Telescope
Peering Into the Depths of TLS Traffic in Real-Time

Caragea Radu

May 26, 2016
About me

- Bitdefender: Vulnerability Researcher
- PwnThyBytes: CTF team captain
This presentation

Is not about:

- Faults in the TLS protocol/implementation
- Attacks on the crypto level of TLS
This presentation

Is not about:

- Faults in the TLS protocol/implementation
- Attacks on the crypto level of TLS

Is about:

- An experiment into what can be done from the hypervisor (powerful adversary; consider cloud providers)
- How far you can go and how noticeable it would be to an end-user
- Some unexpected results
Operating a honeypot farm
- weak root (!) credentials: do what you will
- analyze traffic (unless it’s encrypted)

Malvertising
- Automated in-browser crawlers
- Scour the net with the hope of getting infected
- Need the infection vector (when served through TLS)
Other solutions: SSLKEYLOGFILE
Other solutions: SSLKEYLOGFILE

Environment Variables

keys.txt - Notepad

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT_RANDOM</td>
<td>aa288229b8c273d5937d60646c6898ab964766b6f33a2ae541029772889eba52729965930874b076af2c2239113b8b9e4a112614fa546cb2951c540f595cda12fab2a968b6d902d887450a80c8b779</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
<tr>
<td></td>
<td>CLIENT_RANDOM</td>
</tr>
</tbody>
</table>
Other solutions: SSLKEYLOGFILE

SSL segment of a reassembled PDU

TCP segment of a reassembled PDU

Frame (7306 bytes)  Decrypted SSL data (296 bytes)
### Other solutions: SSLKEYLOGFILE

<table>
<thead>
<tr>
<th>Frame (7306 bytes)</th>
<th>Decrypted SSL data (296 bytes)</th>
</tr>
</thead>
</table>

| 0000 | 48 54 54 50 2f 31 2e 31 20 32 30 30 20 4f 4b 0d | HTTP/1.1 200 OK. |
| 0010 | 0a 44 61 74 65 3a 20 46 72 69 2c 20 30 38 20 41 | Date: Fri, 08 A |
| 0020 | 70 72 20 30 32 30 32 30 30 20 31 33 3a 33 35 3a 32 39 | pr 2016 13:35:29 |
| 0030 | 20 47 4d 54 0d 0a 53 65 63 6f 72 65 72 3a 20 41 70 | GMT/Se rver: Ap |
| 0040 | 61 63 68 65 2f 32 32 32 32 2e 31 36 61 73 | ache/2.2.16..Las |
| 0050 | 74 2d 4d 6f 64 69 66 69 65 64 3a 20 54 68 75 2c | t-Modifi ed: Thu, |
| 0060 | 30 33 32 30 41 70 72 20 32 30 31 35 3a 30 3a 33 | 05:24 GM T..ETag: |
| 0070 | 30 33 32 30 41 70 72 20 32 30 31 35 3a 30 3a 33 | 05:24 GM T..ETag: |
| 0080 | 30 33 32 30 41 70 72 20 32 30 31 35 3a 30 3a 33 | 05:24 GM T..ETag: |
| 0090 | 30 33 32 30 41 70 72 20 32 30 31 35 3a 30 3a 33 | 05:24 GM T..ETag: |
| 00a0 | 41 63 63 65 70 74 2d 52 | Accept-R anges: b |
| 00b0 | 79 74 65 73 0d 0a 43 6f | ytes..Co ntent-L |
| 00c0 | 6e 77 64 6a 3a 20 36 30 | ngth: 60 1754..Ke |
| 00d0 | 65 70 2d 41 6c 79 66 65 | e-Ac tivity: tim |
| 00e0 | 43 7d 31 35 2c 20 6d 61 | t=15, ma x=100..C |
| 00f0 | 6f 6e 6e 65 63 74 69 66 | onnectio n: Keep- |
| 0100 | 63 6f 69 6e 65 0d 0a 43 | Alive..C ontent-T |
| 0110 | 6f 6e 74 65 66 74 2d 54 | ype: app lication |
| 0120 | 2f 70 6d 64 66 0a 0a 0a | /pdf.... |
Other solutions: SSLKEYLOGFILE

