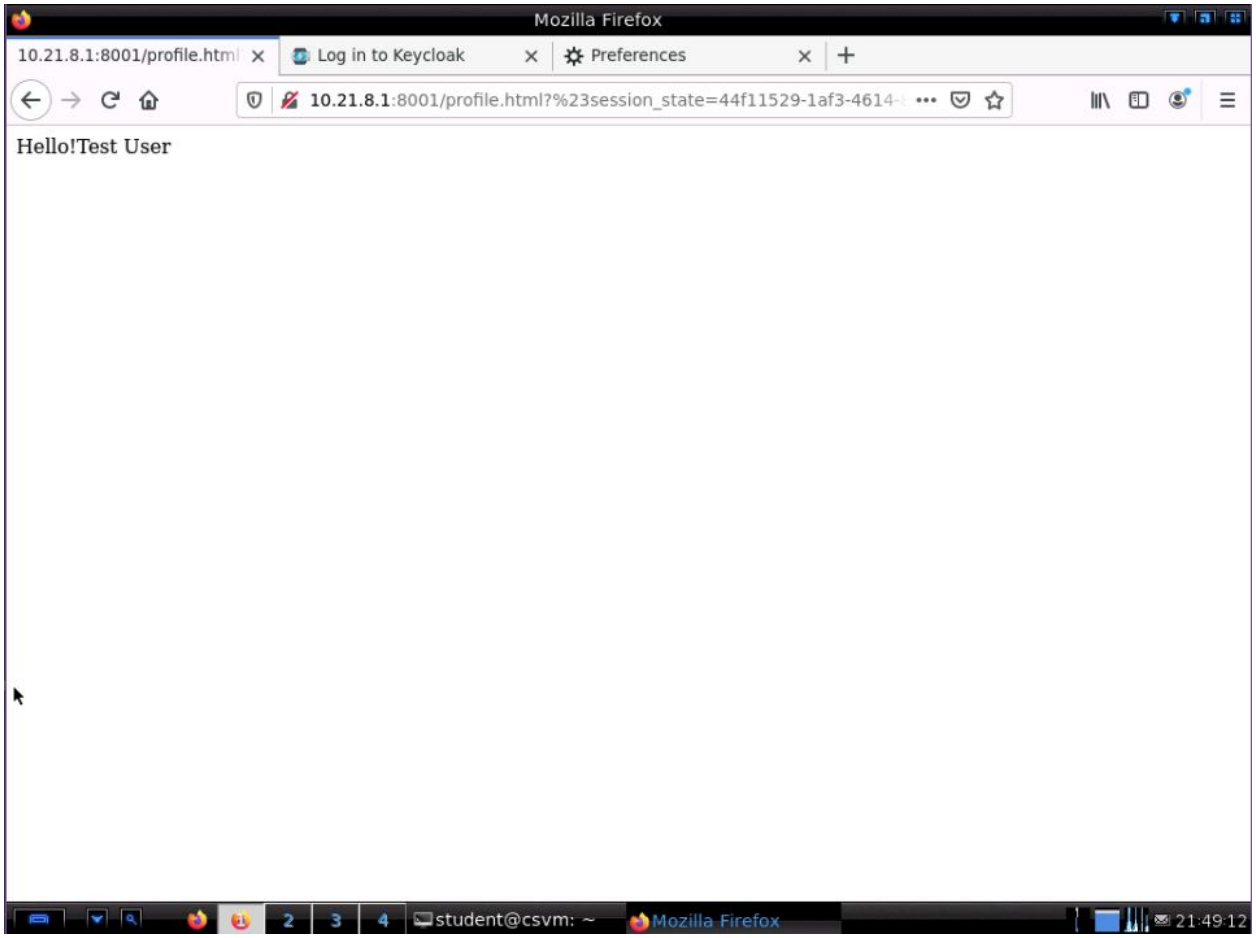
Screenshot 2: Highlighted is the http request once the user has been redirected back to RP



