When TLS Hacks You

JOSHUA MADDUX
Demo
Overview

- Where I Started
- Testing Approach
- Implications
  - Concrete Vulnerabilities
- Defense
SSRF

- Send a URL, server hits it
SSRF

- Send a URL, server hits it
- Common in webhooks & Apple Pay support

https://www.youtube.com/watch?v=m4Bxl9PUx0
Webkit.org: Apple Pay SSRF

Webkit.org:
Apple Pay
SSRF

EC2 IMDS V1
Webkit.org: Apple Pay SSRF

EC2 IMDS V1

Website 2: no data back 😞
Webkit.org: Apple Pay
SSRF

Website 2: no data back 😞

Website 3: PUT request

EC2 IMDS V1

405 not allowed
Webkit.org: Apple Pay
SSRF

Website 2: no data back 😞

Website 3: PUT request 😞
405 not allowed

Website 3: validation 😐

EC2 IMDS V1
Getting around limitations
Past approaches

Weird protocols

- `gopher://localhost:11211/_%0aset%20foo%20...`
  - Doesn’t work against modern libraries

SNI injection

- `https://127.0.0.1 %0D%0AHELO orange.tw%0D%0AMAIL FROM...:25/`
  - From Orange Tsai’s talk “A new era of SSRF”
    - https://www.youtube.com/watch?v=2MslLrPinm0
  - Really cool, but depends on specific bugs
### SSL Handshake Details

- **ssl.handshake.type == 1**

<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>5215</td>
<td>291.739479443</td>
<td>192.168.1.13</td>
<td>192.30.255.112</td>
<td>TLSv1.3</td>
<td>58</td>
</tr>
<tr>
<td>5242</td>
<td>292.029938053</td>
<td>192.168.1.13</td>
<td>185.199.111.154</td>
<td>TLSv1.2</td>
<td>65</td>
</tr>
</tbody>
</table>

**Server Name List Length:** 26

**Server Name Type:** host_name (0)

**Server Name Length:** 23

**Server Name:** github.githubassets.com

```plaintext
00b0 00 35 00 0a 01 00 01 d4 00 00 00 1c 00 1a 00 00
00c0 17 67 69 74 68 75 62 2e 67 69 74 68 75 62 61 73
00d0 73 65 74 73 2e 63 6f 66 00 17 00 00 ff 01 00 01
00e0 00 00 0a 00 0e 00 0c 00 1d 00 17 00 18 00 19 01
00f0 00 01 01 00 0b 00 02 01 00 00 23 00 d0 c2 09 ea
0100 7b 3f 89 eb d7 12 d0 05 95 bd 12 02 70 0b b6 64
```
Step 1

jmaddux.com

Hello

+ Payload

jmaddux.com
Step 2

Saved Payload
<table>
<thead>
<tr>
<th>No.</th>
<th>Time</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Length</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>5215</td>
<td>291.739479443</td>
<td>192.168.1.13</td>
<td>192.30.255.112</td>
<td>TLSv1.3</td>
<td>583</td>
<td></td>
</tr>
<tr>
<td>5242</td>
<td>292.029938053</td>
<td>192.168.1.13</td>
<td>185.199.111.154</td>
<td>TLSv1.2</td>
<td>652</td>
<td></td>
</tr>
</tbody>
</table>

Random Bytes: 4f82a084a4e441e2c776f0fb53f11c66fb2725f7c705480a...
Session ID Length: 32

Session ID: b98ddc30103ef10d116f2b668705bd8b1a9842c42925fd55...
Cipher Suites Length: 36

| 0060 | 66 fb 27 25 f7 c7 05 48 0a fb a5 a4 51 20 b9 8d | f % H Q ... |
| 0070 | dc 30 10 3e f1 0d 11 6f 2b 66 87 05 bd 8b 1a 98 | .0 > .0 +f ... |
| 0080 | 42 c4 29 25 fd 55 d0 a8 96 23 52 be 73 ee 00 24 | B . % U . #R . s . $ |
| 0090 | 13 01 13 03 13 02 c0 2b c0 2f cc a9 cc a8 c0 2c | ........+ ./ ... |
| 00a0 | c0 30 c0 0a c0 09 c0 13 c0 14 00 33 00 39 00 2f | 0 ........ 3.9 / |
| 00b0 | 00 35 00 0a 01 00 01 d4 00 00 00 1c 00 1a 00 00 | 5 ............ |
| 00c0 | 17 67 69 74 68 75 62 2e 67 69 74 68 75 62 61 73 | github. githu... |
| 00d0 | 73 65 74 73 2e 63 6f 6d 00 17 00 00 ff 01 00 01 | sets.com ....... |
for(i = 0; i < data->set.general_ssl.max_ssl_sessions; i++) {
    check = &data->state.session[i];
    if(!check->sessionid)
        /* not session ID means blank entry */
        continue;
    if(strcasecmp(name, check->name) &&
        ((!conn->bits.conn_to_host && !check->conn_to_host) ||
          (conn->bits.conn_to_host && check->conn_to_host &&
            strcasecmp(conn->conn_to_host.name, check->conn_to_host)) &&
          ((!conn->bits.conn_to_port && check->conn_to_port == -1) ||
            (conn->bits.conn_to_port && check->conn_to_port != -1 &&
              conn->conn_to_port == check->conn_to_port)) &&
          (port == check->remote_port) &&
          strcasecmp(conn->handler->scheme, check->scheme) &&
          Curl_ssl_config_matches(ssl_config, &check->ssl_config)) {
/* information stored about one single SSL session */

