

CS 4404 Mission 1:

Attack and Defense of Shueworld Elections

By: Jonathan Hsu, Hasan Gandor, Seamus Sullivan

Introductory Background

In Shueworld, we must guarantee that a few security principles are upheld when elections take place. These are:

Confidentiality

-When a vote is cast, it should not be possible for an adversary to witness what the vote was cast in favor for, where the vote came from, and what the voter's credentials were when signing into the voting portal. If this goal is not met, an adversary can figure out who people are voting for, log in as them and change their vote, and also target/threaten voters who are not voting for the attacker's desired candidate. In order to guarantee confidentiality, cryptography and/or encryption of data should be utilized in order to mitigate risk of confidentiality being broken.

Integrity

- The voting system should be able to authenticate votes. Thus, the integrity of the voting system is preserved by disallowing ballot stuffing and voter fraud in the form of impersonation. If either of these two events were to occur, then the outcome of the election could be affected by disingenuous individuals, who could alter the result of the election. An on-path adversary with access to packet payloads could compromise the integrity of the system by altering packet data before forwarding the packets to their eventual destination. This would allow for voter fraud to occur, as well as potential ballot stuffing if there are no countermeasures to this on the server.

Availability

- In order for an election to be considered fair, it should be in the best interests of the organizers to have the means of voting be readily available to all who make themselves available to cast a vote. A Denial of Service (DoS) attack could hamper the availability of the voting platform, by disallowing users from casting their vote. This could be achieved via a Man-in-the-Middle (MitM) attack, where traffic between client and server is intercepted by the adversary, and thus the client is no longer able to communicate directly with the server in order to cast their vote.

For this scenario, we are evaluating the Man in the Middle attack method, where an attacker places themselves between the client and the server. This allows the attacker to inject false data or modify existing data. This is an important method to evaluate, as it can be used to change or edit votes, as well as inject fake votes into the system. To defend against this, we will be encrypting the username, password, and selected candidate of the voters, to prevent the adversary from altering the voting data.

Reconnaissance

Through our analysis, we determined that a MitM attack would be the most effective at affecting all 3 of the aforementioned security goals. An adversary employing such an attack would be able to:

1. Break Confidentiality:

By observing all of the incoming and outgoing traffic between the client and server, if user and voting data is left unsecured or unencrypted, it would be fairly trivial for the adversary to learn of user credentials and the candidates for which they are voting.

2. Compromise Integrity:

Once the adversary is able to place themselves as a MitM, the integrity of the voting system would be compromised if they are able to cast votes while posing as genuine voters connecting from clients. By taking voter information from HTTP requests made to the server, dropping those packets, and then sending packets with voting data altered to favor a specific candidate, an effective means of ballot stuffing and voter fraud could be carried out by the adversary.

3. Remove Availability:

The availability of the voting platform can only be guaranteed if clients are able to connect directly to the voting server. An effective on-path adversary could drop all packets that are routed through it between the client and server. The overall impact of such an attack could be mitigated by distributing servers across multiple networks,

thus allowing multiple routing options for the clients to connect to a server. However, the overall impact would only be diminished in this case, and not completely eliminated. In order for the impact of an on-path adversary to be completely removed, the adversary themselves would have to be isolated from the rest of the network, or client infrastructure would have to be built that allows the client to connect to alternate servers automatically upon timing out on a request to the affected servers.

