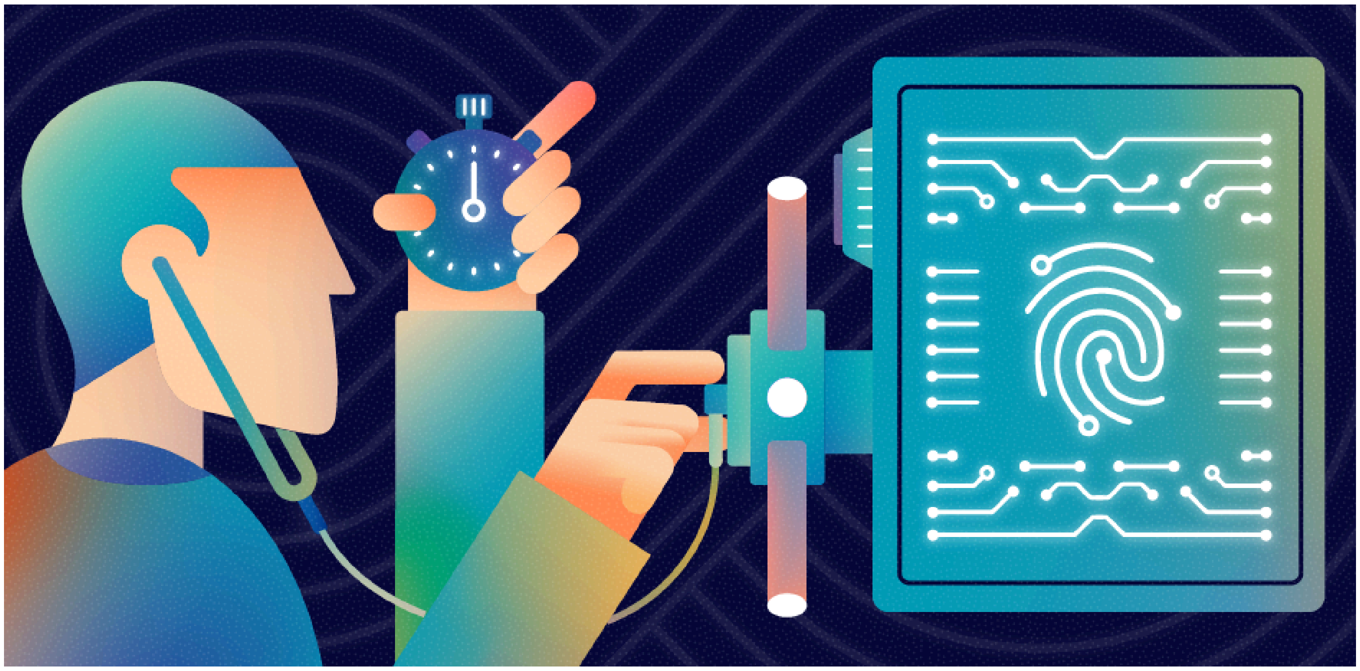
Listen to the whispers: web timing attacks that actually work

James Kettle

Director of Research

@albinowax

Published: 07 August 2024 at 18:10 UTC **Updated:** 18 November 2024 at 08:32 UTC



Websites are riddled with timing oracles eager to divulge their innermost secrets. It's time we started listening to them.

In this paper, I'll unleash novel attack concepts to coax out server secrets including masked misconfigurations, blind data-structure injection, hidden routes to forbidden areas, and a vast expanse of invisible attack-surface.

This is not a theoretical threat; every technique will be illustrated with multiple real-world case studies on diverse targets. Unprecedented advances have made these attacks both accurate and efficient; in the space of ten seconds you can now reliably detect a sub-millisecond differential with no prior configuration or 'lab conditions' required. In other words, I'm going to share timing attacks you can actually use.

To help, I'll equip you with a suite of battle-tested open-source tools enabling both hands-free automated exploitation, and custom attack scripting. I'll also share a little CTF to help you hone your new skillset.

Want to take things further? I'll help you transform your own attack ideas from theory to reality, by sharing a methodology refined through testing countless concepts on thousands of websites. We've neglected this omnipresent and incredibly powerful side-channel for too long.

This research paper accompanies a presentation at Black Hat USA and DEF CON:

DEF CON 32 - Listen to the Whispers: Web Timing Attacks that Actually Work - James Kettle

You can also read this whitepaper in a print-friendly PDF format.

Outline

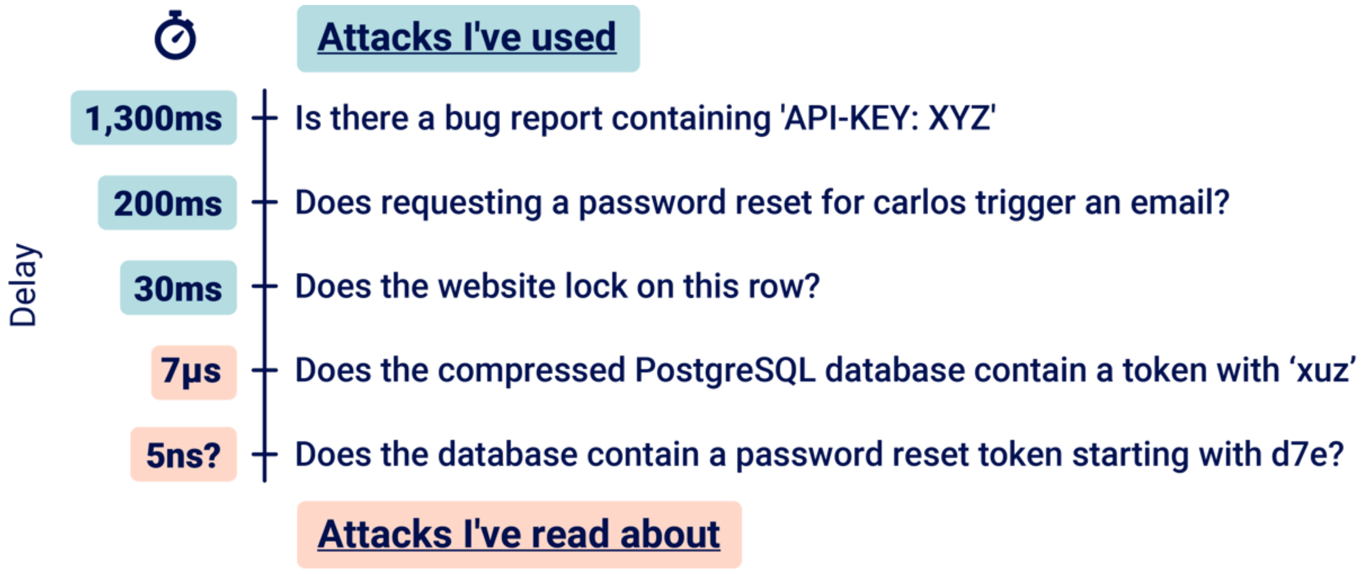
- Takeaways

Background

Web timing attacks are notorious for two things; making big promises, and failing to deliver. Examples are often theoretical, and even where a technique is dubbed 'practical' everyone knows it'll stop working as soon as you try to apply it outside a lab environment.

This reputation might be why we've ignored a huge opportunity.

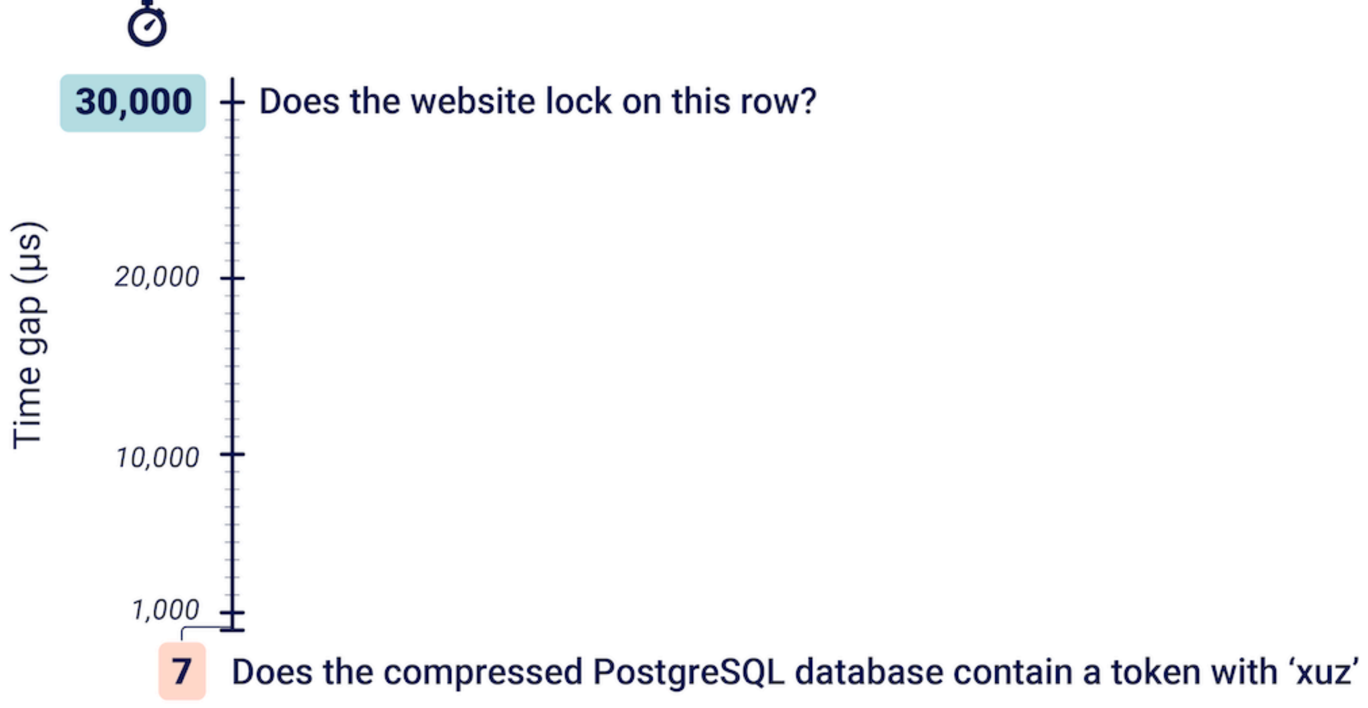
My first foray into researching timing attacks yielded results firmly in the 'theoretical' bucket. For my second attempt, I started by looking back over attacks that I'd successfully applied in the wild, alongside others that I'd read about:


