Lucky Microseconds

Martin R. Albrecht and Kenny Paterson

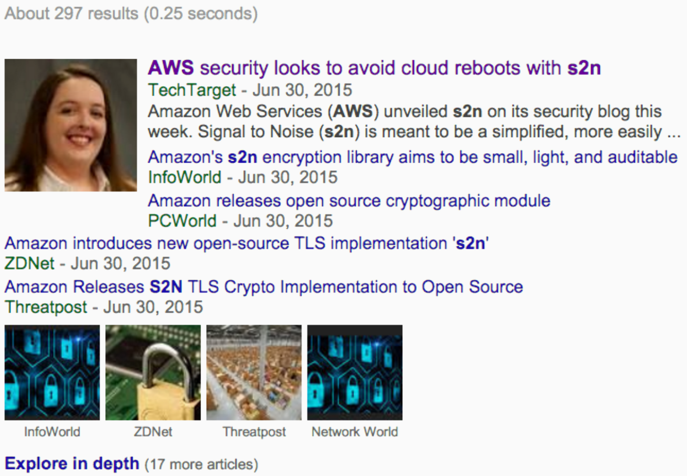
Eurocrypt 2016

Information Security Group, Royal Holloway, University of London

s2n

- s2n is a new implementation of TLS from Amazon Web Services (AWS).
- Source code released on GitHub on 30 June 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

InfoWorld
ZDNet
Threatpost
Network World

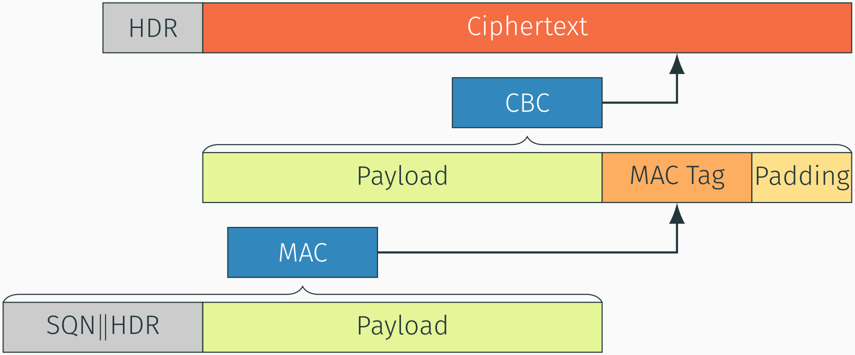
Explore in depth (17 more articles)

s2n and CBC-mode encryption

- `s2n` implements SSLv3 and TLS 1.0, 1.1 and 1.2.
- So supports CBC-mode encryption.
- Lucky 13:[1]
    - Timing attack based on low-level internals of cryptographic processing for CBC-mode.
- Countermeasures to Lucky 13 in OpenSSL needed 500 lines of code.
- Can `s2n` be secure against Lucky 13 in just 6 kLoC?

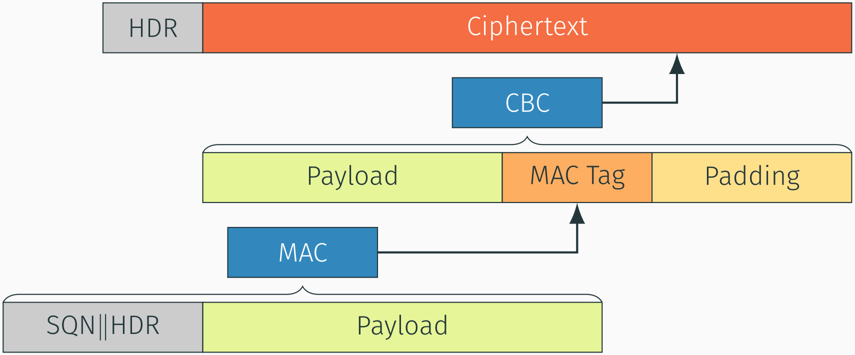
[1]Nadhem AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P 2013)*. Ed. by Robin Sommer. San Diego, CA, USA: IEEE Press, May 2013, pp. 526–540. DOI: `10.1109/SP.2013.42`.

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?

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[2]

TL;DR

It's hard to do it properly.

