Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure

Eyal Ronen, Kenny Paterson, Adi Shamir
Talk Outline

1. TLS and CBC_HMAC ciphersuite
Talk Outline

1. TLS and CBC_HMAC ciphersuite
2. Side channel attack mitigations: Pseudo Vs Fully constant time
Talk Outline

1. TLS and CBC_HMAC ciphersuite
2. Side channel attack mitigations: Pseudo Vs Fully constant time
3. Padding attack on CBC_HMAC
Talk Outline

1. TLS and CBC_HMAC ciphersuite
2. Side channel attack mitigations: Pseudo Vs Fully constant time
3. Padding attack on CBC_HMAC
4. New cache attacks on CBC_HMAC
Transport Layer Security (TLS)

- The most widely used cryptographic protocol
- Provides communication security (https, VPN, etc.)
  - TLS handshake is used for authentication and secure key exchange
  - TLS Record layer protects the communication
- Allows for cryptographic agility using different cipher suites
Transport Record Layer

- Handshake Protocol
- Change Cipher Spec Protocol
- Alert Protocol
- HTTP, other apps

Record Protocol

TCP
CBC_HMAC Ciphersuite in TLS
CBC_HMAC Ciphersuite in TLS

• Implements the HMAC-then-CBC scheme
CBC_HMAC Ciphersuite in TLS

• Implements the HMAC-then-CBC scheme
• Once the most popular TLS record cipher suite
CBC_HMAC Ciphersuite in TLS

• Implements the HMAC-then-CBC scheme
• Once the most popular TLS record cipher suite
• Long history of practical implementation attacks
CBC_HMAC Ciphersuite in TLS

- Implements the HMAC-then-CBC scheme
- Once the most popular TLS record cipher suite
- Long history of practical implementation attacks

- Still widely used (Oct 2018)
  - ~8% by Mozilla's Telemetry
  - ~11% by ICSI Certificate Notary
CBC_HMAC Ciphersuite in TLS

• Implements the HMAC-then-CBC scheme
• Once the most popular TLS record cipher suite
• Long history of practical implementation attacks

• Still widely used (Oct 2018)
  • ~8% by Mozilla's Telemetry
  • ~11% by ICSI Certificate Notary
• Better alternatives now available (e.g. AES-GCM)
CBC_HMAC Ciphersuite in TLS

• Implements the HMAC-then-CBC scheme
• Once the most popular TLS record cipher suite
• Long history of practical implementation attacks

• Still widely used (Oct 2018)
  • ~8% by Mozilla's Telemetry
  • ~11% by ICSI Certificate Notary
• Better alternatives now available (e.g. AES-GCM)
• Supported for backwards compatibility
Crypto Scheme Vs Implementation

- HMAC-then-CBC *functionality* for TLS is secure* [Krawczyk01, PRS11]
Crypto Scheme Vs Implementation

Diagram showing an implementation box with input and output arrows connecting to a secret component within the implementation box.
Crypto Scheme Vs Implementation

- Securely implementing CBC_HMAC for TLS is hard
  - Padding oracle attacks due to non constant time implementation
  - All implementations were vulnerable to Lucky 13 [AP 2013]
  - Multiple rounds of attacks and patches
Side channels attack mitigations
Side channels attack mitigations

• Secret values should not change the code flow in any way measurable by attacker
Side channels attack mitigations

• Secret values should not change the code flow in any way measurable by attacker
• Attacker might be able to see error messages, measure running time, detect memory access patterns, and more
Side channels attack mitigations

- Secret values should not change the code flow in any way measurable by attacker
- Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

If \texttt{SecretValue} == 0 then \texttt{Send2Attacker(“Bad secret value!”)}
Side channels attack mitigations

• Secret values should not change the code flow in any way measurable by attacker
• Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

If SecretValue == 0 then Send2Attacker(“Bad secret value!”)
Side channels attack mitigations

• Secret values should not change the code flow in any way measurable by attacker
• Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

\[\text{If } \text{SecretValue} \equiv 0 \text{ then } \text{Send2Attacker("Bad secret value!") }\]
\[\text{If } \text{KeyBits}[1] \equiv 1 \text{ then } \text{SlowFunction()} \]
\[\text{else } \text{FastFunction()} \]
Side channels attack mitigations

