Lucky Microseconds

Martin Albrecht and Kenny Paterson
Information Security Group
@kennyog; www.isg.rhul.ac.uk/~kp
s2n

• s2n is a new implementation of TLS from AWS (Amazon Web Services).
• Nice logo!
• Source code released on github June 30\textsuperscript{th} 2015.
• 6,000 lines of C instead of 70,000 lines in OpenSSL.
• Three external security audits/code reviews were performed before release.
s2n press at launch

About 297 results (0.25 seconds)

**AWS security looks to avoid cloud reboots with s2n**
TechTarget - Jun 30, 2015
Amazon Web Services (AWS) unveiled s2n on its security blog this week. Signal to Noise (s2n) is meant to be a simplified, more easily...

Amazon's s2n encryption library aims to be small, light, and auditable
InfoWorld - Jun 30, 2015
Amazon releases open source cryptographic module
PCWorld - Jun 30, 2015

Amazon introduces new open-source TLS implementation 's2n'
ZDNet - Jun 30, 2015

Amazon Releases S2N TLS Crypto Implementation to Open Source
Threatpost - Jun 30, 2015

Explore in depth (17 more articles)
s2n and CBC-mode encryption

• s2n implements SSLv3 and TLS 1.0, 1.1 and 1.2.
• So supports CBC-mode encryption.
• Lucky 13:
  • Timing attack based on low-level internals of cryptographic processing for CBC-mode.
• Countermeasures to Lucky 13 in OpenSSL needed 500 lines of code.
• Our first reaction: there’s no way s2n can be secure against Lucky 13 in just 6 kLoC!
TLS Record Protocol: MAC-Encode-Encrypt (MEE)

Problem: how to parse unauthenticated plaintext as payload, padding and MAC fields without leaking any information via error messages, timing or anything else?
Constant Time Decryption for MEE

- Lucky 13 exploits leakage from TLS’s MEE decryption processing for CBC-mode.
- Proper constant-time, constant-memory access implementation is needed to fully prevent it.
- Hard when plaintext is a mix of unauthenticated padding, MAC and payload fragment.
- See Adam Langley’s blogpost at:
  
  https://www.imperialviolet.org/2013/02/04/luckythirteen.html

  for full details on how Lucky 13 was fixed in OpenSSL and NSS.
- **TL;DR:** it’s a bit of a nightmare to do it properly.
s2n and Lucky 13

- s2n protected against Lucky 13 using two countermeasures:
  - Dummy HMAC computations and padding checks to try to equalise running time.
  - Addition of random timing delays on decryption failure, to mask any residual timing differences.

- Each countermeasure had a problem...
s2n_verify_cbc

```c
int payload_and_padding_size = decrypted->size - mac_digest_size;

/* Determine what the padding length is */
uint8_t padding_length = decrypted->data[decrypted->size - 1];

int payload_length = payload_and_padding_size - padding_length - 1;
if (payload_length < 0) {
    payload_length = 0;
}

/* Update the MAC */
GUARD(s2n_hmac_update(hmac,
GUARD(s2n_hmac_copy(&copy,

/* Check the MAC */
uint8_t check_digest[S2N_MAX_DIGEST_LEN];
lte_check(mac_digest_size, sizeof(check_digest));
GUARD(s2n_hmac_digest(hmac, check_digest, mac_digest_size));
```

Uses the last byte of the last block to decide how long padding should be. 
Sets `payload_length` by subtracting this value from total size. 
(Padding check done later.)
Updates the HMAC value (but does not yet finalise it).