[2] `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`

It's hard to do it properly, revisited (3/5/16)

```
Padding oracle in AES-NI CBC MAC check (CVE-2016-2107)
======================================================

Severity: High

A MITM attacker can use a padding oracle attack to decrypt traffic
when the connection uses an AES CBC cipher and the server support
AES-NI.

This issue was introduced as part of the fix for Lucky 13 padding
attack (CVE-2013-0169). The padding check was rewritten to be in
constant time by making sure that always the same bytes are read and
compared against either the MAC or padding bytes. But it no longer
checked that there was enough data to have both the MAC and padding
bytes.

OpenSSL 1.0.2 users should upgrade to 1.0.2h
OpenSSL 1.0.1 users should upgrade to 1.0.1t

This issue was reported to OpenSSL on 13th of April 2016 by Juraj
Somorovsky using TLS-Attacker. The fix was developed by Kurt Roeckx
of the OpenSSL development team.
```

https://www.openssl.org/news/secadv/20160503.txt

https://github.com/RUB-NDS/TLS-Attacker

s2n and Lucky 13

The version of `s2n` we looked at protected against Lucky 13 using two countermeasures:

1. Dummy HMAC computations and padding checks to equalise running time.
2. Addition of random timing delays on decryption failure, to mask any residual timing differences.

s2n_verify_cbc

```
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];
```

Use the last byte of the last block to decide padding length.

```
int payload_length = payload_and_padding_size - padding_length - 1;
```

Set `payload_length` by subtracting this value from total size.

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

Update MAC (but do not finalise), passing `payload_length` bytes.

s2n_verify_cbc

```
GUARD(s2n_hmac_copy(&copy, hmac));
```

Make copy of HMAC data structure for later time equalisation.

```
/* 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));
```

Finalises MAC value.

Running time depends on value of `payload_length`, which in turn depends on `padding_length`, which might leak plaintext information.

s2n_verify_cbc

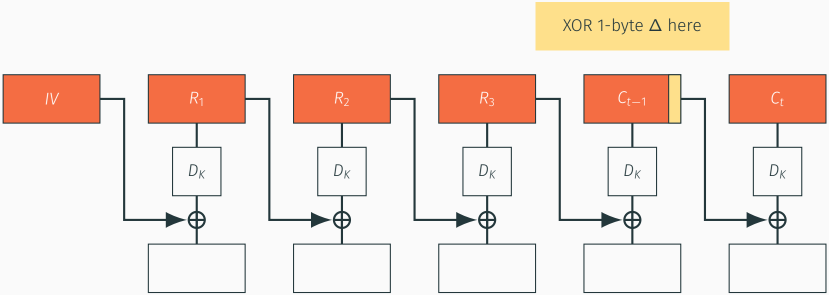
```
int mismatches = s2n_constant_time_equals(decrypted->data
                                          + payload_length, check_digest,
                                          mac_digest_size) ^ 1;
```

Constant-time compare the computed HMAC value to the one extracted from `decrypted->data`.

```
/* Compute a MAC on the rest of the data so that we perform
   the same number of hash operations */
GUARD(s2n_hmac_update(&copy,
                      decrypted->data + payload_length + mac_digest_size,
                      decrypted->size - payload_length - mac_digest_size
                         - 1));
```

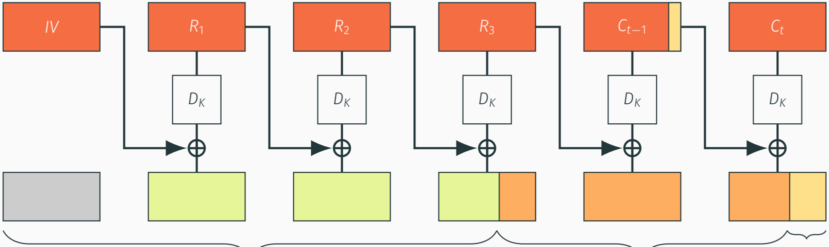
Perform dummy `hmac_update` operations to equalise running time of HMAC.

Building Attack Ciphertext



CASE 1: LAST BYTE ≤ 4

XOR 1-byte Δ here



$\geq 13 + 16 + 16 + 11 = 56$ bytes

32 bytes

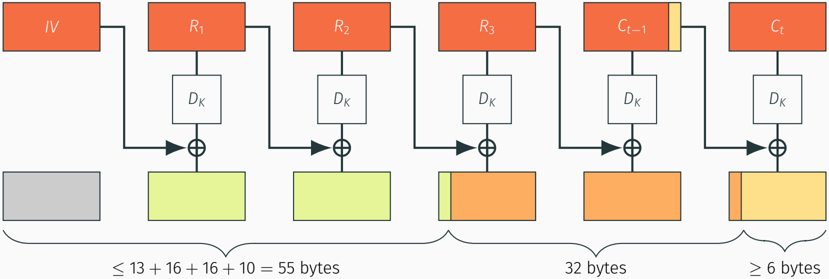
≤ 5 bytes

5 SHA-256 compression function calls

00, 01, 02, 03, 04

CASE 2: LAST BYTE ≥ 5

XOR 1-byte Δ here



4 SHA-256 compression function calls

05, 06, 07, . . ., ff

Dummy HMAC computations in s2n

- There is 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`}.
- 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`?

