Imam Khomeini International University

Faculty of Engineering

Department of Computer Engineering

B.S. Thesis

DNS Spoofing to Compromise Web Activity

Supervisor:

Dr. Hamidi

By:

Hamed Nazarian

Apr 2023

TABLE OF CONTENTS

1. Introduction and Research Objectives

My research project involved an exploration of the vulnerabilities inherent in the DNS and SSL protocols, which can be exploited by an attacker to manipulate a target's connection with a web server, thereby spying on the user and even manipulating the received data from the web server to facilitate further attacks. I adopted an offensive approach, resulting in the development of a Linux-based software capable of executing this attack. This software requires root privileges to effect changes on the target system and initiate the attack. Multiple attack scenarios can be envisaged, including social engineering of the target to run the script, the weaponization of legitimate files or applications with the script, or even the combination of this malware with other malwares to facilitate a sophisticated, multi-stage attack.

While my research project does not address the initial compromise vector and malware delivery, it provides a step-by-step overview of the script's operation and the related technologies. The developed software can manipulate the "/etc/hosts" file to map an arbitrary URL to the attacker's server, and by acting as a man-in-the-middle, it can take full control of the victim's web browsing. Additionally, the software can get around SSL and HTTPS by using a self-signed certificate.

2. Key Concepts and Terminology

2.1 DNS and DNS Resolution Process

To access web services, we rely on domain names that are resolved to the corresponding server's IP address. This process of obtaining the IP address of a server from its domain name is referred to as DNS resolution. DNS resolution works in a hierarchical manner, starting with a check of local DNS records to see if there is a static record mapping the domain name to an IP address. If such a record is not found, the system performs a lookup using an external DNS server.

On a Linux platform, the file responsible for providing static DNS records is the well-known "/etc/hosts" file. Through the manipulation of this file, a system administrator or an attacker can cause the system to look for a website at an arbitrary location (IP address). Such an attack can be accomplished by mapping an

arbitrary URL to the attacker's server and, by acting as a man-in-the-middle, taking control of the victim's web browsing. The attacker can further exploit this attack to manipulate the received data from the web server on the victim's browser to perform additional attacks.

2.2 SSL and HTTPS

The prevalence of cyber attacks has made it essential to use cryptographic technologies to safeguard online communication in the public internet against attackers, governments, and other unauthorized entities. These technologies are used to encrypt internet traffic, thereby preventing attackers from eavesdropping or manipulating and tampering with the data transmitted over the internet.

In the realm of web communication, SSL (Secure Sockets Layer) technology, and in newer versions, TLS (Transport Layer Security), are used to secure internet traffic. When clients initiate communication with a web server, they go through a process known as the "TLS handshake," during which the web server provides them with an SSL certificate. These certificates are issued by third-party entities known as "CAs (Certificate Authorities)," which are globally trusted.

It is worth noting that individuals can also create their own certificates, known as self-signed certificates. However, when systems encounter such certificates, they often display a warning message asking users if they wish to proceed. This is because self-signed certificates are not verified by a trusted third-party entity like CAs, and therefore their authenticity cannot be fully guaranteed.

3. Technical Details and Methodology

The script developed in this research is a 250-line Python code designed to be run on the command line of a Linux host with root privileges. The script primarily employs native Python libraries, with the exception of the "requests" library, which can be easily downloaded using the "pip" Python package manager utility. It also makes use of the "openssl" Linux package, which can be acquired from official repositories in linux systems.

3.1 Script Design and Implementation Overview

3.1.1 Modules and Libraries

The script utilizes a variety of modules and libraries, including Python's native libraries and other specialized network and HTTP modules. These include:
"http": used to set up a web server
"requests": acts as an HTTP agent
"ssl": used to implement SSL and HTTPS certificates
"urllib3": for DNS resolution
"socketserver": used as a base class for a custom "request handler"

Operating system integration libraries, such as "os" for system APIs, "threading" to implement threading, and "signal" to use system processes APIs

General libraries, such as "shutil" for some file system functions, re for regular expressions, and "argparse" to define and use command-line arguments.

3.1.2 Global Functions

The script also utilizes several global functions, which are called in the main section to perform specific tasks. These functions include:
"sigint_handler()": rewrites the handler function when encountering the INTERRUPT signal (Ctrl + C)
"create_host_files()": creates a self-signed SSL certificate by executing a system command and stores it inside the domain name's working directory. This function uses the openssl Linux package and runs the following command: openssl req -newkey rsa:4096 -x509 -sha256 -days 3650 -noenc -out "selfsigned.pem" -keyout "selfsigned.key" -subj "/CN=example.com"
"delete_files()": executed upon exit and removes the current target's working directory entirely, including SSL certificates
"etc_cleanup()": executed upon exit and cleans up added DNS records from "/etc/hosts" by running a system command
"etc_update()": adds static DNS records to "/etc/hosts" file by running a system command.