- Secret values should not change the code flow in any way measurable by attacker
- Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

If SecretValue == 0 then Send2Attacker("Bad secret value!")
If KeyBits[1] == 1 then SlowFunction()
else FastFunction()
Side channels attack mitigations

• Secret values should not change the code flow in any way measurable by attacker

• Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

If SecretValue == 0 then Send2Attacker(“Bad secret value!”)
If KeyBits[1] == 1 then SlowFunction()
else FastFunction()

Result = MyTable[KeyBytes[5]]
Side channels attack mitigations

• Secret values should not change the code flow in any way measurable by attacker
• Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

If `SecretValue == 0` then `Send2Attacker(“Bad secret value!”)`
If `KeyBits[1] == 1` then `SlowFunction()`
Else `FastFuntion()`

Result = `MyTable[KeyBytes[5]]`
Side channels attack mitigations

- Secret values should not change the code flow in any way measurable by attacker.
- Attacker might be able to see error messages, measure running time, detect memory access patterns, and more.

```c
if SecretValue == 0 then Send2Attacker("Bad secret value!")
if KeyBits[1] == 1 then SlowFunction()
else FastFunction()
Result = MyTable[KeyBytes[5]]
```
Side channels attack mitigations

- Secret values should not change the code flow in any way measurable by attacker
- Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

```plaintext
If SecretValue == 0 then Send2Attacker("Bad secret value!")
If KeyBits[1] == 1 then SlowFunction()
else FastFunction()
Result = MyTable[KeyBytes[5]]
```
Side channels attack mitigations

- Secret values should not change the code flow in any way measurable by attacker
- Attacker might be able to see error messages, measure running time, detect memory access patterns, and more

```plaintext
If SecretValue == 0 then Send2Attacker("Bad secret value!")
If KeyBits[1] == 1 then SlowFunction()
else FastFunction()
Result = MyTable[KeyBytes[5]]
```
Pseudo Vs Fully Constant time
Pseudo Vs Fully Constant time

Full Constant time
Pseudo Vs Fully Constant time

Full Constant time  Pseudo Constant time
Pseudo Vs Fully Constant time

Full Constant time

Pseudo Constant time

• Program flow independent from secret values
Pseudo Vs Fully Constant time

Full Constant time
- Program flow independent from secret values
- Blocks all currently known classes of attacks*

Pseudo Constant time
Pseudo Vs Fully Constant time

**Full Constant time**
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test

**Pseudo Constant time**
Pseudo Vs Fully Constant time

**Full Constant time**
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity

**Pseudo Constant time**
Pseudo Vs Fully Constant time

**Full Constant time**
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity
  - Hard to write/review

**Pseudo Constant time**
Pseudo Vs Fully Constant time

**Full Constant time**
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity
  - Hard to write/review
  - OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

**Pseudo Constant time**
Pseudo Vs Fully Constant time

Full Constant time
• Program flow independent from secret values
• Blocks all currently known classes of attacks*
• “Full” is easy to test
• Very high code complexity
  • Hard to write/review
  • OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

Pseudo Constant time
• Mask program flow dependencies on secret values
Pseudo Vs Fully Constant time

Full Constant time
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity
  - Hard to write/review
  - OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

Pseudo Constant time
- Mask program flow dependencies on secret values
- Blocks only currently implemented attacks
Pseudo Vs Fully Constant time

**Full Constant time**
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity
  - Hard to write/review
  - OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

**Pseudo Constant time**
- Mask program flow dependencies on secret values
- Blocks only currently implemented attacks
- Lower code complexity
Pseudo Vs Fully Constant time

Full Constant time
• Program flow independent from secret values
• Blocks all currently known classes of attacks*
• “Full” is easy to test
• Very high code complexity
  • Hard to write/review
  • OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

Pseudo Constant time
• Mask program flow dependencies on secret values
• Blocks only currently implemented attacks
• Lower code complexity
• “Pseudo” is Hard to test
Pseudo Vs Fully Constant time

Full Constant time
• Program flow independent from secret values
• Blocks all currently known classes of attacks*
• “Full” is easy to test
• Very high code complexity
  • Hard to write/review
  • OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