struct curl_ssl_session {
    char *name;          /* host name for which this ID was used */
    char *conn_to_host;  /* host name for the connection (may be NULL) */
    const char *scheme;  /* protocol scheme used */
    void *sessionid;     /* as returned from the SSL layer */
    size_t idsize;       /* if known, otherwise 0 */
    long age;            /* just a number, the higher the more recent */
    int remote_port;     /* remote port */
    int conn_to_port;    /* remote port for the connection (may be -1) */
    struct ssl_primary_config ssl_config; /* setup for this session */
};
Step 3

jmaddux.com:25

35.x.x.x

jmaddux.com?

127.0.0.1

DNS server
SMTP on localhost

Hello + Payload

SMTP on localhost
• • • set z 0 0 14 •
• • • im in u r cache •
c CL;η@r5 ]R+S,6l86<N;
>,Q+$/$(k#'g
9
3=<5/ssltest.jmaddux.com

3t
0.http/1.11
+
-3&S .TkvC](kz+)
set :1:page_hits 1 300 56
-posixsystemopen -a CalculatorR.
FtD<10|
}s
}{kpery@.09X9j?* ?^C
Testing approach
Internet

Alternating DNS Server

Custom TLS

IP of Custom TLS box
127.0.0.1

redis config
Code available at: https://github.com/jmddx/TLS-poison

Fork of https://github.com/SySS-Research/dns-mitm

Fork of https://github.com/ctz/rustls

Thanks to Akash Idnani for writing the redis-based configuration stuff
Implications
What’s now vulnerable

Almost-SSRF

Outbound TLS sessions

Stuff on local ports
Almost-SSRF

- OIDC discovery (sometimes)
- Webpush
- Webmention
- Apple Pay Web
- In browsers, just phishing people (Then we call it CSRF)
  - Wifi captive portals
- SSDP

Surprisingly common

- SVG conversion
- URL-based XXE
- Scraping
- Webhooks
- PDF renderers with images enabled

Outbound TLS sessions

Stuff on local ports
Almost-SSRF

Getting more common

Outbound TLS sessions

Stuff on local ports
<table>
<thead>
<tr>
<th>HTTPS Client library/application</th>
<th>Can haxx you?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java HttpsURLConnection</td>
<td>Yes</td>
</tr>
<tr>
<td>Webkit</td>
<td>Yes</td>
</tr>
<tr>
<td>Chrome</td>
<td>Yes</td>
</tr>
<tr>
<td>Firefox</td>
<td>No</td>
</tr>
<tr>
<td>Curl/libcurl</td>
<td>Yes</td>
</tr>
<tr>
<td>IOS, Android SSDP</td>
<td>Yes</td>
</tr>
<tr>
<td>Python ‘requests’ package</td>
<td>No</td>
</tr>
<tr>
<td>Go http client</td>
<td>Not yet</td>
</tr>
<tr>
<td>node-fetch, axios</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caches by IP address, not domain (should be both)

Open issue on github to cache sessions

Node has built-in cache
Almost-SSRF

Outbound TLS sessions

Stuff on local ports

What stuff?
# Internal SSRF Targets

<table>
<thead>
<tr>
<th>Package</th>
<th>Susceptible?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached</td>
<td>Yes</td>
<td>Common Route to RCE!</td>
</tr>
<tr>
<td>Hazelcast</td>
<td>Yes</td>
<td>Common in Java apps</td>
</tr>
<tr>
<td>Redis</td>
<td>No</td>
<td>Closes connections after null bytes</td>
</tr>
<tr>
<td>SMTP</td>
<td>Yes</td>
<td>All implementations I’ve seen</td>
</tr>
<tr>
<td>FTP</td>
<td>Yes</td>
<td>All implementations I’ve seen</td>
</tr>
<tr>
<td>Mysql, Postgres, etc.</td>
<td>Maybe</td>
<td>Let me know if you make this happen</td>
</tr>
<tr>
<td>FastCGI</td>
<td>Maybe</td>
<td></td>
</tr>
<tr>
<td>Zabbix</td>
<td>No</td>
<td>Similar reasons as redis</td>
</tr>
<tr>
<td>Syslog</td>
<td>Yes</td>
<td>Less severe</td>
</tr>
</tbody>
</table>
Concrete Vulnerabilities
Real-world SSRF: Youtrack
MAIL FROM: <test@jetbrains.com>.
RCPT To: <josh+ethical@pkc.io>.