3.1.3 Custom Request Handler Class

To set up the HTTP (web) server, the script utilizes Python's standard native http module. However, to implement the required functionality to masquerade and spoof legitimate websites, a custom request handler is defined. This is the data object responsible for handling each HTTP request made by the user. Most of the functionality is inherited from the socketserver module's BaseHTTPRequestHandler, with only a few specific methods like do_GET, do_POST, and a few more being overridden.

3.1.4 Server Setup Handler

This is a simple wrapper to execute the necessary setup for the server. Its main job is to create a server object by initializing the http module's HTTPServer class and passing in the binding address and custom request handler class to call upon receiving a request.

3.1.5 Main

The main section of the code is executed first. It starts off by setting up the command-line argument parser, initializing global variables by parsing and extracting passed arguments, and then starting the attack by calling the necessary global functions and setting up the server.

3.2 Demo of Script's Functionality and Workflow

Running script with "-h", or "--help" arguments prints out a help menu.

```
└─# ./http_browser_proxy.py --help
usage: http_browser_proxy.py [-h] [-p PAYLOAD] [-d SPOOFED_ADDR] url

An http proxy, which works by changing local dns records at '/etc/hosts', then
creates local web server to server users rouge version of website.

positional arguments:
  url                   A local or external website address (This could be a domain
                        name or ip address)

options:
  -h, --help            show this help message and exit
  -p PAYLOAD, --payload PAYLOAD
                        Payload to inject into victim's html page. (By default
                        beef-xss hook is used, use empty string to desable it.)
  -d SPOOFED_ADDR, --destination SPOOFED_ADDR
                        Destination or Spoofed host address to redirect users to
                        (used for mapping to domain name, defaults to localhost)
```

Figure 3.2. 1: Command-line help menu

Also running script without any argument will return back an error, printing out the basic usage.

```
└─# ./http_browser_proxy.py
usage: http_browser_proxy.py [-h] [-d SPOOFED_ADDR] url
http_browser_proxy.py: error: the following arguments are required: url
```

Figure 3.2. 2: Error message asking for mandatory arguments

One of the key features of the script is its ability to handle a wide range of URL formats. To achieve this, a complex regular expression (regex) is used to extract the relevant information from the URL, such as the domain name and protocol

(http, or https). This ensures that the script can accurately identify the target domain name to spoof.