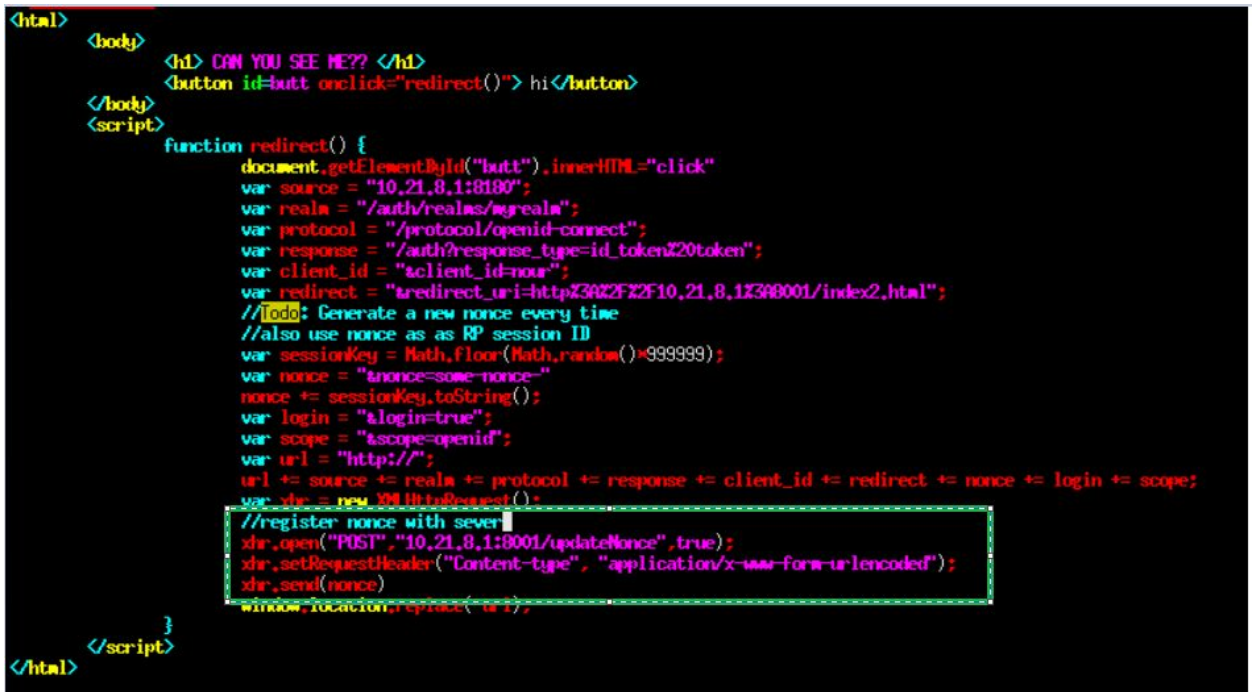
Screenshot 3: How to copy the URI Request to clipboard on Wireshark



Screenshot 4: Attacker hijacking User's profile (that's their username on the webpage)



Screenshot 5: Relying party website with successful login message.



Screenshot 6: User notifies RP of their nonce before being redirected to IDP

```
        def do_POST(self):
            extension = self.path[15:27]
            if extension == '/updateNonce':
                try:
                    source = str(self.rfile.read1(), "utf-8")
                    nonce = source.split("&nonce=")[1]
                    print(nonce)
#                    print(self.rfile.read1().split("&nonce=")[1])
                    if nonce not in nonce_repo:
                        nonce_repo.append(nonce)
                        self.send_response(200)
                        self.end_headers()
                        self.wfile.write(b'Nonce registered!')
                except:
                    print("problem registering nonce")
                    self.send_response(400)
                    self.end_headers()
                    self.wfile.write(b'Sorry something broke')
                    return
            else:
```