Pseudo Constant time
• Mask program flow dependencies on secret values
• Blocks only currently implemented attacks
• Lower code complexity
• “Pseudo” is Hard to test
  • Lucky 13 Strikes back [IIES 2015]
Pseudo Vs Fully Constant time

**Full Constant time**
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity
  - Hard to write/review
  - OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

**Pseudo Constant time**
- Mask program flow dependencies on secret values
- Blocks only currently implemented attacks
- Lower code complexity
- “Pseudo” is Hard to test
  - Lucky 13 Strikes back [IIES 2015]
  - Lucky Microseconds [AP 2016]
Pseudo Vs Fully Constant time

Full Constant time
- Program flow independent from secret values
- Blocks all currently known classes of attacks*
- “Full” is easy to test
- Very high code complexity
  - Hard to write/review
  - OpenSSL AES-NI CBC_HMAC vulnerability (2013-2016)

Pseudo Constant time
- Mask program flow dependencies on secret values
- Blocks only currently implemented attacks
- Lower code complexity
- “Pseudo” is Hard to test
  - Lucky 13 Strikes back [IIES 2015]
  - Lucky Microseconds [AP 2016]
  - ???
Our Findings
Our Findings

``All secure implementations are alike; each insecure implementation is buggy in its own way.'' -- after Leo Tolstoy, *Anna Karenina*
Our Findings

``All secure implementations are alike; each insecure implementation is buggy in its own way.'' -- after Leo Tolstoy, *Anna Karenina*

- All **fully** constant time implementations of HMAC-then-CBC are **secure***
Our Findings

``All secure implementations are alike; each insecure implementation is buggy in its own way.'' -- after Leo Tolstoy, *Anna Karenina*

- All **fully** constant time implementations of HMAC-then-CBC are **secure**
- All **pseudo** constant time implementations are **vulnerable**
Our Findings

``All secure implementations are alike; each insecure implementation is buggy in its own way.'' -- after Leo Tolstoy, *Anna Karenina*

- All **fully** constant time implementations of HMAC-then-CBC are **secure***
- All **pseudo** constant time implementations are **vulnerable**
  - Amazon’s S2N, mbed TLS, GnuTLS, wolfSSL
Our Findings

``All secure implementations are alike; each insecure implementation is buggy in its own way." -- after Leo Tolstoy, *Anna Karenina*

• All **fully** constant time implementations of HMAC-then-CBC are **secure***
• All **pseudo** constant time implementations are **vulnerable**
  • Amazon’s S2N, mbed TLS, GnuTLS, wolfSSL
• All countermeasures were **buggy**
Our Findings

``All secure implementations are alike; each insecure implementation is buggy in its own way." -- after Leo Tolstoy, Anna Karenina

- All **fully** constant time implementations of HMAC-then-CBC are **secure**
- All **pseudo** constant time implementations are **vulnerable**
  - Amazon’s S2N, mbed TLS, GnuTLS, wolfSSL
  - All countermeasures were **buggy**
  - All implementations were vulnerable to different novel variants of cache attacks
CBC_HMAC – Lucky 13 Attack

MAC - HMAC-MD5, HMAC-SHA1, HMAC-SHA256

Decryption - CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Padding - “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Lucky 13 Attack

MAC: HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Decrypt: CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
Padding: “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC Padding oracles [Vaudenay 2002]
CBC Padding oracles [Vaudenay 2002]
CBC Padding oracles [Vaudenay 2002]

$P = \text{Dec}_K(C)$

Padding Oracle

$C$
CBC Padding oracles [Vaudenay 2002]
CBC Padding oracles [Vaudenay 2002]

- Padding Oracle
- $P = \text{Dec}_K(C)$
- Check padding of $P$
- $C$
- Valid/Invalid
CBC Padding oracles [Vaudenay 2002]

- In CBC mode, **Padding Oracles** can be used to build a **Decryption Oracle**.
CBC_HMAC – Timing Padding Oracle