- Implemented in libNSS (firefox) and openSSL (chrome) but not IE/Edge
- The downside: it’s blatantly visible
Other solutions: Custom Root CA
Other solutions: Custom Root CA

Verified by: Superfish, Inc.
Other solutions: Custom Root CA

- Typical solution used by AVs/proxies to intercept TLS traffic
- Visible by malware by scanning the disk.
- Moreover, in "Analyzing Forged SSL Certificates in the Wild" Huang et al show how to do this within the browser
Other solutions: PANDA keyfind plugin

Finding SSL/TLS Master Secrets with PANDA

Introduction

Monitoring SSL/TLS-encrypted traffic is a classic problem for intrusion detection systems. Currently, hypervisor- or network-based IDSes that wish to analyze encrypted traffic must perform a man-in-the-middle attack on the connection, presenting a false server certificate to the client. Not only does this require the client to cooperate by trusting certificates signed by the intrusion detection system, it also takes control of the certificate verification process out of the hands of the client—a dangerous step, given that many existing SSL/TLS interception proxies have a history of certificate trust vulnerabilities.

Instead of a man-in-the-middle attack, we can instead attempt to locate the code that generates SSL/TLS master secret; this secret is sufficient to decrypt any encrypted traffic in a given session, giving us a "man-on-the-inside". Once we have identified the location of the code that generates this secret, we can hook it using any number of standard techniques in order to dump out the master secret. This secret can then be provided to an IDS to decrypt the content of the SSL stream; it may also be provided to a tool like Wireshark to decrypt packet captures after the fact (even if perfect forward secrecy is used).

- really cool solution!
- run the machine under QEMU
- use pre-established "trace points"
- trace memory writes
Other solutions: PANDA keyfind plugin

Place this output into a file named `keyfind_config.txt` in the `panda/qemu` directory. Alternatively, the same information can be derived by hand using a tool like Wireshark and copied into `keyfind_config.txt`, but this is rather more labor intensive.

**Locating the Master Key Code**

Finally, we can run a replay with the `keyfind` plugin enabled to find out what code generates the master secret. Because the `keyfind` plugin tracks the calling function in order to better identify different memory accesses, we also need to enable the `callstack_instr` plugin, which keeps track of function calls and returns. We'll also use QEMU’s VNC output rather than the default SDL because replays don’t show any GUI output.

Using `keyfind` can be quite slow! On my machine, this short session, which takes only 12 seconds to replay with no plugins, takes almost 2 hours to run with `keyfind` enabled. This is what the output looks like:

```bash
brendan@brendantemp:/git/panda/qemu$ echo "begin_replay ssltut" | \
x86_64-softmmu/qemu-system-x86_64 -hda debian_squeeze_i386_desktop_tut.qcow2 \
-m 256 -monitor stdio -vnc :0 -net nic,model=e1000 -net user \
-panda "callstack_instr;keyfind"
Initializing plugin callstack_instr
Initializing plugin keyfind
Couldn't open keyfind_candidates.txt; no key tap candidates defined.
```

- huge overhead
- non portable
Scavenging the keys from memory?

While connection is still active, the keys must still be in memory.
While connection is still active, the keys must still be in memory.

- Problem 1: exact key location is unknown
  - need to dump all memory
  - dumping memory takes time (>10 seconds for 4 GB)
  - multiple connections occur one after the other or interspersed
  - space quickly fills up
Scavenging the keys from memory?

While connection is still active, the keys must still be in memory.

- Problem 1: exact key location is unknown
  - need to dump all memory
  - dumping memory takes time (>10 seconds for 4 GB)
  - multiple connections occur one after the other or interspersed
  - space quickly fills up

- Problem 2: we don’t even know how to distinguish the correct keys from random memory.
Scavenging the keys from memory?

While connection is still active, the keys must still be in memory.

- **Problem 1:** exact key location is unknown
  - need to dump all memory
  - dumping memory takes time ( >10 seconds for 4 GB)
  - multiple connections occur one after the other or interspersed
  - space quickly fills up

- **Problem 2:** we don’t even know how to distinguish the correct keys from random memory.