s2n_verify_cbc

```
/* Compute a MAC on the rest of the data so that we perform
   the same number of hash operations */
GUARD(s2n_hmac_update(&copy,
                      decrypted->data + payload_length + mac_digest_size,
                      decrypted->size - payload_length - mac_digest_size
                         - 1));
```

For our 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`

Byte value	Cycles	Byte value	Cycles	Byte value	Cycles
`00`	2251.96	`05`	1746.49	...	...
`01`	2354.57	`06`	1747.65	`fc`	1640.79
`02`	2252.07	`07`	1705.62	`fd`	1634.61
`03`	2135.11	`08`	1808.73	`fe`	1648.70
`04`	2130.02	`09`	1806.50	`ff`	1634.64

Timing of function `s2n_verify_cbc` (in cycles) with $H =$ `SHA-256` for different values of last byte in the `decrypted` buffer, each cycle count averaged over 2^8 trials.

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 ~300 clock cycles?

Textbook statistical analysis

Trillions of samples would be needed to detect any timing differences if the delay was uniformly random.

Generating random timing delays in s2n

```
if (s2n_record_parse(conn) < 0) {
    conn->closed = 1;
    GUARD(s2n_connection_wipe(conn));
```

Overwriting memory: varying time.

```
    if (conn->blinding == S2N_BUILT_IN_BLINDING) {
        int delay;
        GUARD(delay = s2n_connection_get_delay(conn));
```

Generate random delay (uses calls to RNG + rejection sampling)

```
        GUARD(sleep(delay / 1000000));
```

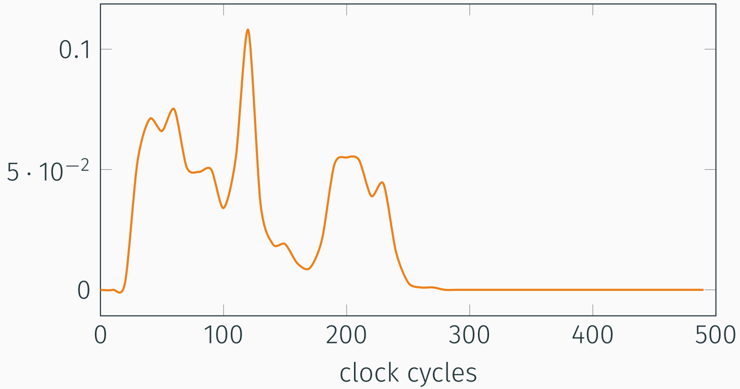
Sleep for at least number of seconds

```
        GUARD(usleep(delay % 1000000));
    }
    return -1;
}
```

Sleep for at least number of microseconds

Characterising the noise: `sleep()`

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



Distribution of clock ticks for `sleep(0)` on Intel® Xeon® CPU E5-2667 v2 @ 3.30GHz.

Characterising the noise: `usleep()`

> *The* `usleep()` *function suspends execution of the calling thread for (at least) usec microseconds. . . .*

The timing signal we are looking for is just a few hundred clock cycles. So take all timing measurements modulo 1 microsecond (3,300 clock cycles on test system).

Characterising the noise: `usleep()`

> *The* `usleep()` *function suspends execution of the calling thread for (at least) usec microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.*

`usleep()` does not give a delay that is an exact number of microseconds, but has its own complex distribution.

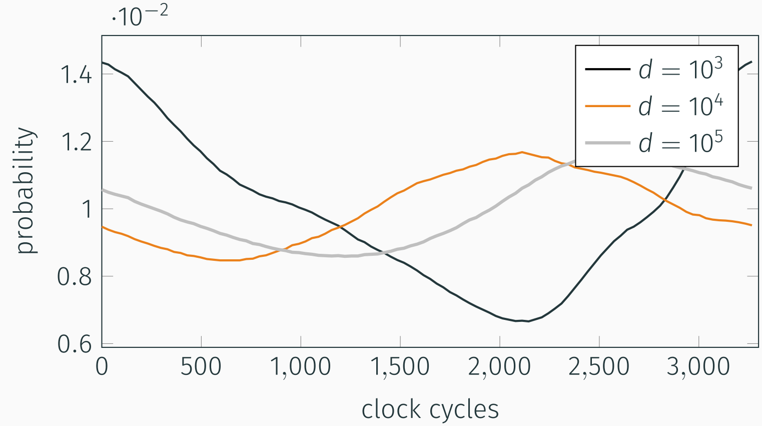
Characterising the noise: `usleep()`

> *The* `usleep()` *function suspends execution of the calling thread for (at least) usec microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.*

`usleep()` does not give a delay that is an exact number of microseconds, but has its own complex distribution.