Attacks I've used
Delay
1,300ms
Is there a bug report containing 'API-KEY: XYZ'
200ms
Does requesting a password reset for carlos trigger an email?
30ms
Does the website lock on this row?
7μs
Does the compressed PostgreSQL database contain a token with 'xuz'
5ns?
Does the database contain a password reset token starting with d7e?
Attacks I've read about

From the top, these are examples of:

- Cross-site search
- Username enumeration
- A probe for potential race conditions
- A lab-proven attack
- A theoretical attack that would be absolutely amazing... if it actually worked

In the hunt for novel techniques that work in the wild, I focused on the divide between the two categories, which is massive:



Timing attack research is often focused on a single target, but this constrains its real-world value. I wanted techniques that could be applied to arbitrary live targets. To ensure my new attack concepts met this standard, I validated them on a test bed of 30,000 live websites. Based on bbscope and Rapid7's Project Sonar DNS database, the test platform was a 20 GB Burp Suite project file containing every known website with a bug bounty program.

Before this research, the smallest time gap I'd personally exploited was 30,000μs. Now, it's 200μs. This was made possible by massive advancements in timing-attack accuracy, and enables multiple powerful new techniques.

Three key attack techniques stood out as providing valuable findings on a diverse range of live systems: discovering hidden attack surface, server-side injection vulnerabilities, and misconfigured reverse proxies. In this paper, I'll explore each of these in depth.

Making timing attacks that work everywhere

All three techniques are now available in Param Miner so, if you wanted to, you could stop reading and try them out right now. The true value of this research comes from understanding that it doesn't stop here; these are just a sample of what's possible. Timing attacks can take you almost anywhere, but to grasp this potential, we need to start from the beginning.

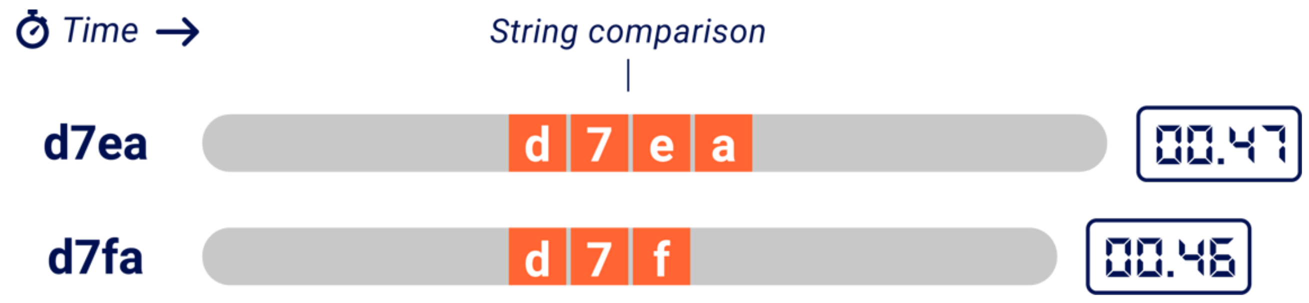
Let's have a closer look at the key factors that real-world timing attacks live or die by, and how to overcome them. In this section, I'll show how to make timing attacks 'local', portable, and feasible.

Answering difficult questions

It's easy to assume that all web timing attacks are exploits, but this is a mistake because it limits your thinking around potential applications. At their core, web timing attacks are simply about answering difficult questions - ones that can't be answered by observing the server's response.

I started this research by attempting a timing-based exploit on password resets. It went badly, but nicely illustrates the gap between theory and reality. Many websites implement password resets by storing a secret token in their database and sending the token in a link to the user's registered email address. When the user clicks the link, the website compares the user-supplied token with the one in the database.

Under the hood, string comparisons typically compare one character at a time until they either finish the string or encounter a non-matching character pair. This means that the more characters match, the longer the comparison takes:



In this illustration, we're using two HTTP requests to ask the question 'Does the database contain a password reset token starting with d7e?' The server is taking one second to compare each character, so by comparing the response times an attacker can tell that the token starts with 'd7e' rather than 'd7f.

Unfortunately, the actual time to compare each character is somewhere in the realm of 5 nanoseconds, or 0.000000005 seconds. Good luck exploiting that.

Noise vs signal

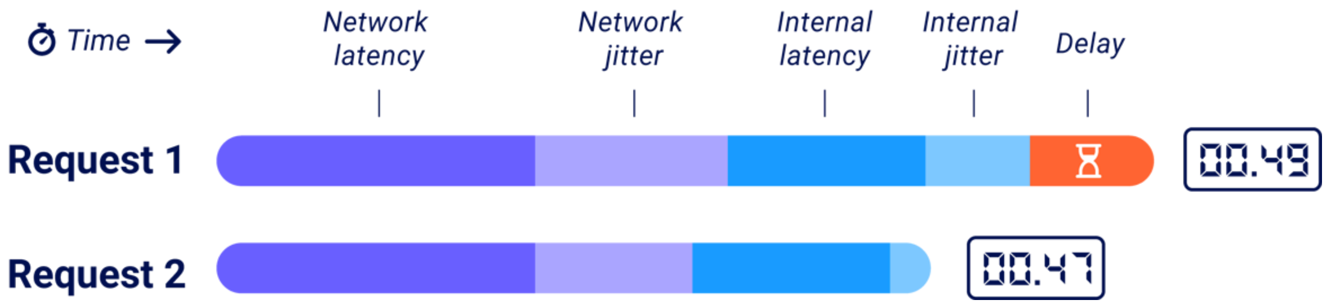
The success of every timing attack comes down to two competing variables - signal and noise. Signal refers to the size of the timing difference you want to detect, and noise refers to everything else that affects the response timing. If the signal is too quiet relative to the background noise, you won't hear it:

$$\text{success} = \frac{\text{signal}}{\text{noise}}$$

For an attack that actually works, you need to maximize the signal and minimize the noise. The rest of this section is focused on how to do this.

Note that this equation does not include 'number of measurements'. You can attempt to cancel out noise by taking repeated measurements, but this approach scales poorly. Once noise heavily outweighs signal you'll quickly need billions of measurements, resulting in an attack that takes so long the target will probably be decommissioned before it's complete.

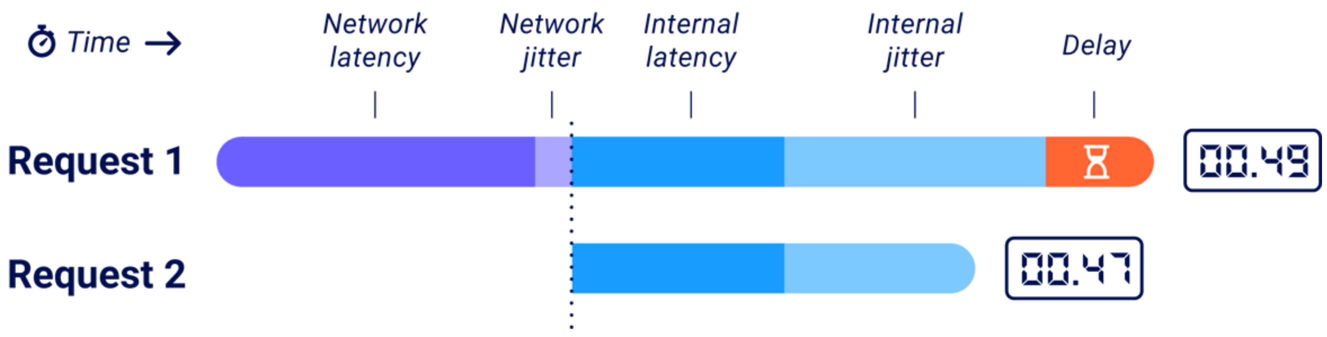
You can split noise into two parts - network noise (jitter), and server noise (internal jitter):



Network jitter is the *variation* in latency - the time taken for a packet to get to a target system and back again. It's the classic nemesis of remote timing attacks. When someone sees a timing attack demonstrated against a local system and says 'That'll never work on a remote system', they're basically saying that network jitter is going to make the attack impossible. Five years ago, this might have been true.