To understand our approach we must dig deeper!
How exactly does TLS work? Client Hello

Client

Client hello: version, ciphers, client random

Server

Further traffic is now encrypted
How exactly does TLS work? Server Hello

Client Hello:
- version
- ciphers
- client random

Server Hello:
- certificate
- chosen cipher
- server random

Further traffic is now encrypted.
How exactly does TLS work? Client Finished

Client hello: version, ciphers, client random

Server hello: certificate, chosen cipher, server random

Client Finished (first encrypted pkt)

Further traffic is now encrypted
How exactly does TLS work? Server Finished

Client

- Client hello: version, ciphers, client random

Server

- Server hello: certificate, chosen cipher, server random

- Client Finished (first encrypted pkt)

- Server Finished (first encrypted pkt)

Further traffic is now encrypted
How exactly does TLS work? Handshake Complete

**Client**

---

Client hello: version, ciphers, client random

---

Server hello: certificate, chosen cipher, server random

---

Client Finished (first encrypted pkt)

---

Server Finished (first encrypted pkt)

---

Further traffic is now encrypted

**Server**
Excerpt from RFC5246/4346:

To generate the key material, compute
key_block = PRF(SecurityParameters.master_secret,
    "key expansion",
    SecurityParameters.server_random +
    SecurityParameters.client_random);

until enough output has been generated.
Key material = [client_write_MAC_key][server_write_MAC_key]
    [client_write_key][server_write_key]
    [client_write_IV][server_write_IV]
Key events in the TLS handshake

- **Client**
  - Client hello: version, ciphers, client
  - Client Finished (first encrypted pkt)

- **Server**
  - Server hello: certificate, chosen cipher, server random
  - Server Finished (first encrypted pkt)
  - Further traffic is now encrypted

Keys have not been generated

Keys have been generated
Implications

- Only track memory between events
- Events signalled by passing through netfilter queue
- Dramatic decrease in memdump size
Implications

- Only track memory between events
- Events signalled by passing through netfilter queue
- Dramatic decrease in memdump size
- But how do you actually "track" pages?
Think VM live migration

Logdirty mechanism

\[ t_0 \]: start tracking pages written to from \( t_0 \) and flush the RAM to the target on the network when this finishes, get the "dirty" pages (at \( t_1 \)) and send the delta to target again. Repeat this for every \( t_i - t_{i+1} \) until the number of pages is under threshold. Stop VM1, do iteration one last time, start VM2.
Think VM live migration

Logdirty mechanism

- $t_0$: start tracking pages written to from $t_0$ and flush the RAM to the target on the network
Think VM live migration

Logdirty mechanism

- $t_0$: start tracking pages written to from $t_0$ and flush the RAM to the target on the network
- when this finishes, get the "dirty" pages (at $t_1$) and send the delta to target again
Think VM live migration

Logdirty mechanism
- $t_0$: start tracking pages written to from $t_0$ and flush the RAM to the target on the network
- when this finishes, get the "dirty" pages (at $t_1$) and send the delta to target again
- repeat this for every $t_i - t_{i+1}$ until number of pages is under threshold
Think VM live migration

Logdirty mechanism

- $t_0$: start tracking pages written to from $t_0$ and flush the RAM to the target on the network
- when this finishes, get the "dirty" pages (at $t_1$) and send the delta to target again
- repeat this for every $t_i - t_{i+1}$ until number of pages is under threshold
- stop VM1, do iteration one last time, start VM2
Similar mechanisms exist in most (all) modern hypervisors that support VM migration

- Page fault based (basic)
- EPT A/D
Similar mechanisms exist in most (all) modern hypervisors that support VM migration

- Page fault based (basic)
- EPT A/D
- Recently, a processor extension especially for this: Intel PML. Convenient, right ??
Putting the TeLeScope together

- Filter target network events and send to netfilter queue
- Start logging on Server Hello
- Stop logging and dump pages on Client Finished
- The result is a micro-memdump
- Can be processed **offline** anytime
TeLeScope results