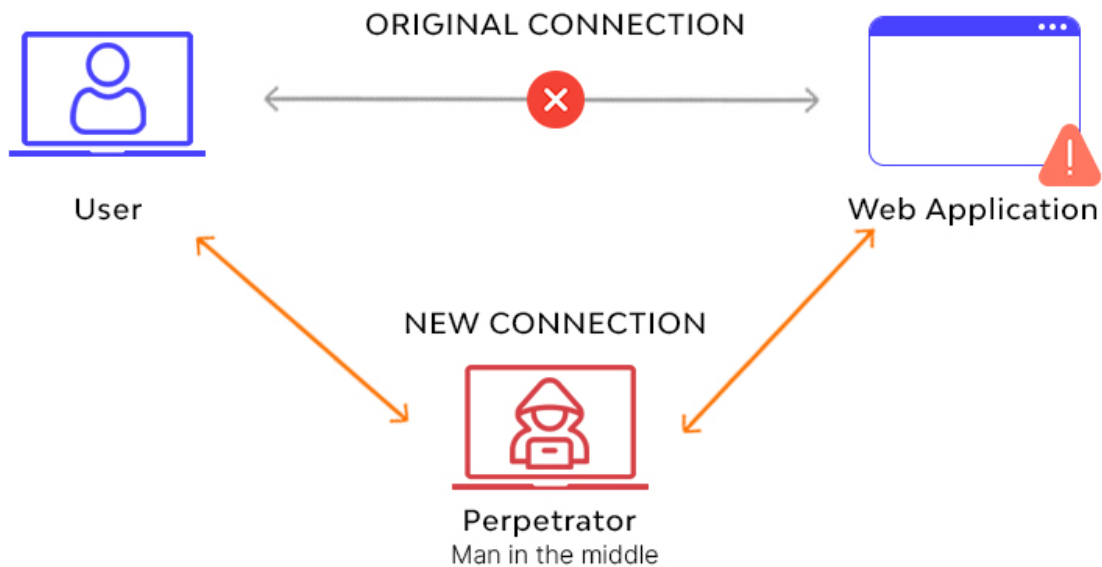


Figure 1: A diagram illustrating the structure of an on-path adversary attack (MitM)
(Source: <https://www.wallarm.com/what/what-is-mitm-man-in-the-middle-attack>)

An on-path adversary places themselves between a user (client) and the server which they are attempting to communicate with. In our Shueworld Election infrastructure, the client takes the form of a voter attempted to access the election portal, while the server is the web application that serves the voting form to the client, and tabulates voting data that is received back from the client. All

traffic between the VM 1 (client) and VM2 (server) is routed through VM3 (on-path adversary) allowing, the adversary complete control over the packets that are being passed along the network between these two endpoints.

Infrastructure Building

Client

Note: The client will only work for the un-secured version of the web server as the web page does not involve any javascript, but because the secured version includes javascript browsers such as w3m are not able to run properly. Ensure that a CLI browser such as Brow.SH, lynx, or elinks has been properly installed and configured in order to properly receive requests from the secured server.

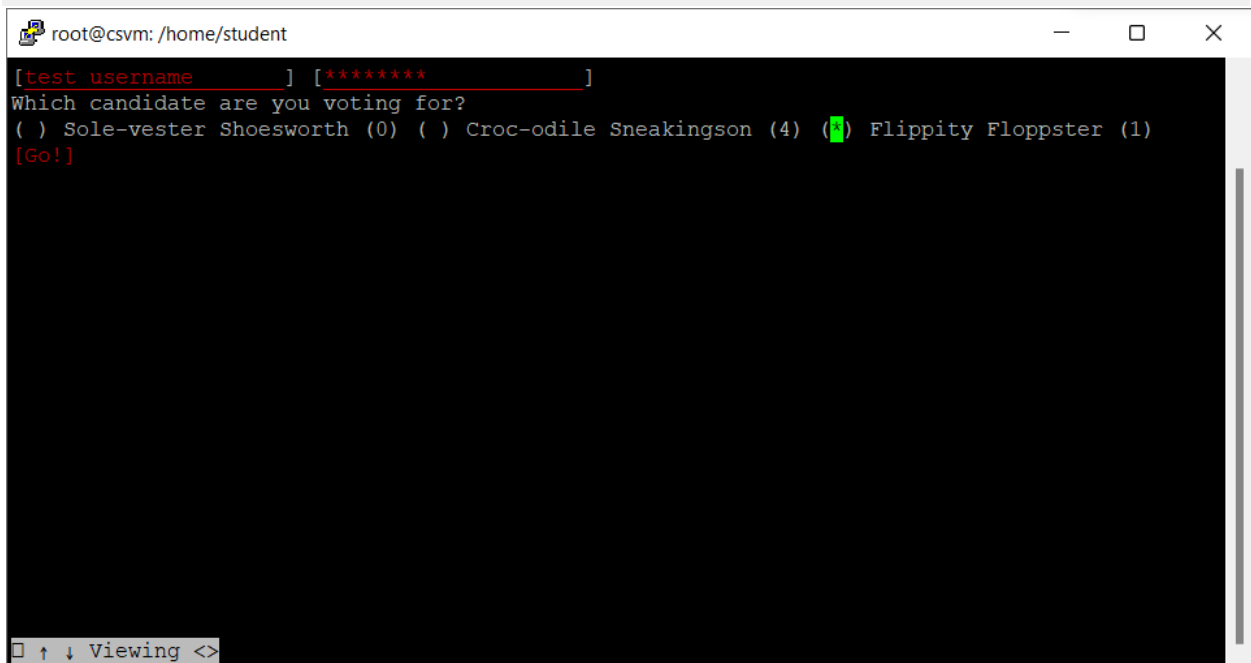
Steps to set up client (if not using built-in cURL to make HTTP requests):