Making timing attacks 'local'

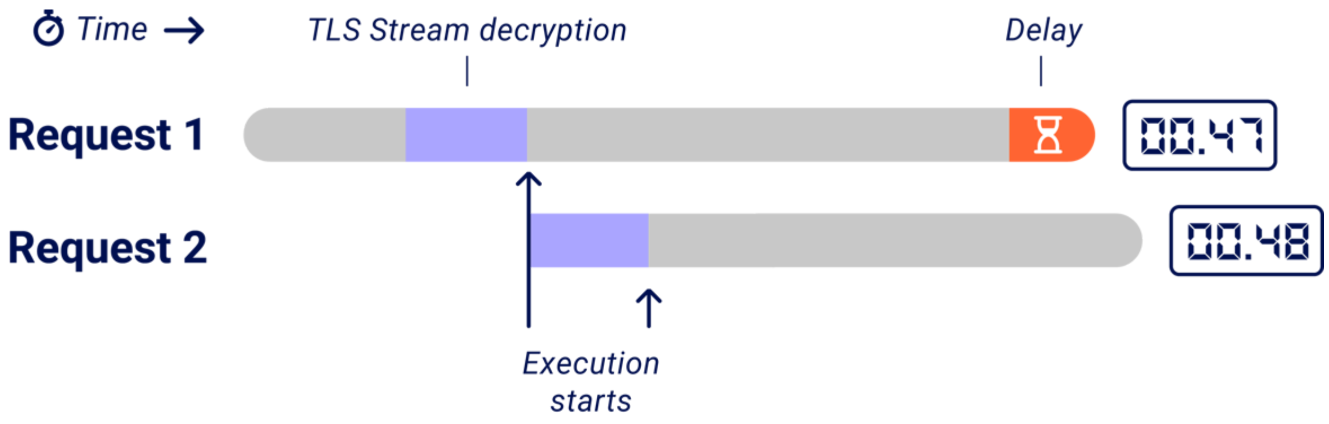
In 2020, Timeless Timing Attacks showed that you could fully eliminate network jitter from measurements using HTTP/2. You could place two HTTP/2 requests into a single TCP packet, ensuring they arrive at the server simultaneously. Then you could look at the order the responses arrive in, inferring which took longer to process on the server:



This single discovery eliminated the biggest source of noise and shifted the boundaries of what's detectable. There's just one small catch.

The sticky request-order problem

At the HTTP/2 layer, the two requests are completely concurrent, but the underlying TLS data is a stream so one request is still 'first' i.e. one will be fully decrypted before the other. If you try this technique out, you'll notice that websites show a significant bias towards answering the first request first. This bias probably stems from multiple factors, including the time it takes to decrypt the second request and resource availability. Unfortunately, this can mask the delay that you're trying to detect:



The authors noticed this problem and tackled it by adding dummy parameters to slow down parsing of the first request, in an attempt to resynchronise execution.

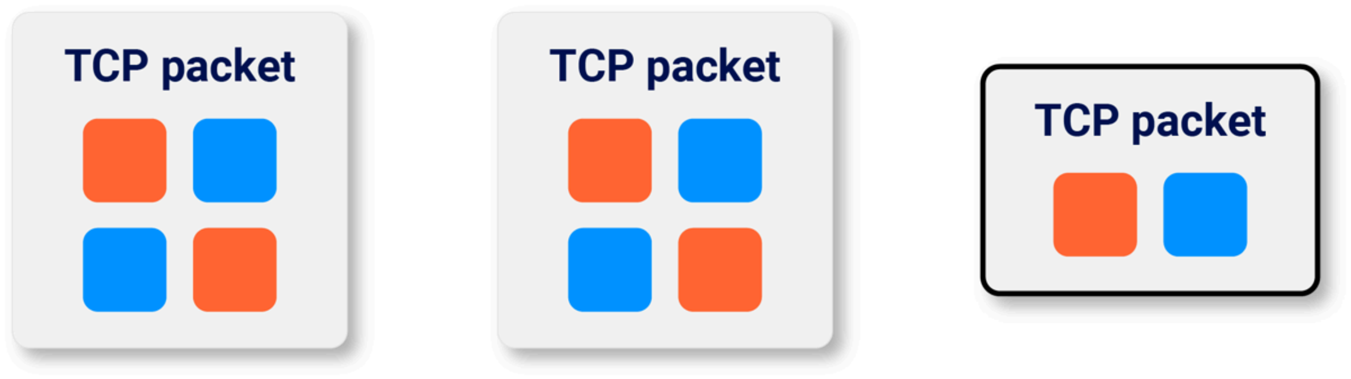
Making timing attacks portable

Lab environments are known for having less noise than real targets, but there's also a second, subtler issue. Focusing on a single target often yields target-specific techniques that require extensive tuning to apply anywhere else This makes them significantly less valuable for anyone working to a deadline.

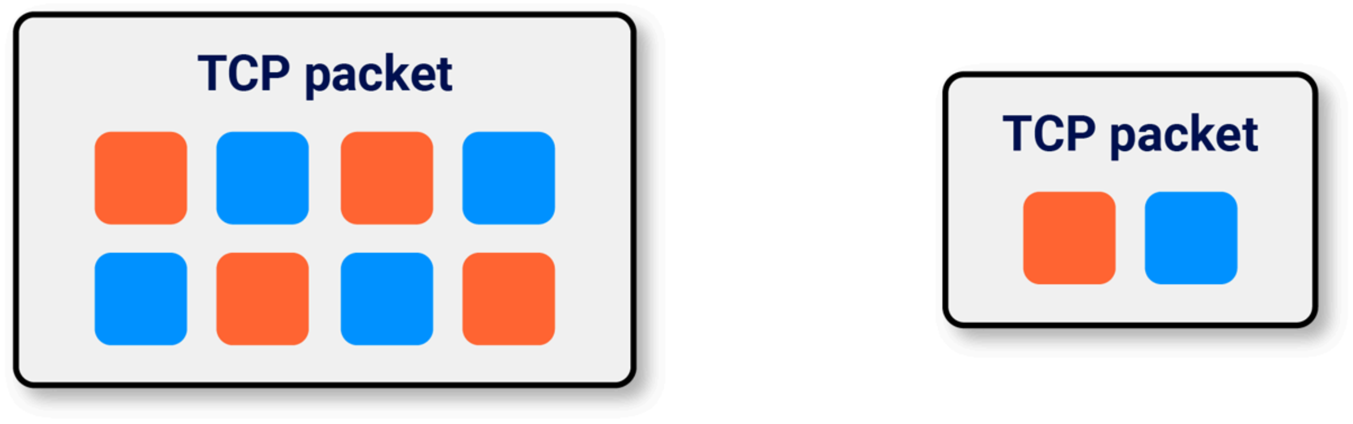
Unfortunately, dummy parameter padding is an example of this problem - its effectiveness depends on how the target implements parameter parsing, and how much processing capacity the system has available at that moment. Since spare processing capacity is affected by other systems, parameter padding can actually end up increasing the level of noise. I've observed different numbers of parameters being required on a single lab system, ten minutes apart.

What we really need is a way of tackling the sticky request-order problem that doesn't require per-target configuration. The single-packet attack, which I developed last year for race-condition discovery, provides a good starting point for this. The single-packet attack fragments the request in order to reduce the size of the 'critical packet' - the packet that completes the request and initiates execution.

It works by sending the bulk of the requests in an initial few packets, then completing the requests and triggering execution with a tiny final packet. In this diagram, the final critical packet is outlined in black:



Unfortunately, this introduces a different catch - some servers start to process HTTP requests as soon as they've got the headers, without waiting for the body. To fix that, we need to persuade our OS network stack to coalesce the header frames into a single packet so that regardless of which stage the server starts processing at, both requests get processed at the same time:



You might be wondering why I opted to split the requests into just two critical packets, instead of one packet per HTTP header. That would indeed be ideal, but unfortunately the HTTP/2 RFC forbids interleaving header frames from separate requests so it's unlikely to work.

