Tricks for Reliable Split-Second DNS Rebinding in Chrome and Safari

Tricks for Reliable Split-Second DNS Rebinding in Chrome and Safari - Author not stated, intruder.io.

- Published: date not stated
- Original: <https://www.intruder.io/research/split-second-dns-rebinding-in-chrome-and-safari>
- Preserved from: https://www.intruder.io/research/split-second-dns-rebinding-in-chrome-and-safari (stored) on 2026-08-14
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

*This is the second post in a two-part series on DNS rebinding. ***The first post** covered a real-world exploit using DNS rebinding against our own product. In this post, I introduce new techniques for achieving reliable, split-second DNS rebinding in Chrome, Edge, and Safari when IPv6 is available, as well as a technique for bypassing the local network restrictions applied to the fetch API in Chromium-based browsers. This post assumes you have a basic understanding of DNS rebinding, as covered in the previous post.*

DNS rebinding in browsers has traditionally been seen as a way for attackers to access internal network services by tricking victims into loading a malicious website, but with many modern web applications now driving headless browsers for part of their functionality, it's become a useful tool for attacking web applications. In the previous post, I covered an example of this using perhaps the simplest method for rebinding. In that scenario, I had the luxury of a long time for the exploit to run, but this is unlikely to be the case on many web applications, where faster techniques are necessary.