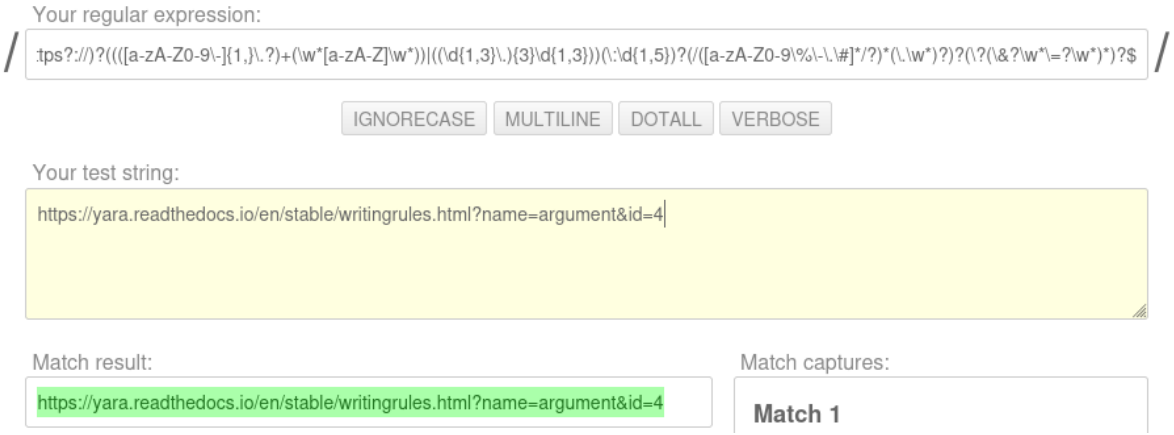

Figure 3.2. 3: Show case of custom, complicated regex

After parsing the URL, the program proceeds to add two DNS records to the "/etc/hosts" file on the target system. These DNS records map the provided URL (both with and without the www prefix) to the IP address of the attacker's server(By default local host). This manipulation ensures that any subsequent requests to the target URL are redirected to the attacker's specified server, rather than the intended web server. By doing so, the attacker can intercept and modify the traffic to and from the target server. It is worth noting that modifying the "/etc/hosts" file requires root privileges, which is why the script needs to be executed with elevated privileges.

```
61  def etc_update():
62    # Update /etc/hosts file
63    for host in host_list:
64      shell_command = f"""sudo sh -c 'echo -n "\n{spoofed_addr} {host["host"]}">>/et
65      os.system(shell_command)
66      shell_command = f"""sudo sh -c 'echo -n "\n{spoofed_addr} www.{host["host"]}">
67      os.system(shell_command)
```

Figure 3.2. 4: Implemented function to add static dns records by running system command

The next step in the script is to set up local webservers on ports 80 and 443. This is accomplished using Python's built-in HTTP server library, which allows the script to serve web pages on those ports. Since the spoofed website is now mapped to the localhost, any requests made to that website will now be sent to the local webservers.

```
267     threading.Thread(target=run, kwargs={"port":443, "https": True}).start()
268     threading.Thread(target=run).start()
```

Figure 3.2. 5: Run server instances for http and https in the background using threading

For HTTPS traffic, the script generates a self-signed SSL certificate using the openssl command-line utility. This certificate is then used by the script to enable SSL/TLS encryption on the local webserver. This allows the attacker to intercept and decrypt any traffic sent over HTTPS to the spoofed website. Since the certificate is self-signed, however, web browsers will display a warning to the user that the certificate is not trusted.

```
34  def create_host_files():
35    # Create self-signed ssl certificates
36    for host in host_list:
37      os.mkdir(host["dir"])
38
39      shell_command = f'''openssl req -newkey rsa:4096 -x509 -sha256 -days 3650 -no
40      os.system(shell_command)
```

Figure 3.2. 6: Implemented function to generate certificate using openssl system utility

The use of self-signed certificates is not uncommon in the context of malicious attacks, as they provide a means for attackers to masquerade as legitimate entities and thereby deceive victims into trusting the authenticity of the connection. However, in legitimate contexts, the use of self-signed certificates can raise suspicion and trigger security warnings in modern browsers.

In order to effectively impersonate a legitimate website, the program must retrieve the actual website content to respond to victim requests. To accomplish this, a custom request handler class was defined, which is invoked upon receiving a request. The requests library is then used to send an authentic request and retrieve the legitimate webpage or server response. By obtaining the authentic response, the program can manipulate it and subsequently send it back to the victim.

```
71  class My_Class(BaseHTTPRequestHandler):
72    protocol = None
73    certfile = None
74    keyfile = None
75
76    def __init__(self, *args, **kwargs):
77      self.protocol = "{}://".format(My_Class.protocol)
78      self.host = None
79      self.request_respond = None
80      super(My_Class, self).__init__(*args, **kwargs)
81
82    def send_response(self, code, message=None):
83      self.log_request(code)
84      self.send_response_only(code, message)
85
86    def _set_headers(self, status_code, headers):
87      self.send_response(status_code)
88
89      # headers_filter_list = ["Content-Encoding", "Transfer-Encoding", "content-l
90      headers_filter_list = ["Content-Encoding", "Transfer-Encoding", "content-len
91      for header in headers_filter_list:
```

Figure 3.2. 7: Custom "request handler" class

My script can be leveraged to spy on victim's online activities, harvest sensitive data and credentials, and even inject malicious payloads. By utilizing the payload argument, a wide range of attacks can be carried out, including the injection of HTML entities, preferably JavaScript, into the victim's browsing session. This capability provides the attacker with the flexibility to integrate a host of powerful hacking tools into the script, such as the "beef-xss" framework and other social engineering toolkits.

A common application of this program could involve creating a fake banking page that redirects unsuspecting users to it while they are browsing an ecommerce site and attempting to make a payment.

```
106   def set_html(self, message=None):
107     if message and b"<html" in message:
108       # For default payload "beef-xss" is used
109       self.html_page = message.replace(b"<head>", My_Class.payload)
110     else:
111       self.html_page = message
```

Figure 3.2. 8: Payload injection point

3.3 Testing the Script:

3.3.1 Testing script on a target:

A target website (digikala.com) is specified to be spoofed in the victim's system.

```
┌──(root㉿kali)-[/home/hm/lab/project/active/software project]
└─# ./http_browser_proxy.py digikala.com
```

Figure 3.3. 1: Running script with "digikala" as target

When the victim attempts to open the website using https, a warning is displayed. This warning would not occur if the victim were to use http.