Implementing this dual-packet sync turned out to be extremely easy - just add an extra ping frame! This harmless sacrificial packet ensures that the operating system coalesces the subsequent header frames.

```
disable TCP_NODELAY
send a ping frame  for each request with no body:
send the headers
withhold an empty data frame
for each request with a body:
send the headers, and the body except the final byte
withhold a data frame containing the final byte
wait for 100ms
send a ping frame
send the final frames
```

We integrated this improved technique into Burp Suite's built-in single-packet attack as soon as we discovered it, so you might have already benefited from it! I'm currently working with the developer of the open-source implementation h2spacex to get it in there too.

Making timing attacks feasible

With network noise out of the picture, our next target is server noise. Do not underestimate server noise. It stems from numerous sources including load on the target server, other systems it interacts with, other virtual systems running on the same physical hardware, and probably the weather near the datacenter. Server noise is the reason I haven't made any claims about what time-delay you can expect to detect with the enhanced single-packet attack - any such claim is so target-specific it's effectively meaningless.

To minimize server noise, take the shortest code path possible, and take full advantage of performance features like caching, object reuse, and connection reuse. Committed attackers may also reduce noise from other users using DoS techniques like CPDoS and resource consumption.

To maximize signal, focus on the slow code path and make it even slower by using random inputs to avoid server-side caching, incurring access to slow resources where possible, and multiplying the workload. For example, this request uses multiple headers with a fixed prefix to try to expand the delay caused by a server looking for a header starting with 'X-U':

```
GET / HTTP/1.1
X-Uaa: a
X-Ubb: a
X-Ucc: a
{256}
```

Modern web technologies like ORMs and GraphQL also are particularly suited for delay-expansion techniques. Remember that a DoS attack is just a really easy timing attack and adapt classic techniques like ReDoS, batching, and recursive XML entities.

Hidden attack surface

Vulnerabilities often lurk out of sight in disused and forgotten features that get overlooked by developers and security testers alike. As such, vulnerability discovery journeys often start with the detection of a hidden parameter, cookie, or HTTP header.

At its core, discovering these hidden inputs involves guessing potential parameter names and observing if they change the response. Parameters that don't alter the response may remain undetected, alongside any associated vulnerabilities. For my first bulk timing-attack, I decided to fix this.

Conveniently, I'm the core developer of Param Miner - possibly the first tool for bulk parameter discovery. Param Miner compares responses using attributes like 'word count', 'status' and 'line count'. For this research, I simply added 'response time' as an extra attribute, bumped up the repeat count, and got scanning.

I could have made Param Miner use the single-packet attack for these measurements, but this would have involved significant refactoring and, when researching unproven concepts, I take every possible shortcut to avoid wasting time, so I didn't bother.

Instead I just measured the time from the last byte of the request to the first byte of the response, and compared the bottom quartile of the two sets of 30 timing measurements to see if they were distinct (indicating a valid parameter), or overlapped. The bottom quartile is ideal for this comparison because it reflects the measurements with the least noise.

Discovery overload

Running the time-augmented Param Miner on the test bed of 30,000 live sites yielded a huge number of hidden parameters, including some really weird ones.

One highlight was a webserver that took 5ms longer to respond to requests containing the mystery HTTP header "commonconfig", unless the header value was valid JSON:

Header	Response	Time

`foo: x`	`HTTP/1.1 200 OK`	`50ms`
`commonconfig: x`	`HTTP/1.1 200 OK`	`55ms`
`commonconfig: {}`	`HTTP/1.1 200 OK`	`50ms`

Another discovery was on a webserver that refused to respond to any requests - it always reset the connection. This extremely defensive behavior wasn't sufficient to stop my scan discovering that it supported a certain HTTP header, because the header made it take significantly longer to reset the connection! Intriguing, but not terribly useful.

Header	**Response**	**Time**
`foo: x`	`--connection reset--`	`30ms`
`authorization: x`	`--connection reset--`	`50ms`

One frequent finding was much more practical:

Request	**Response**	**Time**
`GET /?id=random`	`HTTP/1.1 200 OK`	`310ms`
`GET /?foo=random`	`HTTP/1.1 200 OK`	`22ms`

This pair of responses tells us two valuable things. First, the site is only including specific parameters like 'id' in the cache key, so it's highly exposed to parameter-based cache poisoning attacks. Second, we know the 'id' parameter is keyed and this configuration is typically done site-wide. This means that using time analysis, Param Miner has detected a parameter that applies to a different page!

The hardest problem

When I tried this concept out, I anticipated two problems. First, I expected many of the techniques to fail completely. Second, I suspected that any valid results I encountered would be hidden in a morass of false positives.

The biggest challenge came from neither. It's that timing attacks are too powerful. They can detect so much that it's incredibly easy to misunderstand what you've detected. They're incredibly good at detecting 'something', but that something isn't necessarily what you're trying to detect.

This video shows what initially looks like a potential remote code execution vulnerability due to an 'exec' parameter causing a visible response delay. This delay turns out to be an indicator of a WAF doing additional processing on more suspicious requests. We then see that the delay stacks when the parameter is repeated, unless the request body is over a certain size threshold. Ultimately this leads to the discovery of a complete WAF bypass. This bypass discovery was completely unexpected to me, but it's since been found by others and is now implemented in the nowafpls tool. It remains a beautiful demonstration of how timing analysis can reveal insights into the target's control flow.

That was one of the easy cases - sometimes you may never fully understand what you've detected. Carry your assumptions lightly and test them from different angles wherever possible.

Proving the concept

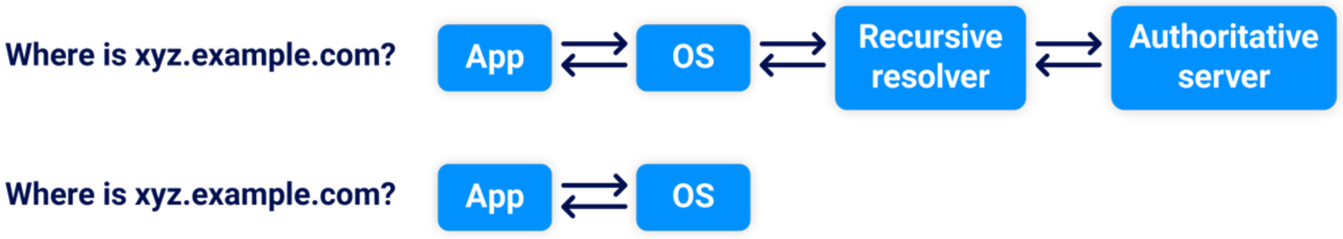
To avoid being misled by false assumptions, I decided to focus on specific parameters that provide a clear security impact without any time-consuming manual investigation and a straightforward way to gather additional corroborating evidence.

IP address spoofing via HTTP headers fulfilled these requirements perfectly. It's a relatively common misconfiguration and directly enables various exploits including rate-limit bypasses, forged logs, and even access control bypasses in some cases. By placing an IP address in a spoofed front-end header, you're effectively impersonating the front-end. We'll explore front-end impersonation attacks in more depth later.

Conveniently, if you place a domain inside a spoofed header, vulnerable servers will often perform an in-band DNS lookup to resolve it, causing an easily detectable delay. Here's a typical detection:

Header	Time
`Random-header: xyz.example.com`	`65ms`
`True-Client-IP: xyz.example.com`	`70ms`
`True-Client-IP: xyz.example.com`	`65ms`

The first response comes back quickly because it doesn't trigger a DNS lookup. The second response triggers a DNS lookup for xyz.example.com, so it's slower, and the third response arrives faster because the DNS response has been cached:



We'll revisit DNS caching later. In total, scanning for IP address spoofing revealed:

- 375 vulnerable domains
- 206 of these also caused a DNS pingback
- 217 visibly cached the result

This might leave you wondering about the ~170 vulnerable domains that didn't cause a DNS pingback - were they false positives? Here's one example:

Header	Time
`Random-header: x.psres.net`	`170ms`
`True-Client-IP: x.psres.net`	`90ms`
`True-Client-IP: 1.1.1.1`	`170ms`

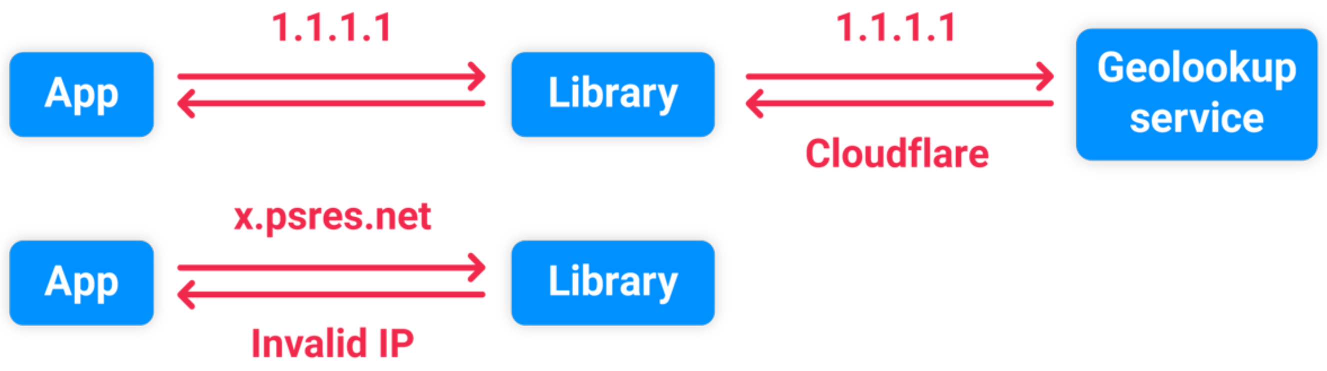
What do you think is happening here?