w3m:

```
sudo apt install w3m w3m-img
```

To connect to web server at 10.64.13.2 on port 5000 using w3m:

```
w3m http://10.64.13.2:5000
```

A screenshot of a terminal window titled 'root@csvm: /home/student'. The terminal displays a web browser interface with a red prompt '[test username] [*****]'. Below this, it asks 'Which candidate are you voting for?'. There are four options listed: '() Sole-vester Shoesworth (0)', '() Croc-odile Sneakingson (4)', '(x) Flippity Floppster (1)', and '[Go!]' in red. The 'x' in the third option is green. At the bottom left of the terminal, there is a status bar that says 'Viewing <>'.

```
root@csvm: /home/student
[test username] [*****]
Which candidate are you voting for?
( ) Sole-vester Shoesworth (0) ( ) Croc-odile Sneakingson (4) (x) Flippity Floppster (1)
[Go!]
Viewing <>
```

Figure 3.1: Client connected to voting interface and filling out form using w3m

Server

Steps to set up web server:

1. Copy project directory to the VM

```
scp -P 8247 -r /path/to/project student@secnet-gateway.cs.wpi.edu:~/
```

2. SSH into the VM

```
ssh -p 8247 student@secnet-gateway.cs.wpi.edu
```

3. Set up Python virtual environment

- a. Create virtual environment

```
python3 -m venv app_env
```

- i. If this fails, run `sudo apt-get install python3.8-venv`

- b. Activate virtual environment

```
source app_env/bin/activate
```

4. Install Python dependencies

- a. Navigate into project folder that was copied over in step 1

```
cd project/
```

- b. Run script to install dependencies

```
bash install_dependencies.sh
```

5. Start web server

- a. Navigate to app folder (located in the project folder)

```
cd app/
```

- b. Run script to start server

```
bash start_server.sh
```

If the cryptography library is not imported, run steps 4-5 again outside of the python virtual environment. The environment can be exited by typing “deactivate”. This is because the cryptography library is already installed on the Zoo Lab VM’s.

On-path Adversary

Steps to set up on-path adversary:

1. Set up all the VM routes

Assuming VM 1 (10.64.13.1) and VM 2 (10.64.13.2) are trying to communicate while VM 3 (10.64.13.3) is the on-path adversary, the following commands should be run:

IP Forwarding

```
echo 1 > /proc/sys/net/ipv4/ip_forward - Run on VM 3
```

Static Routes

```
route add -host 10.64.13.1 gw 10.64.13.3 - Run on VM 2  
route add -host 10.64.13.2 gw 10.64.13.3 - Run on VM 1
```

Stopping ICMP Redirects:

Sending: Put a 0 in: `/proc/sys/net/ipv4/conf/*/send_redirects`

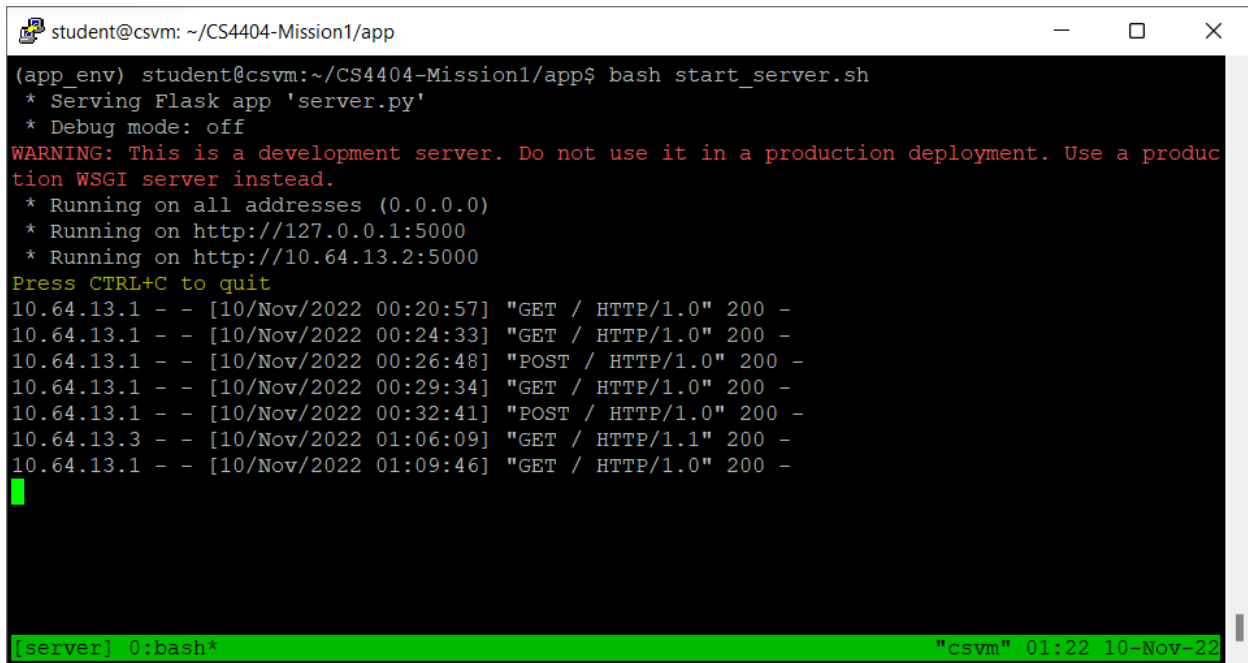
Receiving:

```
sysctl net.ipv4.conf.all.accept_redirects=0  
net.ipv4.conf.eth0.accept_redirects=0  
net.ipv4.conf.eth1.accept_redirects=0
```

Filtering:

```
iptables -A INPUT -p icmp --icmp-type redirect -j DROP  
iptables -A OUTPUT -p icmp --icmp-type redirect -j DROP
```

2. Copy python script to the selected on-path VM
 - a. Located under app/mitm.py
3. Configure the python script by setting these values:
 - a. url at the top to the target web server url
 - b. iface option in the sniff function to the correct interface (In our case “ens3”)
 - c. filter option in the sniff function to the correct port (5000)
4. Run python script



```
student@csvm: ~/CS4404-Mission1/app
(app_env) student@csvm:~/CS4404-Mission1/app$ bash start_server.sh
* Serving Flask app 'server.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.64.13.2:5000
Press CTRL+C to quit
10.64.13.1 - - [10/Nov/2022 00:20:57] "GET / HTTP/1.0" 200 -
10.64.13.1 - - [10/Nov/2022 00:24:33] "GET / HTTP/1.0" 200 -
10.64.13.1 - - [10/Nov/2022 00:26:48] "POST / HTTP/1.0" 200 -
10.64.13.1 - - [10/Nov/2022 00:29:34] "GET / HTTP/1.0" 200 -
10.64.13.1 - - [10/Nov/2022 00:32:41] "POST / HTTP/1.0" 200 -
10.64.13.3 - - [10/Nov/2022 01:06:09] "GET / HTTP/1.1" 200 -
10.64.13.1 - - [10/Nov/2022 01:09:46] "GET / HTTP/1.0" 200 -
[server] 0:bash* "csvm" 01:22 10-Nov-22
```

Figure 3.2: Running web server serving requests on the Zoo Lab virtual machine following the steps mentioned above