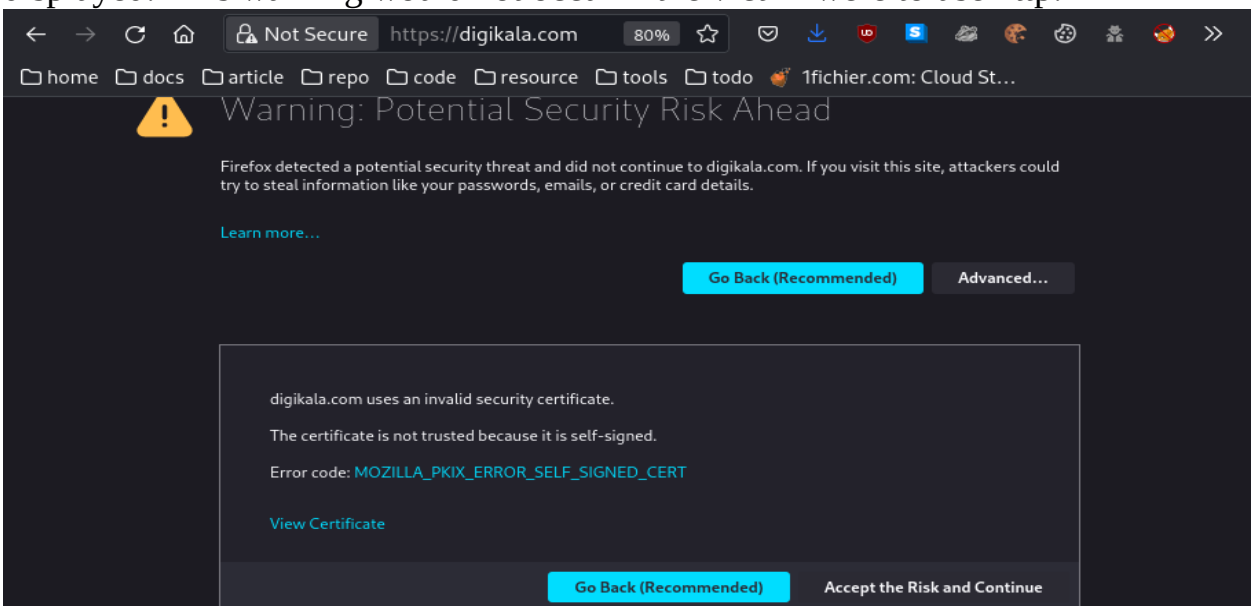

Figure 3.3. 2: Firefox picking at presented self-sign certificate

By accepting the risk (as many users do), the website will load properly.