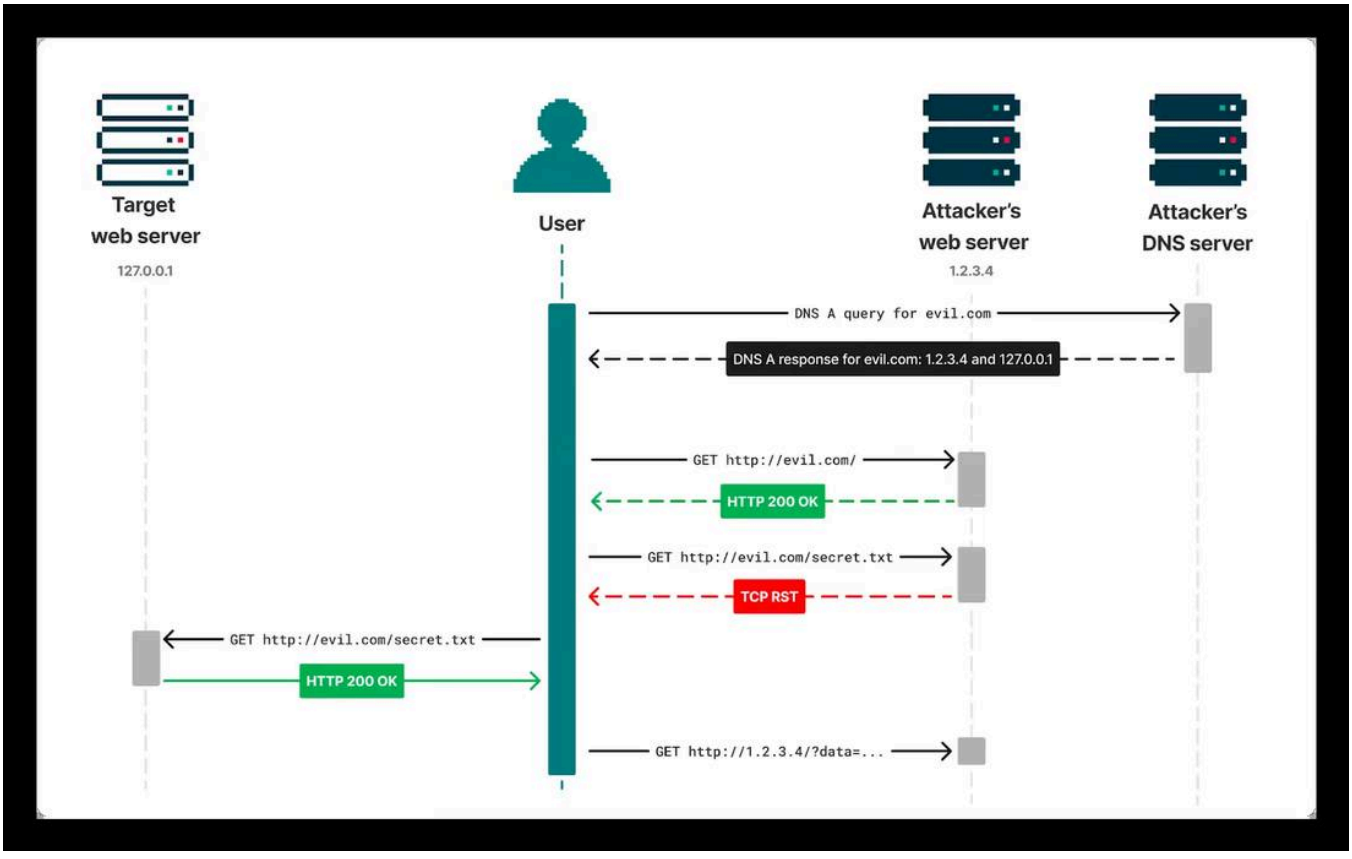
Slow Caches

Simple DNS rebinding techniques rely on returning different DNS records for successive lookups of the same hostname. For these attacks, the minimum time taken is the time between two successive DNS lookups being performed by the browser. This can sometimes be sped up by flushing the browser cache - generating a large number of DNS lookups to fill up the available cache space, and causing older entries to be discarded before they've expired - to cause the browser to perform a second lookup of the same hostname sooner.

When this works, it will still take somewhere in the order of 10 seconds, and often this technique won't work because of intermediate caches which can't be cleared as easily as the browser's cache. For example, during testing I found that on a freshly created Ubuntu EC2 instance I would only be able to get a different response for the same domain every 5 minutes due to the cache of *systemd-resolve*. On a VPN, I've seen DNS responses being cached for a minimum 30 minutes on the default resolver. It will often be a struggle to get users to keep a page open for this long to allow you to pull off a DNS rebinding exploit, let alone a headless browser being driven as part of a web application.

To speed up exploits, in 2010 Craig Heffner presented the idea of performing DNS rebinding by replying with multiple A records for the same domain in the same response, a technique which was used by Gerald Doussot and Roger Meyer in singularity in 2019. The 2 records returned are the IP address of a public, attacker-controlled server and the (usually private) IP address of the target server.

The attack will only work if the browser attempts to communicate with the public server first and loads the attacker's malicious page. The attacker's web server then starts blocking traffic from the victim's browser, causing the browser to fall back to sending all future requests to the target server. In this case, JavaScript in the attacker's page will be able to send requests to the target IP address under the same origin.



This technique does bypass the caching problem as the browser only has to perform one DNS lookup, though during my testing all the major browsers would consistently attempt to communicate to private IP addresses before public ones, meaning these techniques didn't work. While I don't believe this behaviour is intended as a protection against DNS rebinding (I even got

confirmation of this from one of the Chrome developers when informing them of my findings), it is effective at stopping this technique.