Attack

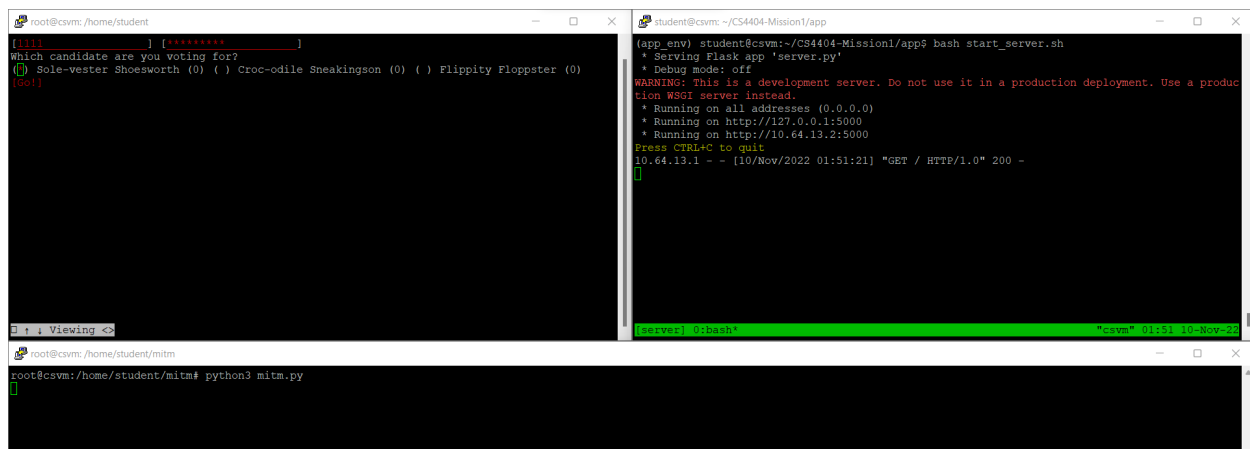
1. Vector

For our attack vector, we decided to proceed with a MitM attack on the Shueworld election infrastructure. This is because it poses perhaps the largest threat to three of the four security principles: confidentiality, integrity, and availability. In the case that election data is sent unencrypted, confidentiality and integrity of the data would be completely thwarted. The adversary would have full viewing access to the authentication fields for the users, as well as the voting data. Integrity of the data would also certainly be compromised, as the payload of the packets received through the middleman could be directly edited and passed on to the election server, thus editing the votes that people have made.

2. Setup

Our attack vector consists of a python script running on the MitM virtual machine (VM 3 in our example network). The script will allow the adversary to view all packets that pass between the client and server (VM 1 and VM 2 respectively) and view the payload data that lies within each packet. When a POST request is

made by the client to the server, the script will attempt to locate username, password, and vote data fields within the payload, and then save this data. It will then craft its own POST request in order to attempt to alter the voter's selection to a predetermined target vote after the voter has been greeted with an updated page indicating that their vote has gone through. Thus, the voter is made to believe that their vote has been tallied successfully, and will navigate away from the voting interface. However, if they were to perform another GET request, they would see that their voter credentials had been used to vote for another candidate.



```
root@csvm: /home/student
[1111] [*****]
Which candidate are you voting for?
(0) Sole-vester Shoesworth (0) ( ) Croc-odile Sneakingson (0) ( ) Flippity Floppster (0)
[001]
```

```
(app_env) student@csvm:~/CS4404-Mission1/app$ bash start_server.sh
* Serving Flask app "server.py"
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.64.13.2:5000
Press CTRL-C to quit
10.64.13.1 - - [10/Nov/2022 01:51:21] "GET / HTTP/1.0" 200 -
```

```
root@csvm: /home/student/mitm# python3 mitm.py
```

Figure 4.1: Before client sends vote request. Client (Top left) enters information but has not sent yet. Server (Top right) shows the GET request made by the client to retrieve the web page. Adversary (Bottom) has not intercepted as the vote has not been sent.