`payload_length` bytes are passed to HMAC here.
int payload_and_padding_size = decrypted->size - mac_digest_size;

/* Determine what the padding length is */
uint8_t padding_length = decrypted->data[decrypted->size - 1];

int payload_length = payload_and_padding_size - padding_length - 1;
if (payload_length < 0) {
    payload_length = 0;
}

/* Update the MAC */
GUARD(s2n_hmac_update(hmac, decrypted->data, payload_length));
GUARD(s2n_hmac_copy(&copy, hmac));

/* Check */
uint8_t lte_check[MAC_T_LEN];
lte_check[0] = s2n_hmac_verify(copy, check_digest));
GUARD(s2n_verify_cbc);
Finalises the HMAC value. Running time depends on value of `payload_length`, which in turn depends on `padding_length`, which might leak plaintext information.
Compares (in constant time!) the computed HMAC value to the one extracted from `decrypted->data`. 
Perform dummy `hmac_update` operations to equalise running time of HMAC.
Let’s build a magic ciphertext!

\[ IV \quad R_1 \quad R_2 \quad R_3 \quad C_{t-1} \quad C_t \]

XOR 1-byte \( \Delta \) here and submit for decryption
Case 1: last byte is 00, 01, 02, 03, 04

\[ \text{SQN} || \text{HDR} \geq 13 + 16 + 16 + 11 = 56 \text{ bytes} \]

\[ \leq 5 \text{ bytes} \]

5 SHA-256 compression function evaluations

XOR 1-byte \( \Delta \) here and submit for decryption

32 bytes

00, 01, 02, 03 or 04
Case 2: last byte is 05, 06,..., FF

SQN||HDR

$\leq 13 + 16 + 16 + 10 = 55$ bytes

4 SHA-256 compression function evaluations

32 bytes

$\geq 6$ bytes

05, 06,..., FF

XOR 1-byte $\Delta$ here and submit for decryption
So there’s a timing difference for the entire HMAC computation depending on whether the last byte is in \{00, 01, 02, 03, 04\} or in \{05, 06, ..., FF\}. But this last byte relates to the corresponding target plaintext byte in a controlled way. The timing difference is of the same size as in the original Lucky 13 attack.

But what about that equalisation code, using dummy call to `hmac_update`?
For the magic ciphertexts, the input size is always 60 bytes. So zero extra HMAC compression function computations are done, in either case!
Experimental results: timing `s2n_verify_cbc`

<table>
<thead>
<tr>
<th>Byte value</th>
<th>Cycles</th>
<th>Byte value</th>
<th>Cycles</th>
<th>Byte value</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>2251.96</td>
<td>0x05</td>
<td>1746.49</td>
<td>0xfc</td>
<td>1640.79</td>
</tr>
<tr>
<td>0x01</td>
<td>2354.57</td>
<td>0x06</td>
<td>1747.65</td>
<td>0xfd</td>
<td>1634.61</td>
</tr>
<tr>
<td>0x02</td>
<td>2252.07</td>
<td>0x07</td>
<td>1705.62</td>
<td>0xfe</td>
<td>1648.70</td>
</tr>
<tr>
<td>0x03</td>
<td>2135.11</td>
<td>0x08</td>
<td>1808.73</td>
<td>0xff</td>
<td>1634.64</td>
</tr>
<tr>
<td>0x04</td>
<td>2130.02</td>
<td>0x09</td>
<td>1806.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Timing of function `s2n_verify_cbc` (in cycles) with $H = \text{SHA-256}$ for different values of last byte in the decrypted buffer, each cycle count averaged over $2^8$ trials.
Rebooting Lucky 13

- The timing differences would allow for a novel variant of the original Lucky 13 attack to be mounted against the `s2n_verify_cbc` code.
- The attack would recover the last byte of any target block of plaintext.
- Can be upgraded to full plaintext recovery for session cookies using malicious Javascript running in the browser.
- Can be adapted to HMAC-SHA-1 and HMAC-MD-5.
- Can be executed remotely over a network by timing TLS error messages.
  - Attack is in the “challenging but not impossible” category.
But wait …. random timing delays in s2n!