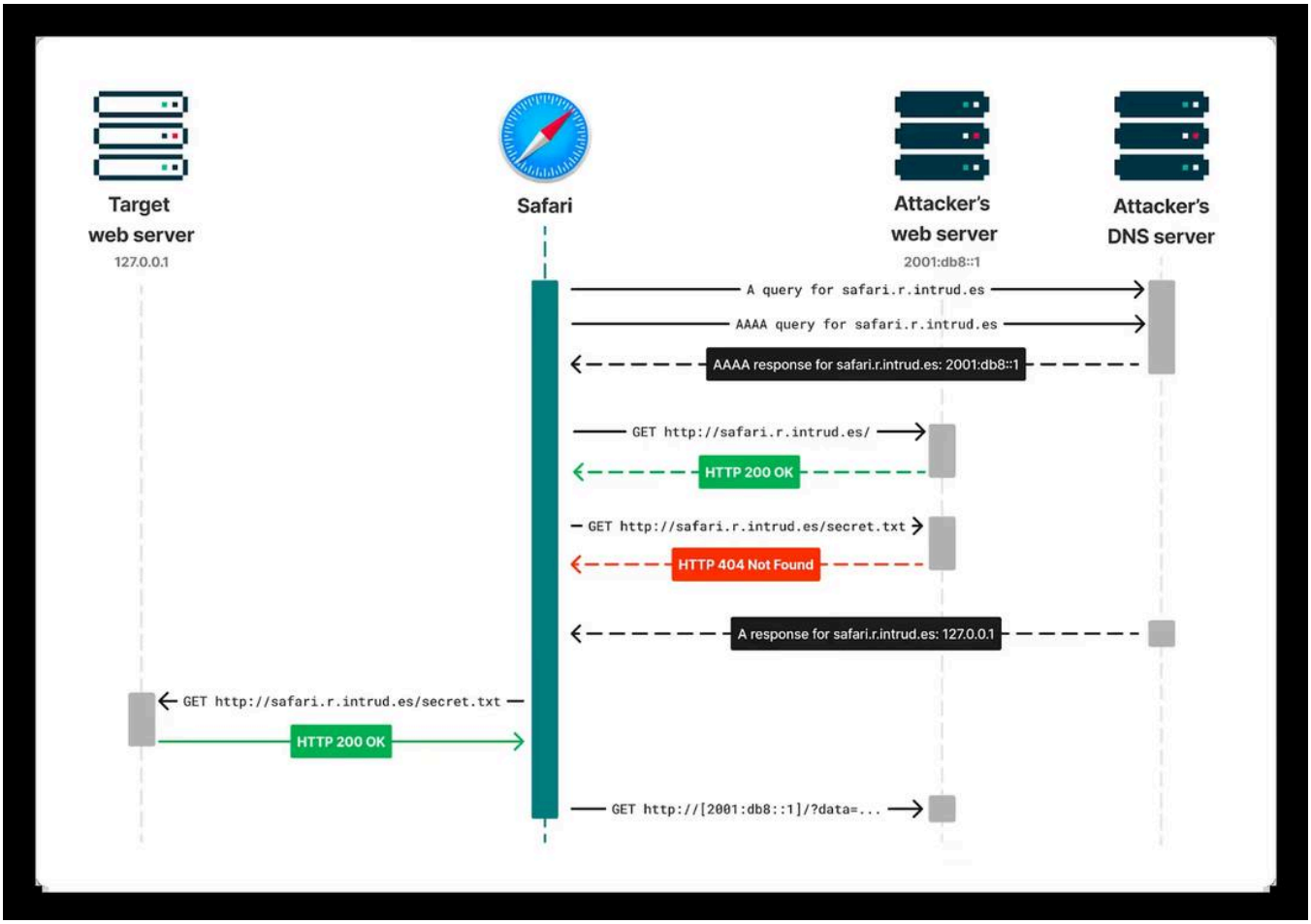
This new behaviour led me to research new techniques which can be used to achieve split-second DNS rebinding in Safari and Chromium-based browsers. The key to these techniques is finding new ways to make browsers initially use a public IP then switch to using a private IP when loading a website. Opening up Wireshark, I noticed that while loading websites in modern browsers, both an A and a AAAA query are sent. I started investigating whether this behaviour could be used to reliably perform DNS rebinding.

Attacking Safari: Delaying DNS Responses

When you load a web page in Safari on a host with access to the internet over IPv6, A and AAAA DNS queries are sent for IPv4 and IPv6 addresses respectively. Safari will prioritise private IP addresses over public ones when multiple IP addresses are returned.

The interesting behaviour which allows for fast DNS rebinding in Safari occurs when either the A or AAAA response is delayed. In this case, Safari doesn't wait for all DNS responses, but instead sends HTTP requests as soon as the first DNS response is received. When the delayed DNS response is received, the IP addresses in this response are added to the pool of IP addresses that Safari can use for future requests to the domain.

This means that if the first DNS response is for a public IP address, and the delayed DNS response is for a private IP address, Safari will send the first requests to the public IP address until the delayed DNS response is received, at which point it will start sending requests to the private IP address.



This provides a simple method for achieving DNS rebinding in Safari using a custom DNS server which handles queries for **.r.intrud.es*:

- Make the target browser load *http://safari.r.intrud.es*, triggering A and AAAA lookups for *safari.r.intrud.es*.
- Have the DNS server return a AAAA record instantly, containing the IPv6 address of an attacker-controlled web server on the internet. Do not return the A response yet.
- Safari will make the first request to the attacker's web server once it has received the AAAA response. From the attacker's web server, return a page with JavaScript to repeatedly request *http://safari.r.intrud.es/secret.txt*.
- Send the A response from the DNS server containing the IP address of the target server on the local network.
- Safari will now send the requests for *http://safari.r.intrud.es/secret.txt* to the target server on the local network. The responses of these requests can be read by the page loaded from the attacker's server without violating the same-origin policy.