Here's a clue - in your login history, the website specified the login IP address and location:

Time	Browser	IP	Location

```
5 minutes ago     Chrome in Windows     1.1.1.1     Cloudflare
```

I think this system was passing the spoofed IP address into a library, which validated the format before passing it to a third-party Geolookup service. Supplying an invalid IP address like 'x.psres.net' caused an exception and stopped the slow IP-lookup from happening:



So, we've gained a new technique for parameter discovery, proved timing attacks can work at scale in the wild, and also spotted something significant: inputs that trigger errors can short-cut large code paths and result in significantly faster responses. In other words, timing attacks are exceptionally good at detecting exceptions

Server-side injection

Triggering and spotting exceptions is a foundational part of testing for server-side injection vulnerabilities, from SQLi to OS command injection. This makes timing analysis a perfect match for server-side injection detection.

I attempted to replicate my success with Param Miner by adding 'time' as a response attribute to Backslash Powered Scanner, but this fell flat. Without the single-packet attack, I could only detect major time differences and these predominantly came from WAFs rather than real vulnerabilities. Also, the tool's complexity made it hard to adapt it to overcome challenges.

For my second attempt, I reused some code from Param Miner to build a much simpler test that used the single-packet attack. I issued up to 50 request pairs per probe, and recorded the response order of each pair. If the response order was at least 80% biased towards one payload, I reported it as a valid finding.

Fully blind SQLi

The first finding was a fully blind SQL injection, detected with a classic payload pair:

Request	Response	Time
`GET /api/alert?mic='`	`{}`	`162ms`
`GET /api/alert?mic=''`	`{}`	`170ms`

Unfortunately, when I reported this it turned out to be a duplicate. In retrospect, I should have seen this coming - you could easily detect the same vulnerability using the well-known `'||sleep(5)||'` payload. Advanced timing analysis simply isn't required to detect vulnerabilities where you can inject sleep statements. Likewise, timing isn't great for finding code injection because you can normally find those better by using OAST techniques.

For powerful vulnerabilities like command injection, SQLi, and code injection, timing-based detection is only really useful when you've got a WAF or filtering in place that blocks the classic detection techniques. Let's look elsewhere.

Blind JSON injection

Timing comes into its own when looking for the injection underclass; vulnerabilities that allow manipulation of data structures and formats, but stop shy of full code execution. This includes injection into formats like JSON, XML, CSV, and server-side query parameters and HTTP headers. Many of these bugs are rarely spoken of because they're so hard to detect.

They're hard to exploit too, but sometimes you can combine timing information with visible features to gain extra insight into what's happening behind the scenes. For example, I spotted one target where an invalid JSON escape sequence made the response come back 200us (0.2ms) faster:

Parameter	Response	Time
`key=a\"bb`	`"error": {` `"message": "Invalid Key: a\"bb"` `}`	`24.3ms`
`key=a"\bb`	`"error": {` `"message": "Invalid Key: a"\bb"` `}`	`24.1ms`

What do you think is happening server-side?

There's a clue in the response formatting - the invalid syntax we injected hasn't altered the formatting in the response. I would expect a JSON formatter to fail when run on invalid syntax, or at least return visibly different output.

Also, lengthy inputs got redacted in the response:

Parameter	Response	Time
`key=aaa…a"bbb`	`"error": {` `"message": "Invalid Key: ****bbb"` `}`	`24.3ms`

This feature provides a second clue: when our invalid JSON sequence got redacted, the timing difference disappeared! Taken together, this strongly suggests that the delay is happening due to a component parsing the response being sent to us. My best guess is that it's some kind of error logging system. I was pretty pleased about figuring this out from a 0.2ms time differential but with no clear path to an exploit, I decided to move on.

Blind server-side parameter pollution

My most prolific probe was for blind server-side parameter pollution. This worked by comparing the response times for reserved URI characters like ? and #, with non-reserved characters like !.

In some cases, sending an encoded # made the response come back faster:

Request	Response	Time
`/path?objectId=57%23`	`Can't parse parameter`	`180ms`
`/path?objectId=57%21`	`Can't parse parameter`	`430ms`

This could be due to the fragment breaking a server-side path and getting a speedy static response from the back-end, or the application's HTTP client simply refusing to send a HTTP request containing a raw #. Of course, it's crucial not to assume which way around the delay will land - on other targets, the encoded # made the response arrive slower.

Server-side parameter pollution was the most common type of injection discovery by a huge margin, so I think it's a promising area for further research. For more information on this attack class, check out server-side parameter pollution, and Attacking Secondary Contexts in Web Applications.

Bug doppelgangers

As we've seen, high-precision timing is great for detecting blind injection bugs but they aren't always easy to exploit. While analyzing these findings I often gained some understanding of what was happening server-side, but stalled short of actual exploitation. Also, timing tends to surface lesser-known attack classes that we're less familiar with exploiting.

Gathering enough information for an exploit based purely on timing evidence is often tricky and time-consuming. Testing each idea on a regular, non-blind vulnerability typically involves a single repeater request, whereas with many of these, you're potentially looking at a 30-second Turbo Intruder attack.

One thing that can help here is 'bug doppelgangers' - non-blind variations of the target bug class. Param Miner will report these, and they're great for learning how to interpret and exploit these bugs in a less challenging environment.

Bug doppelgangers form part of a broader, recurrent theme from this research. If you ignore timing, you'll miss out, but if you focus too much on timing, you'll also miss out. For success, use every available information channel.

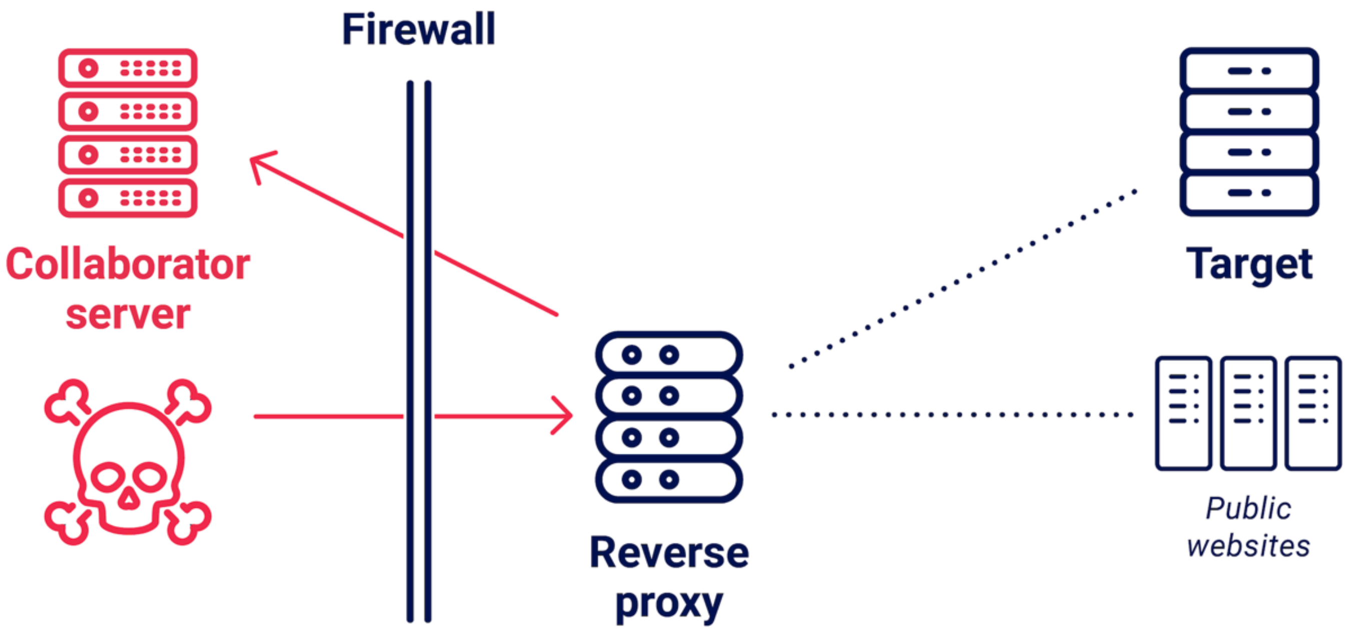
Reverse proxy misconfigurations

The single biggest breakthrough in this research was when I realized I could use timing to detect a widely overlooked type of SSRF.