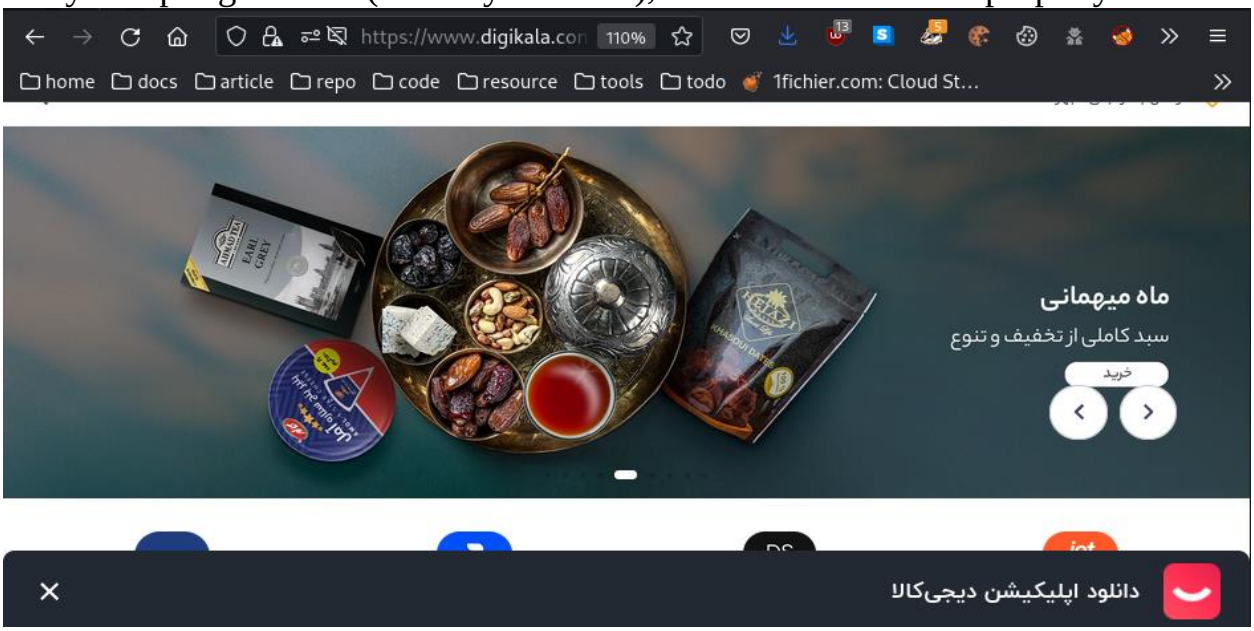

Figure 3.3. 3: Spoofed Digikala.com

As the spoofed website loads, we can see https requests being made to the fake webserver and being handled.

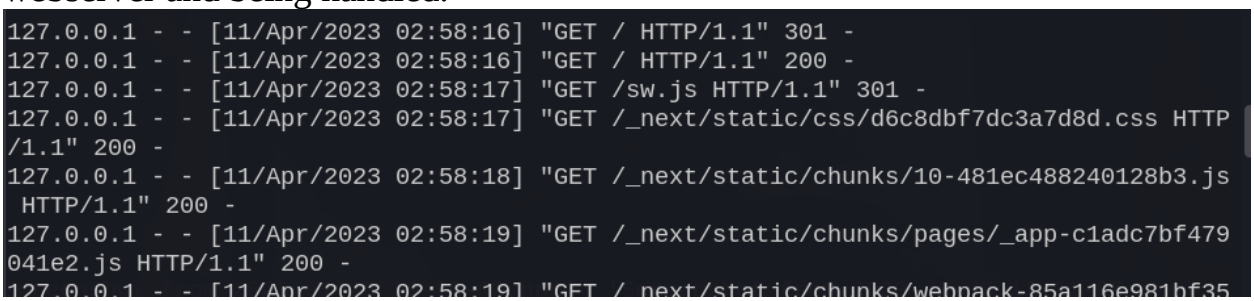

Figure 3.3. 4: Webserver logs (displayed as output by default)

3.3.2 Testing Network Route to Spoofed Domain Name:

Using the "traceroute" tool on Linux, the network path to digikala.com is shown to be only one hop, which is the local server.

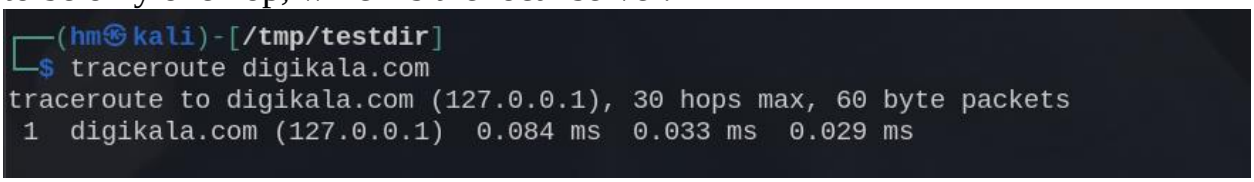

Figure 3.3. 5: "traceroute" output

The "ss" command, a Linux utility that replaces the deprecated "netstat" command, is used to list network sockets on the system and to confirm that the http server is running.

Figure 3.3. 6: "ss" output

4. Conclusion

In this research project, I focused on exploring the practical implications of DNS technology and how it can be used in conjunction with a malware on a Linux platform to manipulate the "/etc/hosts" file. This allowed for the mapping of an arbitrary URL to the attacker's server and subsequently enabling them to act as a man in the middle and take full control of the victim's web browsing. Furthermore, I demonstrated how it is possible to bypass SSL and HTTPS security measures by using a self-signed certificate.

4.1 **Achievements**

One of the key accomplishments of this research project was the creation of a highly practical and applicable script for real-life attack scenarios. The developed script is highly flexible and can be easily integrated with a range of existing hacking tools to enable various attack vectors.

4.2 **Takeaways**

While this research project did not focus on defense mechanisms, there are a few key takeaways that can help protect against these types of attacks. Firstly, users should always attempt to access websites using HTTPS, and they should never accept self-signed certificates. Additionally, since this specific implementation of the attack is executed locally on the system using a malware, it demonstrates importance of avoiding to execute any suspicious software. Another extension of this attack is by performing a man in the middle attack at some point in the network (e.g. Arp spoofing attack in local network (LAN) and acting as a DNS server), however, using VPNs can render network-base man in the middle attacks obsolete.