To achieve this, I've written a small DNS server which can be used to delay DNS responses with command-line arguments. In practice, I found that delaying the A response by 100ms was almost always enough, though a delay of 200ms or more can be used to make the technique even more reliable. This server, as well as instructions for setting it up, can be found here.

The PHP script I've used to exploit this will redirect the user to a random subdomain of r.intrud.es to avoid intermediate caches interfering with the exploit. It also includes the JavaScript directly in the page to avoid another resource load. You can find the code used here.

Here's a video of this script in action, retrieving the contents of a file from a local web server:

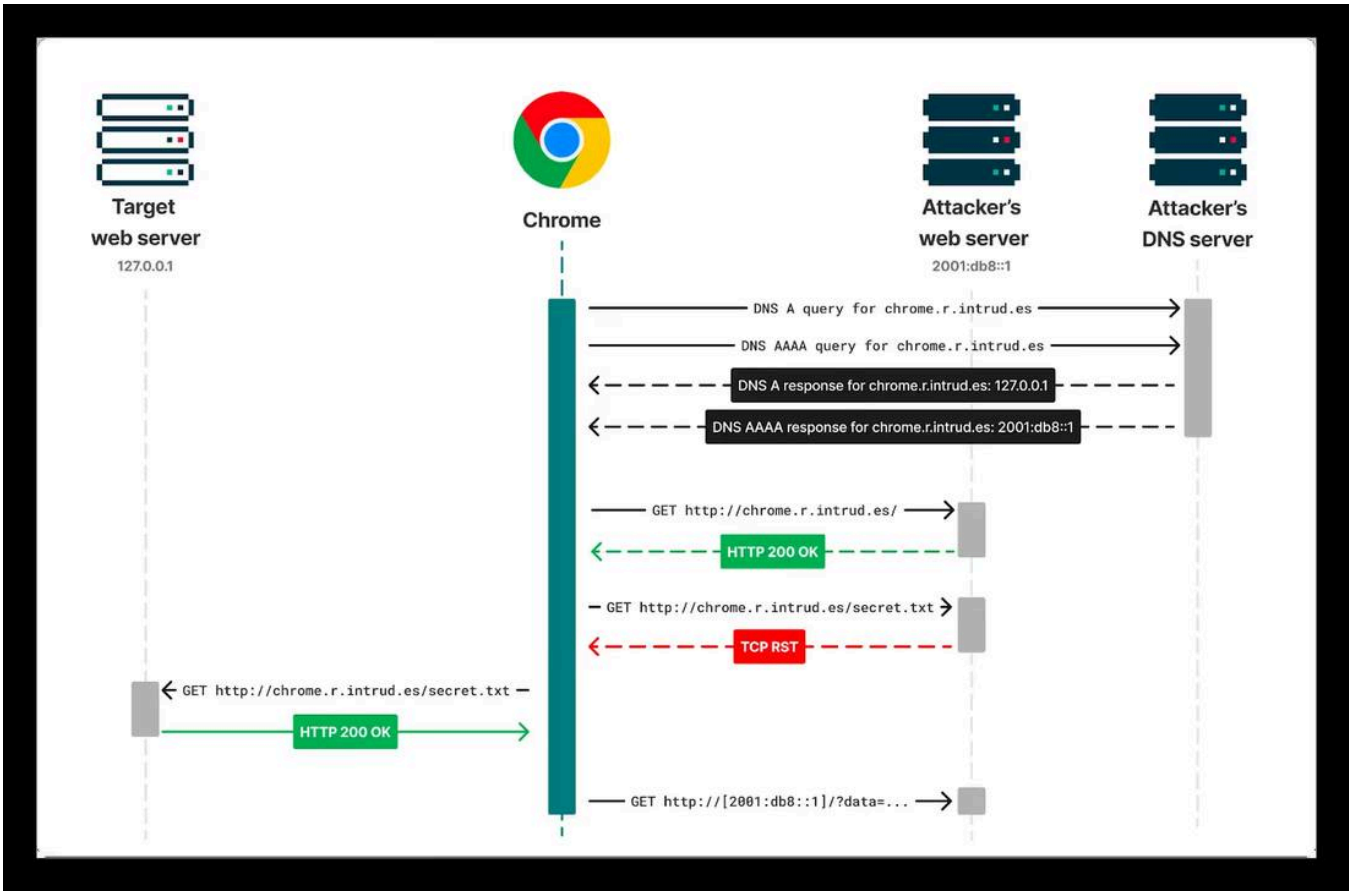
I've tested in Safari and Brave on iOS, and found that this same technique will work to access services on the internal network.