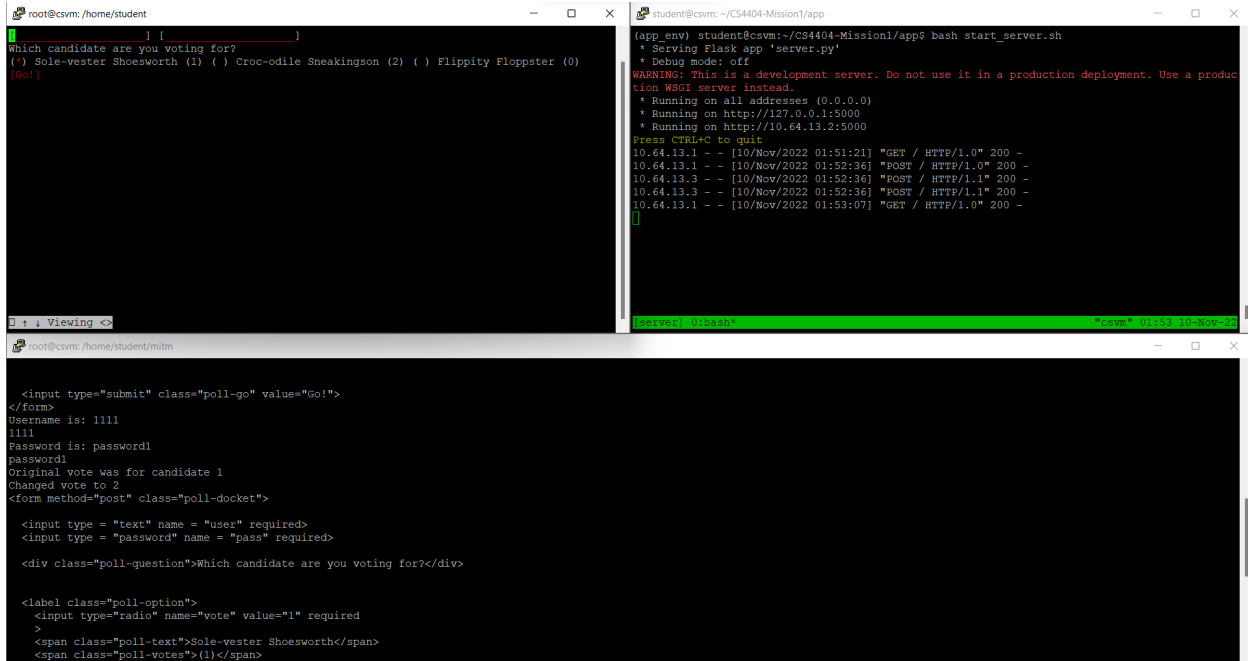


Figure 4.2: After the client sends the vote request and refreshes. Client (Top left) shows votes cast for candidate 2 despite sending a vote for candidate 1. Server (Top right) shows POST requests made by the client and adversary, as well as the GET requests made by the client. Adversary (Bottom) has intercepted the POST requests made by the client and is able to read out the information and send votes on behalf of the client.

SQL Injection Vulnerability (Brief Exploration)

While not the main focus of our attack and defense planning, it is necessary to address the possibility of an SQL injection attack when using an SQL database to manage voter and voting data. While researching such a possibility, low-level attacks such as simply fooling the server into thinking that the username and password pair sent from the client were authentic in order to count a vote could be executed if the database relied on concatenation in order to determine whether or not a username was within the list of authorized users stored.

1. An example of a insecure implementation in library.py that interacts with the database.

```
cursor.execute("SELECT * FROM accounts WHERE username = '%s' AND
```

```
password = '%s' % (username, password))
```

This type of attack is simple enough to thwart with just a single-line change, however there exist many other types of SQL injection vectors that could remain as vulnerabilities in our database.

2. Correction made to the implementation in library.py that does not allow for a “1=1” attack to be made via SQL. The corrected implementation does not use concatenation.

```
cursor.execute("SELECT * FROM accounts WHERE username = ? AND  
password = ?", (username, password,))
```

Given that this type of attack vector was not explored in detail with regards to an on-path adversarial attack, this was not focused on for the outcomes of this mission.

Defense

1. Methodology

Confidentiality can be ensured through the use of asymmetric cryptography, which allows the client to encrypt their message using the server’s public key, and when the server receives the encrypted message, it can be decrypted only with the server’s private key. This allows the message contents to remain confidential even if the encrypted message is compromised.

Integrity is somewhat ensured as well, because tampering with the encrypted data either results in gibberish output when decrypted or in errors. As a result, an encrypted message cannot be reliably modified to insert specific information an adversary wants to include.