Back in 2017, I researched techniques to exploit misconfigured reverse proxies for SSRF and gain access to internal systems. The most common vulnerability was servers which routed requests to the domain specified in the HTTP Host header. To detect these, I would send them a request with a Host pointing to a domain I controlled:

```
GET / HTTP/1.1
Host: uniq-token.burpcollaborator.net
```

If the target was vulnerable, I would see my request arriving on my site at burpcollaborator.net, forwarded by the vulnerable reverse proxy.



After that I would send internal IPs and hostnames to plunder their internal network. This yielded some spectacular findings, including accidentally hacking a system that my ISP put in place to MITM their customers.

Scoped SSRF

Although successful, this detection technique had a major blind spot - scoped SSRF.

After I published the research, someone from Google asked if I'd found any vulnerabilities in their systems, strongly implying that they had been vulnerable. Shortly later, Ezequiel Pereira posted $10k host header in which he exploited an open proxy belonging to Google that I'd failed to detect. My scanning method had failed because Google's proxy was configured to only route requests to their own systems, so my server never received a DNS lookup.

This was a hint at a really common scenario, where companies allow request forwarding to arbitrary subdomains:

Host header	Full SSRF	Scoped SSRF
`random.example.com`	`404 Not Found`	`404 Not Found`

random.notexample.com	404 Not Found	403 Forbidden

I don't think there's an established name for this type of SSRF, so I'll call it scoped SSRF. This restriction can be implemented via an internal DNS server, simple hostname validation, a firewall blocking outbound DNS, or a tight listener config. The outcome is always the same - you've got a bug with an impact close to full SSRF, but it can't be detected using pingback/OAST techniques.

Detecting scoped SSRF

To detect scoped SSRF, we need to answer the question "Did the server try to connect to the specified hostname?". Timing is perfectly suited for this. Consider a server at www.example.com that issues the following responses:

Host header	Response	Time
foo.example.com	404 Not Found	25ms
foo.bar.com	403 Forbidden	20ms

These two responses show that it's doing some kind of validation on the Host header, but there isn't sufficient information to tell if it's an open proxy. If you rely on the response content, you'll end up with both false positives and false negatives.

The following request pair is what proves the issue - the faster second response is evidence of DNS caching:

Host header	Response	Time
abc.example.com	404 Not Found	25ms
abc.example.com	404 Not Found	20ms

Some DNS systems don't cache failed DNS lookups, but I found an alternative solution for this - sending an overlong 64-octet DNS label, leading to the DNS client refusing to issue the lookup and a faster response:

Host header	Response	Time
aaa{62}.example.com	404 Not Found	25ms
aaa{63}.example.com	404 Not Found	20ms

Sifting for secret routes

Scanning with these techniques revealed hundreds of vulnerable reverse proxies, exposing alternative routes to tens of thousands of domains - I desperately needed automation.

When you find an open reverse proxy, the first step is to try using it to access every possible destination. I wrote code to automatically compile a list of target subdomains using three main sources:

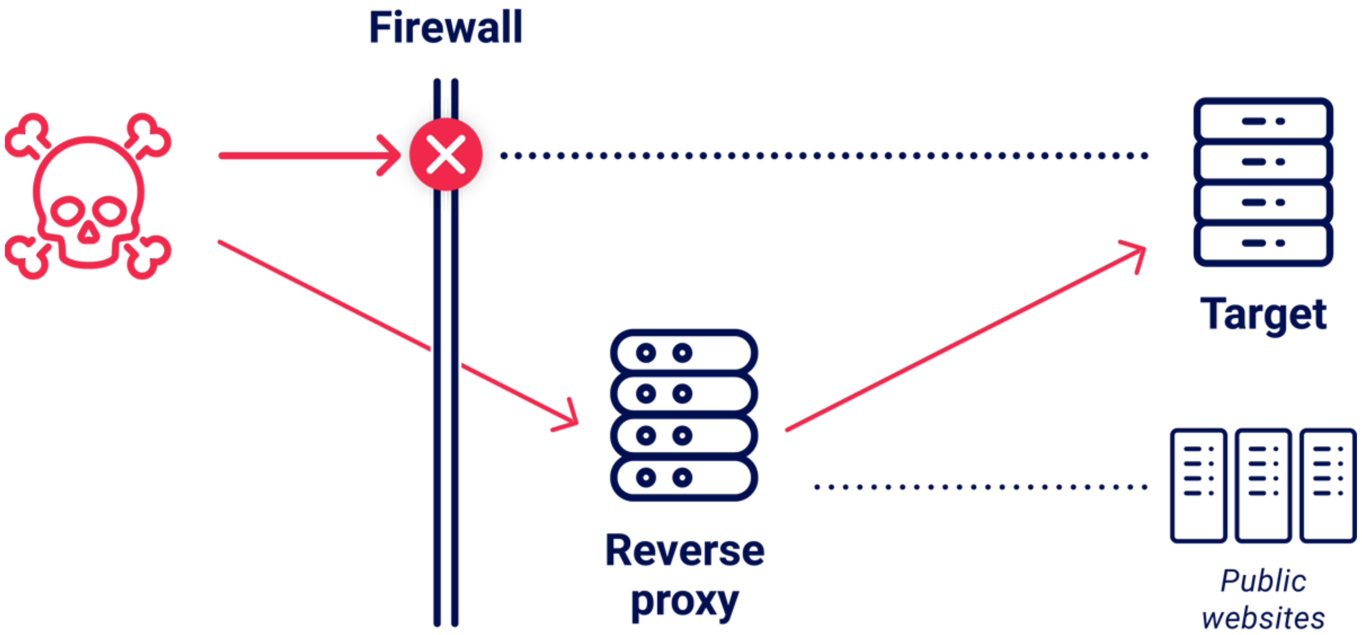
- A hard-coded, generic subdomain wordlist
- A list of known subdomains sourced from Rapid7's Project Sonar 'fdns' file. To quickly parse this 58 GB file for subdomains of a specific target, I used 'rev' to reverse every line, then sorted it alphabetically so sibling domains were next to each other. I then ran the old-school unix 'look' utility to do a binary search. This cut the search time by 99.999% versus grep.
- The online subdomain service at columbus.elmasy.com, which is mostly compiled from certificate transparency logs

I made Param Miner try to access each host twice - once directly and once through the proxy - and report any hosts where the two access attempts triggered significantly different responses. When comparing responses, I focused on response status code, header names, and the Location header as these were the highest-signal areas. This yielded numerous findings, which fell into four broad categories.

Guessing hostnames directly in the Host header is often referred to as 'vhost bruteforcing', but reverse-proxy exploitation often looks completely different, so it's important to understand the distinction. Virtual-host bruteforcing only provides access to other websites on the same server. Meanwhile, reverse proxies will route requests to different systems, enabling unique attacks like front-end rule bypass, front-end impersonation, and exploit chaining opportunities. Let's dive in.

Firewall bypass

The simplest exploit is where you can see the target from outside but can't directly access it.