Attacking Chrome: Using AAAA Prioritisation

Chrome will prioritise loading pages on the local network over pages on the internet, but it gives more priority to loading pages over IPv6 instead of IPv4 when available. So the priority is:

- Local IPv6 (Highest Priority)
- Public IPv6
- Local IPv4
- Public IPv4 (Lowest Priority)

The key part here is that Chrome will prioritise a public IPv6 address over a private IPv4 address. Further, when Chrome knows multiple IP addresses for a domain, it will try a different IP address as soon as the server at one resets the connection.



This gives a plan for fast DNS rebinding against Chrome:

- Load *http://chrome.r.intrud.es* which will trigger A and AAAA lookups for *chrome.r.intrud.es*.
- Have the DNS server return an A record pointing to the target web server on the local network, and a AAAA record pointing to an attacker-controlled web server on the public internet.
- Chrome will prioritise the IPv6 address, and make the first request to load the page from the attacker controlled web-server, which returns JavaScript to repeatedly make requests to *http://chrome.r.intrud.es/secret.txt*.

- Shut down the attacker-controlled server so that all connections are reset. Chrome will now make all requests to the target server on the local network.
- Have the loaded page make requests to *http://chrome.r.intrud.es/secret.txt*. The responses to these requests can be read without violating the same-origin policy.

This plan almost worked. The JavaScript on the page loaded from the internet attempted to make requests to the target on the local network, but these requests were blocked with the following error in the console:

```
Access to fetch at 'http://chrome.r.intrud.es/secret.txt' from origin 'http://chrome.r.intrud.es' has been blocked by COR
```

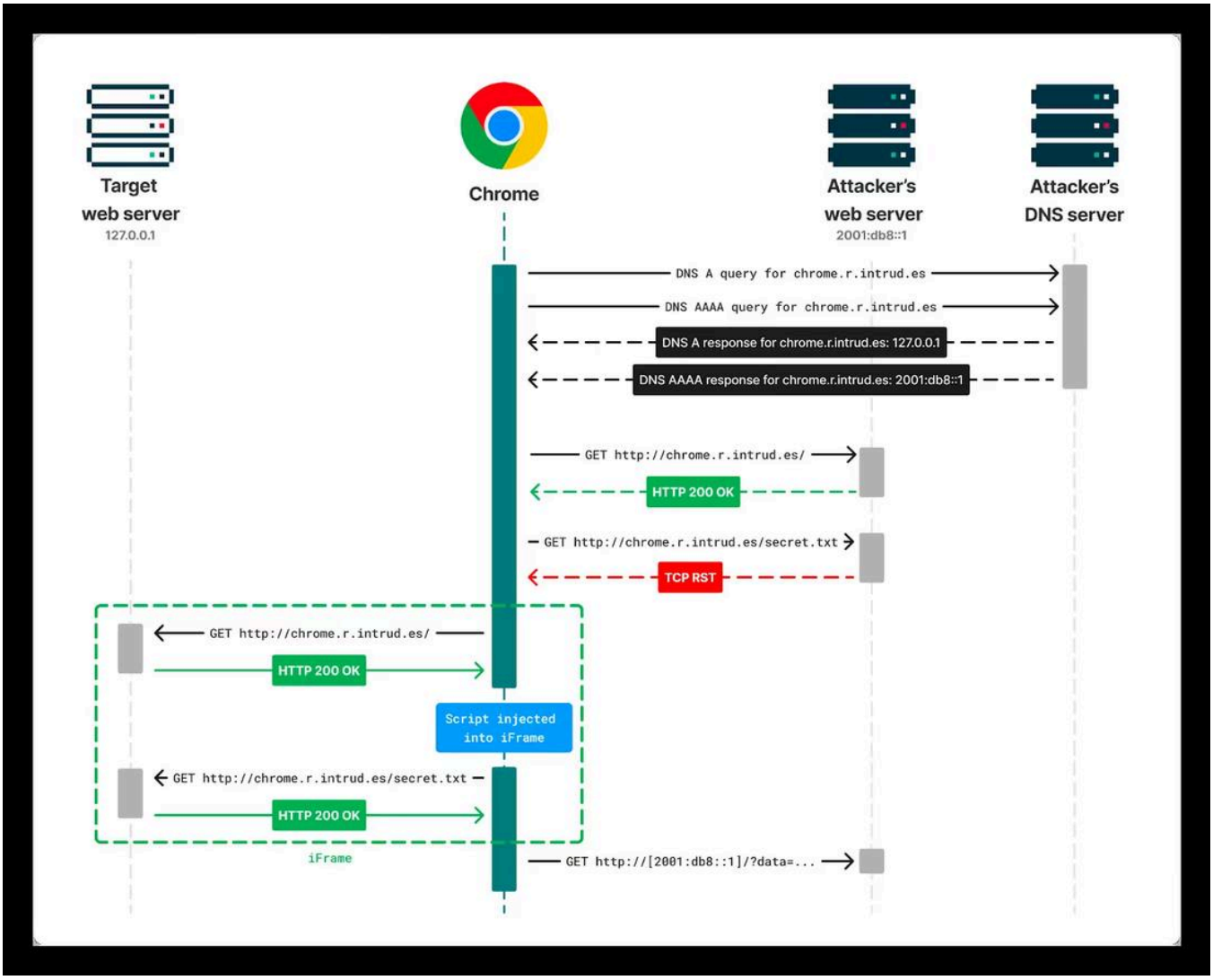
This error occurs because Chrome partially implements the protections outlined in the Private Network Access (PNA) specification.