While confidentiality and integrity can be ensured through the encryption of user data and other sensitive strings such as voting data, the nature of a successful MitM setup means that availability can still be denied by simply refusing to allow

packets to be sent between the voting client and voting server. However, by setting up multiple web servers to handle voting requests, there can be backup locations for voters to go to in case a specific voting server gets taken down. This would mitigate the compromises to availability arising from a MitM attack by allowing voters to submit their votes at other locations. It would also increase the costs of performing several MitM attacks at various locations which could serve as a deterrent for an adversary.

We will focus on disallowing the adversary from compromising the client authentication (username and password) fields and voting selection by encrypting these fields before they are sent to the server.

2. Execution

Our defense consists of using RSA encryption (an example of asymmetric cryptography) in order to hide the user and password fields as they travel from the client to the server. The public and private key pairs must first be generated, and the private key must be kept secret at all times, whereas the public key should be sent to every voting client.

When the clients are ready to submit their credentials and voting information, all their data is joined together into a single plaintext string. This plaintext is then encrypted using the public key, and the resultant ciphertext is the only information that is transmitted to the server.

When the server receives the ciphertext, it uses its own private key to decrypt the ciphertext back into plaintext. This plaintext can then be separated into each individual field to extract the username, password, and voting information.

Thus, the on-path adversary will be denied the ability to view the relevant data after they capture the ciphertext, securing confidentiality with regards to voter credentials and voting data.