- Addition of random timing delay in event of cryptographic processing error.
- Intended to mask any residual timing differences from s2n_verify_cbc.
- Time delay is a random value between 0 and 10 seconds.
- Is that enough to mask a difference of ~500 clock cycles?
- **Textbook statistical analysis:**
  \[ N \geq \sigma^2 + cT^2 \]
- **Outcome:** trillions of samples would be needed to detect any timing differences if the delay was *uniformly* random.
Generating random timing delays in s2n

```c
s2n_recv.c

36  int s2n_read_full_record(struct s2n_connection *conn, \
    uint8_t *record_type, int *isSSLv2)
97  /* Decrypt and parse the record */
98  if (s2n_record_parse(conn) < 0) {
99    GUARD(s2n_connection_wipe(conn));
100   if (conn->blinding == S2N_BUILT_IN_BLINDING) {
101     int delay;
102     GUARD(delay = s2n_connection_get_delay(conn));
103     GUARD(sleep(delay / 1000000));
104     GUARD(usleep(delay % 1000000));
105   }
106   return -1;
107 }
```
Generating random timing delays in s2n

```
s2n_recv.c
36  int s2n_
97  /* Decrypt and parse the record */
98  if (s2n_record_parse(conn) < 0) {
99    GUARD(s2n_connection_wipe(conn));
100   if (conn->blinding == S2N_BUILT_IN_BLINDING) {
101     int delay;
102     GUARD(delay = s2n_connection_get_delay(conn));
103     GUARD(sleep(delay / 10000000));
104     GUARD(usleep(delay % 10000000));
105     return -1;
```

Generates random delay, uses calls to RNG + rejection sampling.

And even more stuff!

Yet more stuff – yuck!

Sleep for whole number of seconds

Sleep for whole number of microseconds

It’s messy, but it’s not necessarily uniform!
Two observations + reality

• We can filter out any noise arising from sleep() call by just ignoring any delays larger than 1 second.
  • Effect is to increase number of samples needed by factor of 10.

• Delay from usleep() is a whole number of microseconds, but the timing signal we are looking for is just a few hundred clock cycles.
  • So take all timing measurements modulo 1 microsecond (3300 clock cycles), and only the signal will remain!
Two observations + reality

• In reality, things are a bit harder than this:
  • `usleep()` does not give a delay that is an exact number of microseconds, but has its own complex distribution.
  • Several additional noise sources to contend with.
  • Platform-dependent behaviour.
Random timing delays in s2n

Figure 8: Distribution of clock ticks modulo 3,300 for timing signals on Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz with the maximum delay restricted to $d = 100,000$. 
Putting it all together

• KL divergence: $3.6 \times 10^{-3}$.

• Hence about 280 ciphertexts are needed to distinguish Ox00 from Ox05, for max delay 100,000 μs.

• So 28k ciphertexts in reality.
  • $10,000,000/100,000 = 100$, so we only use 1 in 100 samples.

• Extends to plaintext recovery attack using a standard maximum likelihood based approach.

• But more samples are needed because now we are trying to identify one correct value amongst 255 wrong values.
Disclosure and interaction with AWS

- s2n was released on June 30\textsuperscript{th} 2015.
- We informed the AWS team about the HMAC processing error in \texttt{s2n\_verify\_cbc} on July 5\textsuperscript{th} 2015.
- AWS patched the s2n code almost immediately.
- They also informed us about their random timing delay countermeasure.
- So we broke that too....
- Meanwhile, AWS switched to using \texttt{nanosleep()}.
- Code as released was vulnerable but AWS say that no production systems could have been attacked.
- Disclosure process was very smooth.
Takeaways

• Lucky 13 is hard to fully protect against.
• OpenSSL does it, but the code is not very.... transparent.
• Don’t mess with MEE unless you really know what you’re doing!
• Pre-release code audits will not catch all subtle crypto flaws.
• AWS invited public analysis of their code and reacted well to our work.
More information

**Paper:**
http://eprint.iacr.org/2015/1129

**Press:**

**Martin’s blog:**
https://martinralbrecht.wordpress.com/2015/11/24/lucky-microseconds-a-timing-attack-on-amazons-s2n-implementation-of-tls/

**AWS blog:**