Bypassing Private Network Access

The PNA protections block pages loaded over plain HTTP from the public internet from making requests to the private networks. In Chrome, these protections are implemented for fetch requests, but aren't yet implemented for iFrames. The incomplete implementation, along with DNS rebinding, allows the PNA restrictions on fetch requests to be bypassed.

We can repeat the exploit through to step 4 above, where the public web server has been shut off and all requests to *http://chrome.r.intrud.es* are now directed to the target server on the local network. The loaded top page can't make requests to the local network as it was loaded over HTTP from the public internet, but we can load *http://chrome.r.intrud.es* in an iFrame. The page in this iFrame will be loaded from the target web server. As this server is on the local network, the page loaded in the iFrame is allowed to make requests to the local network.

The page in the iFrame is also under the same origin as the top page, which allows the top page to fully control the DOM of the framed page. This includes injecting scripts which make fetch requests into the framed page. These scripts can be used to access the target web server and exfiltrate data just as they would be able to from the top page if PNA wasn't implemented at all.



So, putting this all together we end up with a complete plan:

- Load *http://chrome.r.intrud.es* which will trigger A and AAAA lookups for *chrome.r.intrud.es*.
- Have the DNS server return an A record pointing to the target web server on the local network, and a AAAA record pointing to an attacker-controlled web server on the public internet.
- Chrome will make the first request to load the top page from the attacker's web-server, which returns a page to execute the following steps.
- Shut down the attacker-controlled server so that all connection attempts are reset. Chrome will now make all requests to the target server on the local network.
- Load *http://chrome.r.intrud.es* in an iFrame.
- From the top page, inject a script into the framed page to request *http://chrome.r.intrud.es/secret.txt* and send the response to the attacker's web server.

This works to achieve split-second DNS rebinding in Chrome:

To help achieve this exploit, I wrote a small web server which will stop when it receives a request to */block*. You can find the source code and instructions for running it here.

When attacking automated browsers, you'll often want to include an iFrame in the page which takes some time to load. This stops the browser considering the page fully loaded until that iFrame has loaded, and ensures that the exploit script has enough time to run. The following demo shows

this exploit being used to extract credentials from the AWS metadata service when gowitness is used to take a screenshot of a malicious website from an EC2 with IPv6 enabled:

Or, for a scenario more likely to be found in a web application, headless Chromium being used to convert a web page to a PDF:

This bypass of Chrome's PNA restrictions was reported to the Chrome team through their issue tracker. They determined that it wasn't a security issue since the PNA restrictions are still in the process of being implemented.

Summary

DNS rebinding can be a useful weapon in your arsenal for attacking web applications. In the first post in this series, I tried to show how rebinding exploits against web applications can be achievable without much complexity. In this post, I've provided tools and techniques to build reliable exploits against web applications driving automated browsers, even if they only load pages for a short time. I hope you now feel well-equipped to start using DNS rebinding in exploits against the applications you test.