Screenshot 7: Server code to register new nonces

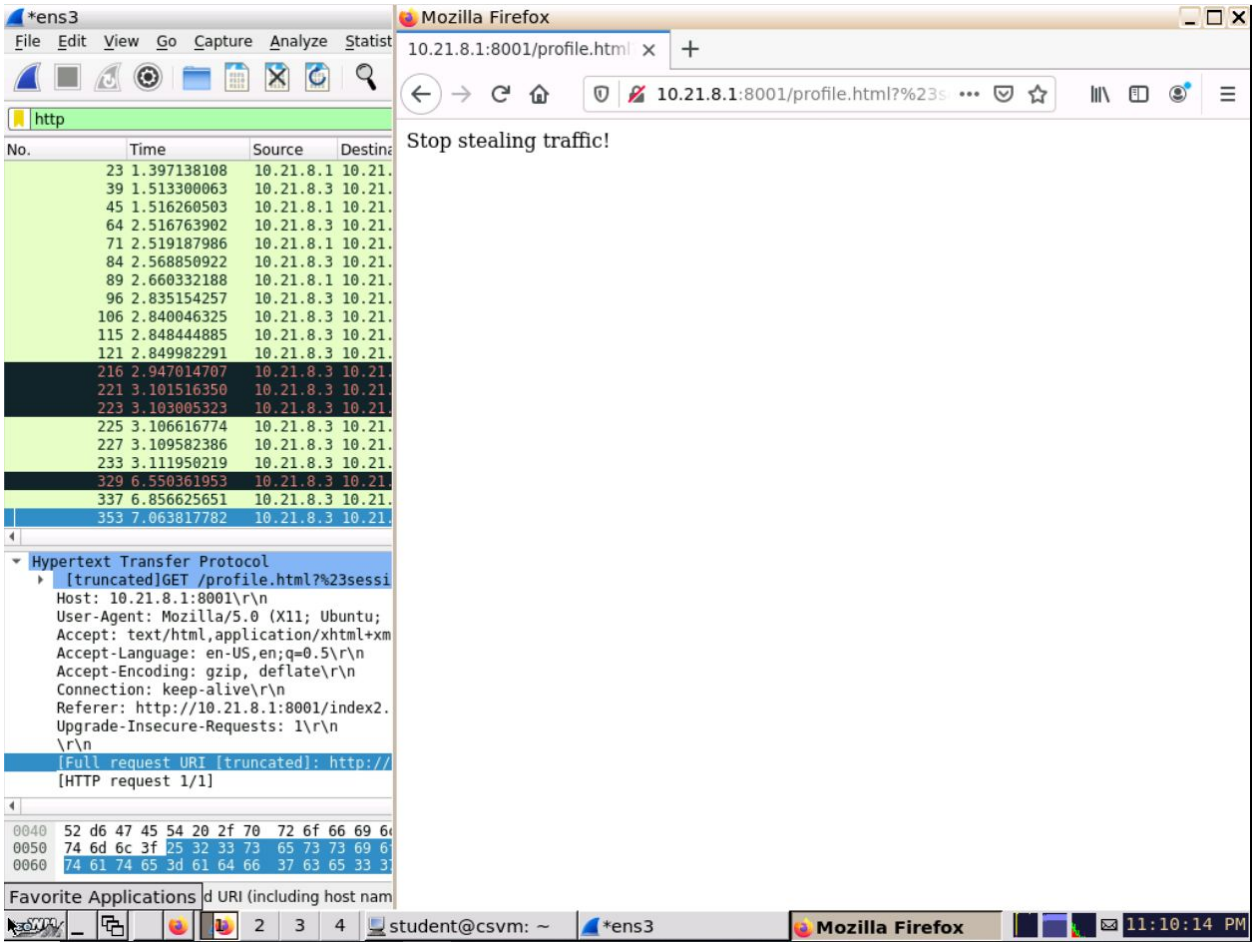
```
                    if verifyNonce(token['nonce'], self.client_address[0]):
                        self.send_response(200)
                        self.end_headers()
                        self.wfile.write(b'Hello!' + bytes(token['name'], "utf-
8'))

                        return
                    else:
                        self.send_response(403)
                        self.end_headers()
                        self.wfile.write(b'Stop stealing traffic!')
                        return
```

Screenshot 8: Verify Nonce on profile request

```
def verifyNonce(nonce, client_address):
    if nonce in nonce_repo:
        if nonce_repo[nonce] == client_address:
            nonce_repo.pop(nonce)
            return True
        else:
            #Remove the nonce anyway because someone else tried accessing it
            print("Wrong address! Expected " + str(nonce_repo[nonce]) + " but go
t " + str(client_address))
            nonce_repo.pop(nonce)
            return False
    else:
        #nonce just doesn't exist
        return False
```

Screenshot 9: Using a dictionary to verify Nonce and associated Ip Address



Screenshot 10: Attempted Replay attack after defense has been implemented