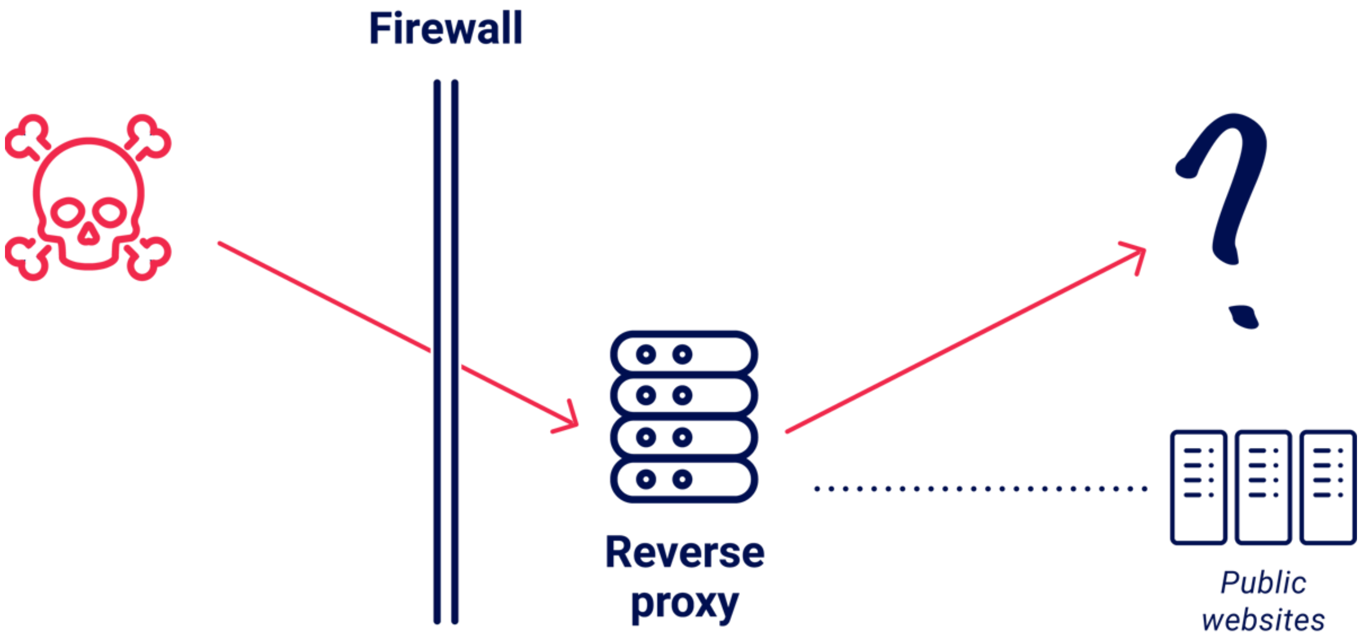


On one company, sonarqube.redacted.com resolved to a public IP address, but attempting to access it triggered a connection reset from a firewall. My probes had identified app.redacted.com as a reverse proxy and, using that, I was able to route around the firewall and access the internal SonarQube instance.

Entry point	Host header	Result
`sonarqube.redacted.com`	`sonarqube.redacted.com`	`--reset--`
`app.redacted.com`	`sonarqube.redacted.com`	`200 OK`

Firewall bypass - invisible route variant

There's a common variation where the internal system doesn't have a convenient public DNS record to let you know it exists:



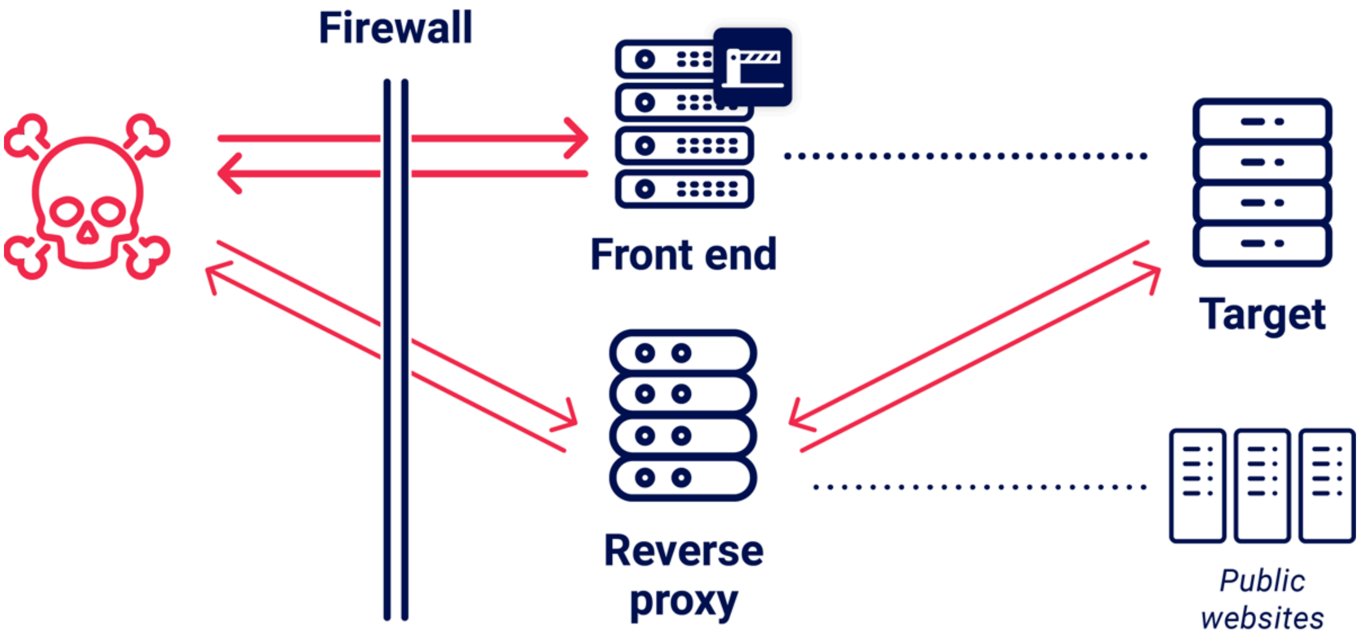
There are a huge number of pre-prod, staging, and development servers exposed to anyone applying this technique. If you get lucky, they'll have debugging enabled or test credentials configured, making them soft targets. These systems may even have real target data, or reused keys from production.

The most interesting targets I found were pre-launch systems still under active development. In particular, I discovered an admin console with apparently-public access on a really cool US government system, which I'm gutted I can't provide any details about. I reported the issue and the system went 'live' a few months later, but the admin console is nowhere in sight.

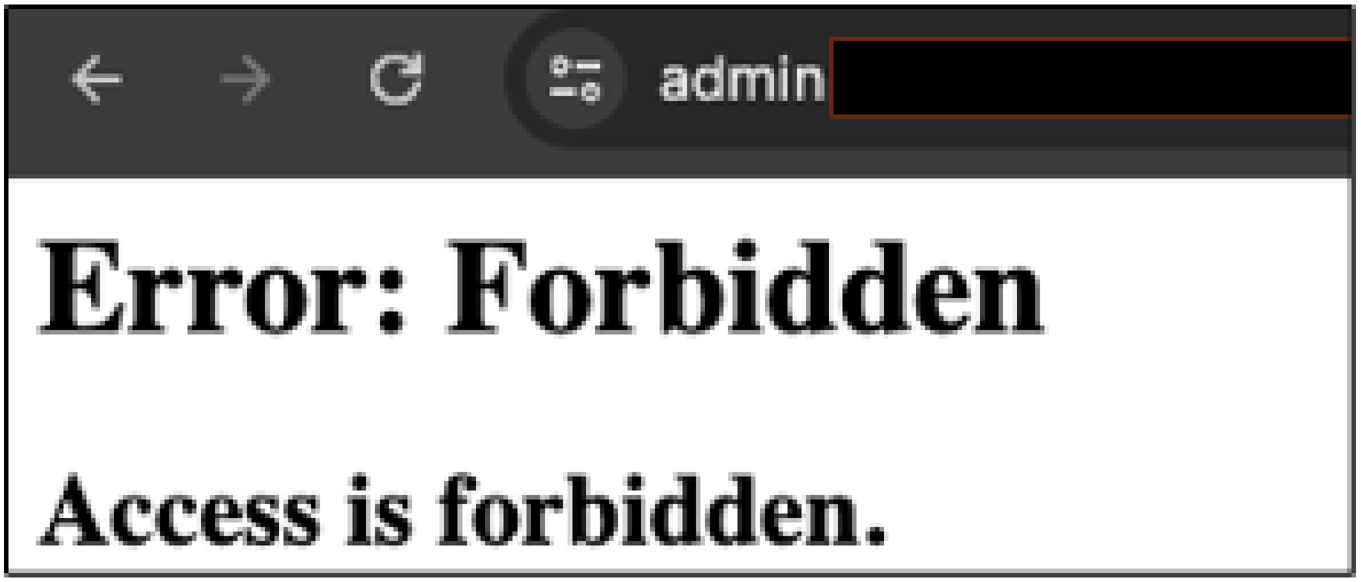
Front-end rule bypass

Some targets are publicly accessible, but sit behind front-end servers that enforce inconvenient security rules that block attacks or restrict access to valuable endpoints. The classic way to handle these is by talking directly to the back-end, but that's often impossible due to firewalls.

Reverse proxies provide a compelling alternative - go around the barrier:



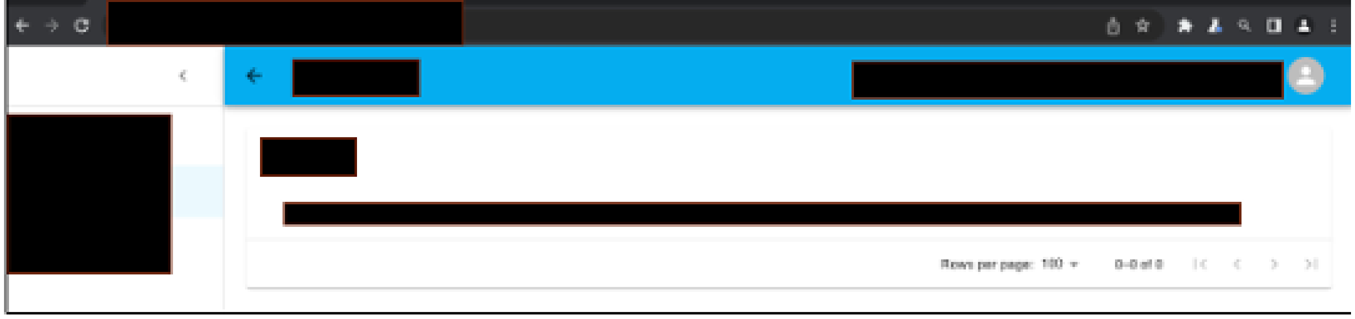
On one target, using an alternative route via a reverse proxy turned this:


admin [redacted]

Error: Forbidden

Access is forbidden.