Subject: Jetbrains.Hello....
Hello
Real-world SSRF: Nextcloud

- Federated sharing
  - @someone@example.com
Real-world SSRF: Nextcloud

- Federated sharing
  - @someone@example.com
  - @someone@example.com:11211
Real-world SSRF: Nextcloud

- Federated sharing
  - @someone@example.com
  - @someone@example.com:11211
  - Use TLS rebinding, write to memcached!
Real-world SSRF: Nextcloud

▶ Federated sharing
  ▶ @someone@example.com
  ▶ @someone@example.com:11211
  ▶ Use TLS rebinding, write to memcached!
  ▶ Fix: no great options
    ▶ Still added a request timeout and gave me a bounty
Demo: Phishing->CSRF->RCE

Assumptions

- Victim is a developer for a project that makes use of `django.core.cache`, configured to use `memcached`
- Victim views web-based emails in a susceptible browser like Chrome
- Attacker knows/guesses this
- Victim is smart enough not to download attachments
import sys

from django.conf import settings
from django.conf.urls import url
from django.core.management import execute_from_command_line
from django.http import HttpResponse
from django.core.cache import cache as django_cache

settings.configure(
    DEBUG=True,
    ROOT_URLCONF=sys.modules["__name__"],
    CACHES={
        'default': {
            'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
            'LOCATION': '127.0.0.1:11211',
        },
    },
)

rate_limited_sloth()
settings.configure(
    DEBUG=True,
    ROOT_URLCONF=sys.modules[__name__],
    CACHES = {
        'default': {
            'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
            'LOCATION': '127.0.0.1:11211',
        },
    },
)

def rate_limited_sloth(request):
    was_visited = django_cache.get('page_hits', False)
    django_cache.set('page_hits', True, timeout=3)
    if was_visited:
        return HttpResponse('<h1>The sloth needs to sleep for 3 seconds.</h1>')
    return HttpResponse(u'<div style="font-size: 50vh">\U0001f9a5</div>')
Further work

- Chain with memory corruption
- NAT pinning
- DOS amplification
  - High amplification factors?
- Better testing infrastructure
  - infrastructure-as-code
- Image-based CSRF on bad IOT devices
  - telnet?
- Hit internal HTTP servers with a session ticket payload
- Attack message queues
- Correct me – my DM’s are open @joshmdx
Defense
My proposal for TLS clients

- Change cache key
  - Currently: (hostname, port)
  - Better: (hostname, port, ip_addr)
My proposal for TLS clients

- Change cache key
  - Currently: (hostname, port)
  - Better: (hostname, port, ip_addr)
  - If you care about big TLS deployments
    - (hostname, port, addr_type(ip_addr))
  - Similar to https://wicg.github.io/cors-rfc1918/
  - Credit to chromium team
Security costs of TLS session resumption

- “Measuring the Security Harm of TLS Crypto Shortcuts”
  - Detrimental to PFS
- “Tracking Users across the Web via TLS Session Resumption”
  - Detrimental to privacy
- “Insecure TLS session reuse can lead to hostname verification bypass” - NodeJS
  - complexity ➔ bugs
- Also everything in the previous slides
Benefit of TLS session resumption

- Full handshake: ~2x real time, ~23x CPU time
Benefit of TLS session resumption

- Full handshake: ~2x real time, ~23x CPU time

- Might not care if you’re a:
  - Regular internet user
  - Web application making API calls
Disabling outbound TLS session resumption

- libcurl: CURLOPT_SSL_SESSIONID_CACHE=false
- firefox: security.ssl.disable_session_identifiers=true
- Tor browser: disabled by default
- Java, Nodejs, Chrome, others: no option 😞
For web apps that can’t disable it

- Careful around stuff like webhooks, apple pay
- Set up a proxy for outbound requests, e.g. https://github.com/stripe/smokescreen
- Avoid running unauthenticated internal TCP stuff, especially if it’s newline-delimited
Takeaways

- Modern TLS is useful for SSRF attacks
- Following the latest specs is a good way to break things
- We need to reconsider the merits of TLS session resumption
Thank you!

Joshua Maddux, @joshmdx
Security Engineer - latacora.com – security teams for startups