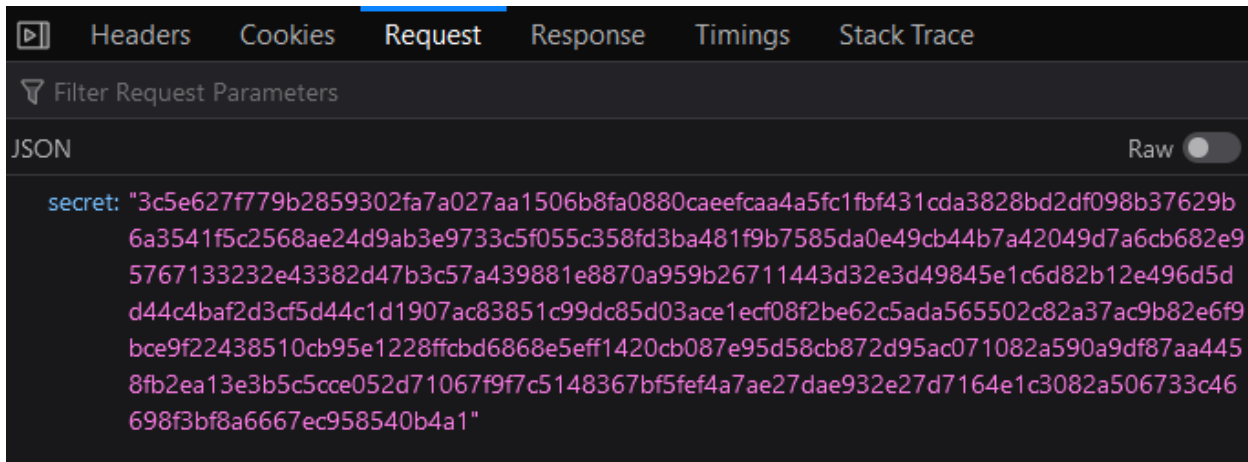


Figure 5.1: An example of the ciphertext data transmitted over the internet and to the web server using RSA encryption with the public key. It is infeasible for an on-path adversary to determine the username, password, or voting selection from this information without the server's private key

```
Decrypted plaintext:
user=1111&pass=password1&vote=1
127.0.0.1 - - [10/Nov/2022 22:18:17] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [10/Nov/2022 22:18:17] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [10/Nov/2022 22:18:23] "GET / HTTP/1.1" 200 -
Recieved secret:
07955b893e4f9008eca85edf8fe7bfb7af413f7e70bb35ef7ebf5c2b83729cc48c3f6ef3b23bf
5f7fa0fb0ba17bda68cdd6d701d1b9fdb8e8f8aac249d428ce7768d28976b169abf951d84cac34
2202ed515e15ea2b4071bc26cc4188a00fb63dd9084c4189d68daa0cb123980f82a16495b7fb8
7cae775162ef38d586866d74dd8a8d400cc7e9a0ac197de396d41b5470d89523294e1d2e84091
50c55663afa3a74ce0d847987324aab50d9d0b012d1822d654698b9e96ff68bb5c5238eaa5f88
986e99521c9a654eb7a8ee7e242b50be62af037fcdae0956cc244abc06010343990ab057a658b
d7c15973e4a8d0038bddfb02545e1c7f38b074a60a35dc1b22
Decrypted plaintext:
user=1111&pass=password1&vote=2
127.0.0.1 - - [10/Nov/2022 22:18:30] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [10/Nov/2022 22:18:30] "GET / HTTP/1.1" 200 -
Recieved secret:
3c5e627f779b2859302fa7a027aa1506b8fa0880caeefcaa4a5fc1fbf431cda3828bd2df098b3
7629b6a3541f5c2568ae24d9ab3e9733c5f055c358fd3ba481f9b7585da0e49cb44b7a42049d7
a6cb682e95767133232e43382d47b3c57a439881e8870a959b26711443d32e3d49845e1c6d82b
12e496d5dd44c4baf2d3cf5d44c1d1907ac83851c99dc85d03ace1ecf08f2be62c5ada565502c
82a37ac9b82e6f9bce9f22438510cb95e1228ffcbd6868e5eff1420cb087e95d58cb872d95ac0
71082a590a9df87aa4458fb2ea13e3b5c5cce052d71067f9f7c5148367bf5fef4a7ae27dae932
e27d7164e1c3082a506733c46698f3bf8a6667ec958540b4a1
Decrypted plaintext:
user=1111&pass=password1&vote=1
127.0.0.1 - - [10/Nov/2022 22:19:07] "POST / HTTP/1.1" 200 -
127.0.0.1 - - [10/Nov/2022 22:19:07] "GET / HTTP/1.1" 200 -
```

Figure 5.2: The updated web server log, showing the received encrypted messages and the decrypted plaintext obtained using the private key. The plaintext allows the server to fulfill the vote request as it now has the username, password, and vote selection. The last log shows the result after the data from Figure 5.1 was transmitted.

Conclusions

In this project, we analyzed the effectiveness of a man in the middle attack against a simple client-server connection. The results of this project showed us an on-path adversary was effective in an attack. All of the data sent from the client to the server went through the adversary and could be edited. The learning goal for the attack side of this project was to understand what an on-path adversary could achieve if they were to find a way to route traffic through itself in-between the client and server. For the defense method, our goal was to implement asymmetric cryptography, generating public and private keys, transferring the public key to the client so that the data received from the client could be encrypted, and subsequently decrypted by the server using its own private key.

An adversary that successfully places itself as a MitM between client and server poses a meaningful threat to the aforementioned security goals. Since all data is routed directly through the adversary, it could choose from many methods of attack, be that stealing user information passively through observing packet data and then forwarding the packets, or taking an active approach in compromising the integrity of the data that is being sent by altering it. An even more extreme attack vector would see the adversary drop all packets between client and server, thereby completely denying access to web services. Our attack vector was able to successfully accomplish the first two methods of attack, printing out user credentials as they were observed through the POST requests, as well as the candidate for which they cast their vote. The second attack method was also carried out successfully with the adversary then using the user credentials it had scraped from the packets to alter the client's vote after they had cast their ballot. While we did not fully explore the last attack vector of dropping all packets, this would have been a fairly trivial task to accomplish, as there would be no way for the

client to reach the server given the setup that had been accomplished with the virtual machines.

The defense methodology to thwart these the attacks on confidentiality and integrity of user credentials and voting data was to encrypt said data. To do so, we included RSA encryption in the implementation of our voting server. The updated version includes embedded JavaScript in the HTML pages served to the client, along with a public key that it is to use to encrypt the data. As the data is encrypted on its way back to the server, the on-path adversary is unable to decrypt and parse the voting data and user credentials, thus preserving confidentiality. In addition to this, integrity can again be guaranteed, as the web server simply reject any attempts that the on-path adversary makes to alter votes, as its attempts to scrape voter credentials need to cast a ballot would result in either complete failure, or it sending a gibberish response to the web server when prompted for credentials. After the web server receives the information back from the client, it is then able to decrypt the client's data and vote using its own private key. Through our exploration of this defense method, we learned how to effectively implement cryptography into the client-server interaction of our infrastructure, including generating public-private keys, and their appropriate usage in encryption and decryption.