- on Linux VM per connection: 500K - 10 MB memdump
- on Windows VM per connection: 15 MB - 60 MB memdump
- VM pause time: under 0.5 ms but on average 0.05 ms
- page dump time: 1-10 ms (disguised as packet delay)
Problem 1: you don’t know where the keys are (partially solved)
- need to dump all memory
- dumping memory takes time (\(>10\) seconds for 4 GB)
- multiple connections occur one after the other or interspersed
- space quickly fills up

Problem 2: we don’t even know how to distinguish the correct keys from random memory.
Problem 2: key discerning

Apparently multiple unknowns

- key format
- key parameters: IV, nonce, etc
- what to encrypt/decrypt
- what it decrypts to
Known Plaintext Attack

The Client/Server Finished messages have a fixed form:
14 00 00 0C [12 random bytes]
Known Plaintext Attack

The Client/Server Finished messages have a fixed form:

14 00 00 0C [12 random bytes]

- $\frac{1}{2^{32}}$ chance of a False Positive
- This works if you can decrypt the first 4 bytes (think stream ciphers, AES/CTR, etc)
Alexa top 1000 ciphers

- 5% RC4
- 21% AES CBC
- 73% AES GCM
AES CBC

- Decrypting each block depends on having the previous block
- For the first block you need the IV (not explicit for TLS 1.0)
- The known plaintext is exactly in the first block
- Decrypting each block depends on having the previous block.
- For the first block you need the IV (not explicit for TLS 1.0).
- The known plaintext is exactly in the first block.
- We use the last block for the padding.

```
0000 14 00 00 0c 7f a6 87 08 ab 2e 6c 32 fd ba f7 c9 ............l2....
0010 9c c5 76 20 da 83 6e b5 27 af ac e0 ac 1a e4 ..v ..n.'........
0020 8c 86 55 fc 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b ..........U.......
```
AES GCM/CTR
AES GCM/CTR Encryption

Counter 0 → incr → Counter 1 → incr → Counter 2

$E_K$

Plaintext 1 → $E_K$ → Ciphertext 1

Plaintext 2 → $E_K$ → Ciphertext 2

AES CTR
AES GCM/CTR Authentication

Counter 0

$E_K$

Ciphertext 1

$\text{mult}_H$

Auth Data 1

Ciphertext 2

$\text{mult}_H$

$\text{len}(A) \parallel \text{len}(C)$

$\text{mult}_H$

Auth Tag

AES GCM
AES GCM bruteforce attempt

- Counter is [8 bytes key material][8 bytes counter]
- Here: 0fddf45e89838e70000000000000000001
- The first half is also from the key material
- Implies $O(N^2)$ which we don’t like!
• Auth Data, the Ciphertexts, Lengths and the Tag Known
• Each key $K$ corresponds to one $H$
• Reverse the Flow and do one extra decryption
• Known Plaintext Attack on Counter format
Telesync benchmarks

- 1 thread, 240 MB completely random data: 6750 ms
- 6 threads, 240 MB completely random data: 557 ms (12x speedup)
- 6 threads + heuristics, 240 MB typical memdump data: 151 ms (44x speedup)

However, a typical memdump size is usually 5-10 times smaller. So we can usually consume as fast as we can produce!
Telesync benchmarks

- 1 thread, 240 MB completely random data: 6750 ms
- 6 threads, 240 MB completely random data: 557 ms (12x speedup)
- 6 threads + heuristics, 240 MB typical memdump data: 151 ms (44x speedup)

However, a typical memdump size is usually 5-10 times smaller. So we can usually consume as fast as we can produce!
Demo 1 (manual)
Demo 2 (integrated)
Actually, this can be applied to other protocols that use a similar negotiation technique for the symmetric keys:

- VPN
- SSH
- Tor
Conclusions and more

- Decrypting TLS on current implementations is definitely feasible with a hypervisor-in-the-middle attack
- We developed a fast and efficient PoC
- * You might not observe if you are the one "under scrutiny" on a VPS
- * Actually, if you’re not in control of the bare metal all bets are off
Questions