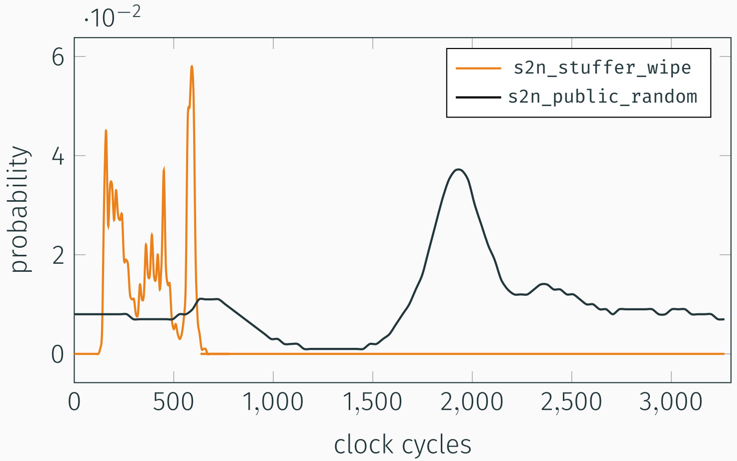
However, despite this, `usleep()` does show exploitable non-uniform behaviour on the systems we tested.

CHARACTERISING THE NOISE: `usleep()`



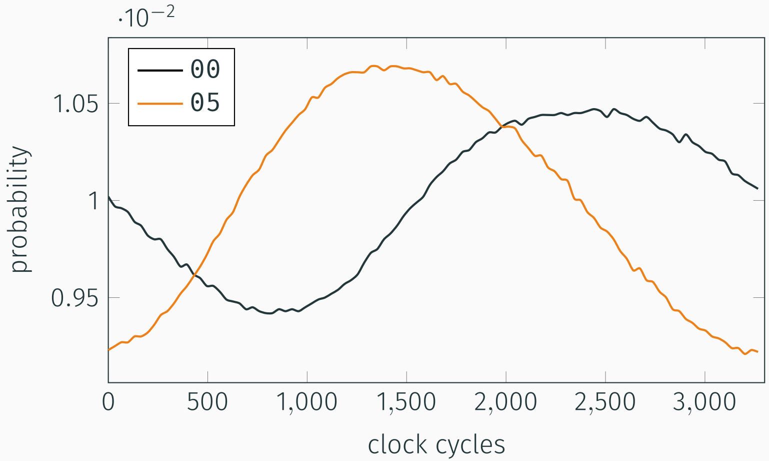
Distribution of clock ticks modulo 3,300 for `usleep(delay)` with `delay` uniformly random in $[0, d)$.

Characterising the noise: wiping & random sampling



Distribution of clock ticks modulo 3,300.

Random timing delays in s2n



Distribution of clock ticks modulo 3,300 for timing signals with the maximum delay restricted to $d = 100,000$.

Putting it all together

- Kullback–Leibler divergence: 3.6×10^{-3}.
- Hence about 280 ciphertexts are needed to distinguish `00` from `05`, for max delay $100,000 \mu s$.
- 28k ciphertexts for real delay of 10s.
- Extends to plaintext recovery attack using a standard maximum likelihood based approach.
- 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 30th 2015.
- We informed the AWS team about the HMAC processing error in `s2n_verify_cbc` on July 5th 2015.
- AWS patched the `s2n` code almost immediately.
- They also informed us about their random timing delay countermeasure, which prompted our study of this countermeasure.
- Meanwhile, `s2n` switched to using `nanosleep()`.
- Disclosure process was very smooth.

`nanonsleep()`

- Our experiments indicate that the distribution of `nanonsleep()` as implemented on Linux is sufficiently close to uniform to thwart the attack.
- However, relying on it puts a high security burden on this function which is not designed for this purpose.

  `nanosleep()` *suspends the execution of the calling thread until either at least the time specified in* `*req` *has elapsed, or the delivery of a signal that triggers the invocation of a handler in the calling thread or that terminates the process.*

Takeaways

- Lucky 13 is hard to fully protect against.
- OpenSSL does it, but the code is not transparent $\rightarrow$ CVE-2016-2107
- MEE makes it hard to produce secure implementations, it should be phased out as soon as possible.
- Pre-release code audits will not catch all subtle crypto flaws.
- AWS invited public analysis of their code and reacted well to our work.

Fin

Thank You

s2n `https://github.com/awslabs/s2n`

Paper `http://ia.cr/2015/1129`

Press `http://arstechnica.com/science/2015/11/researchers-poke`

AWS Blog `https://blogs.aws.amazon.com/security/blog/tag/s2n`