Into this:


[redacted]
[redacted]
[redacted]
[redacted]
[redacted]
[redacted]

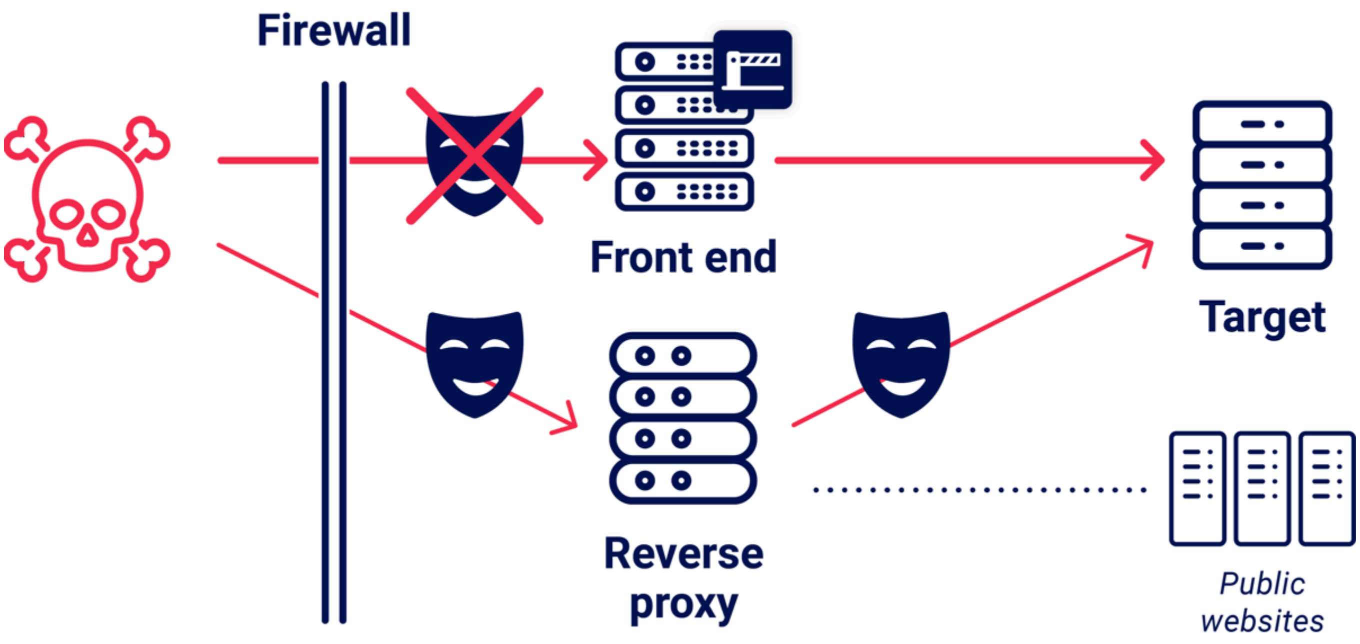
Rows per page: 100 ▾ 0–0 of 0

Front-end impersonation attacks

The most spectacular and surprising exploits happen when there's a trust relationship between the front-end and back-end. It's common knowledge that you can use headers like X-Forwarded-For to spoof your IP address. What's less appreciated is that this is part of a much broader and more powerful bug class. This type of attack has no established name, so I'll call it a front-end impersonation attack.

Front-end systems often add HTTP headers onto requests before forwarding them to the back-end. These contain additional information that the back-end might find useful, such as the user's remote IP address, and the originating protocol. More complex deployments sometimes use custom headers to transmit critical authentication information. Back-end servers trust these headers implicitly.

If an attacker attempts to spoof these headers, the front-end will typically overwrite them. This header overwriting behavior is the single brittle line of defense against front-end impersonation attacks.



The easiest way to bypass this defense is to simply talk directly with the back-end, but this is usually impossible due to network firewalls. Another approach is HTTP request tunneling, which I used to completely compromise New Relic's core internal API using a header called "Service-Gateway-Is-Newrelic-Admin". You can also try obfuscating headers to smuggle them past the front-end.

Misconfigured proxies offer an elegant alternative way to bypass header-overwriting defenses and perform front-end impersonation attacks. To try this out,

- Use Param Miner's 'Detect scoped SSRF' scan to detect a reverse proxy
- Run 'Exploit scoped SSRF' to find alternative routes to internal systems
- On each alternate route, run 'Guess headers' to find useful headers

Applying this successfully requires a robust mental visualization of what's happening behind the scenes. To help out, I've made a little CTF at listentothewhispers.net - see if you can crack it!

Chaining

Finally, scoped SSRF via reverse proxies offers some great exploit chaining opportunities.

If you're able to take over a subdomain on the target company and point the DNS record to an arbitrary IP address, you can use this to upgrade a scoped SSRF into a full SSRF and hit arbitrary IP addresses. This is a lot like chaining a traditional SSRF with an open redirect.

Since reverse proxies let you pick your back-end, they're great for HTTP request smuggling. I didn't have time to properly explore this concept. In short, I think you'll find that, while it should be easy to find back-ends that are vulnerable to request smuggling, cross-user exploitation will often be impossible because no legitimate users will be sharing your front-end/back-end connection. To prove the impact, you'll need to pursue tunneling-based exploits like front-end impersonation and header disclosure.

What's next?

My plan

My goal for this research is to get people using timing attacks day to day. As such, I plan to spend the next month improving the tooling in Param Miner and Turbo Intruder. In particular, I think it's possible to make most timing attacks quite a bit faster simply by using the t-test to decide whether to report a discovery, bail, or get more samples. I'll also be looking out for user feedback - if you have any requests or thoughts, let me know via Github or send me an email.

Timing research roadmap

These findings have just scratched the surface, and timing attacks still have massive potential for further research. If you're interested to see where this attack class might go next, or pushing it further yourself, there are many different avenues to consider.

I think the single most valuable area is looking for new applications of timing attacks. This is relatively easy, and doesn't require a major time commitment just to get started. The main hazard here is accidentally pursuing a concept where the signal you need to detect is drowned out by noise. Fortunately, this is easy to avoid. Start by thinking about the cause of the delay. Does it come from an extra interaction with a remote system, LAN system, hard disk, RAM, or CPU register? Once you're working at the right level, consider building a local benchmark to measure the signal size that you'll need to detect.

If the signal is too small, explore amplification techniques. Remember that most DoS attacks are really just timing attacks, and embrace them. Maybe you can expand the delay using nested XML entities, ReDoS, or hashtable collisions.

Jitter-reduction techniques are incredibly valuable and widely overlooked too - there may be some great techniques waiting for someone to research this area.

There's also scope for universal, technique-level improvements. Maybe the single-packet attack works better if you fragment at the TCP layer. Perhaps it's more effective to send ten requests in a single packet instead of two?

Finally, whichever path you take, try to resist the lure of hyper-focus on a single target - generic and reusable techniques contribute far more to the development of the field.

Defence

Timing attacks are hard to defend against. First and foremost, developers should understand that attacker visibility into their system's inner workings goes beyond the actual response content.

It's safest to over-estimate attackers' capabilities. Assume an attacker can read every line of code that gets executed. This is similar to your code being open-source, but slightly more serious because live data will affect the execution flow. Attackers can't directly access variables, but they can see which branches get taken and how many iterations each loop goes through.

It's especially important to take this into account when implementing performance optimisations such as caching as these tend to provide a massive signal. To mitigate attacks that exploit smaller signals, you could try breaking the single-packet attack by implementing a rate limit restricting each IP address to one request per 1-5 ms.

Likewise if you're a WAF vendor, consider detecting when a single packet contains multiple HTTP requests and breaking them down into separate packets with a tiny delay between each.

Finally, yes I do recommend using constant-time functions when comparing user input with secret keys. Just ask anyone who says this is an actual threat to provide a proof of concept.

Takeaways

It's not just about the exploits. At their core, web timing attacks are about answering difficult questions.

With the single-packet attack, web timing attacks have become 'local', portable, and feasible.

Timing oracles are everywhere. Whatever you're testing, timing murmurs are always present, waiting for you to listen.

Enjoy!

Black Hat SSRF Timing attacks timing

← Back to all articles

Related Research

New crazy payloads in the URL Validation Bypass Cheat Sheet
29 October 2024

Introducing the URL validation bypass cheat sheet
03 September 2024

Gotta cache 'em all: bending the rules of web cache exploitation
08 August 2024