SQN || HDR

Payload fragment

MAC

Payload fragment

MAC tag

Padding

Decrypt

HDR

Ciphertext

MAC

HMAC-MD5, HMAC-SHA1, HMAC-SHA256

Decrypt

CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Padding

“00” or “01 01” or “02 02 02” or … or “FF FF….FF”
CBC_HMAC – Invalid Padding

- SQN || HDR
- Payload fragment
- MAC
- Padding
- MAC tag
- Decrypt
- HDR
- Ciphertext

- MAC: HMAC-MD5, HMAC-SHA1, HMAC-SHA256
- Decrypt: CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
- Padding: “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Invalid Padding

- **SQN || HDR**
- **Payload fragment**
- **MAC**
- **MAC tag**
- **Padding**
- **Decrypt**
- **HDR**
- **Ciphertext**

**MAC**
- HMAC-MD5, HMAC-SHA1, HMAC-SHA256

**Decrypt**
- CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

**Padding**
- “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Invalid Padding

Invalid

**MAC**
- HMAC-MD5, HMAC-SHA1, HMAC-SHA256

**Decrypt**
- CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
  - “00” or “01 01” or “02 02 02” or … or “FF FF….FF”

**Padding**

![Diagram showing invalid padding in CBC_HMAC with elements labeled: SQN || HDR, Payload fragment, MAC, Payload fragment, MAC tag, Padding, Decrypt, HDR, Ciphertext.](image)
CBC_HMAC – Long Valid Padding

- **SQN || HDR**
- **Payload fragment**
- **MAC**

- **Payload fragment**
- **MAC tag**
- **Padding**

- **Decrypt**

**MAC**
- HMAC-MD5, HMAC-SHA1, HMAC-SHA256

**Decrypt**
- CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

**Padding**
- “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Short Valid Padding

- **MAC**: HMAC-MD5, HMAC-SHA1, HMAC-SHA256
- **Decrypt**: CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
- **Padding**: “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
Padding Oracle to Plaintext Recovery
Padding Oracle to Plaintext Recovery

- Needs multiple oracle queries
Padding Oracle to Plaintext Recovery

• Needs **multiple** oracle queries
  • TLS handshakes’ keys are **dropped** after any error
Padding Oracle to Plaintext Recovery

• Needs **multiple** oracle queries
  • TLS handshakes’ keys are **dropped** after any error
  • Can only recover data that is **fixed** between TLS handshakes
Padding Oracle to Plaintext Recovery

• Needs multiple oracle queries
  • TLS handshakes’ keys are dropped after any error
  • Can only recover data that is fixed between TLS handshakes

• BEAST like attack on session cookies
Padding Oracle to Plaintext Recovery

• Needs **multiple** oracle queries
  • TLS handshakes’ keys are **dropped** after any error
  • Can only recover data that is **fixed** between TLS handshakes

• **BEAST like attack on** session cookies
  • Use JavaScript in browser to **repeatedly** reopen connections
Padding Oracle to Plaintext Recovery

• Needs **multiple** oracle queries
  • TLS handshakes’ keys are **dropped** after any error
  • Can only recover data that is **fixed** between TLS handshakes

• BEAST like attack on **session cookies**
  • Use JavaScript in browser to **repeatedly** reopen connections
  • At the start of each connection, the same **session cookie** is sent in the first packet
Padding Oracle to Plaintext Recovery

- Needs **multiple** oracle queries
  - TLS handshakes’ keys are **dropped** after any error
  - Can only recover data that is **fixed** between TLS handshakes

- BEAST like attack on **session cookies**
  - Use JavaScript in browser to **repeatedly** reopen connections
  - At the start of each connection, the same **session cookie** is sent in the first packet
  - From the JavaScript we can **control the offset** of the session cookie in the packet
Attack Scenario:
MiTM + Cache timing side channel
Attack Scenario:
MiTM + Cache timing side channel
Attack Scenario:
MiTM + Cache timing side channel
Attack Scenario: MiTM + Cache timing side channel
Attack Scenario:
MiTM + Cache timing side channel
Attack Scenario:
MiTM + Cache timing side channel
Attack Scenario:
MiTM + Cache timing side channel
Attack Scenario:
MiTM + Cache timing side channel
From Timing to Cache based Oracle
From Timing to Cache based Oracle

• Prior to our attack there was no known attacks against the fully patched pseudo constant time implementations
From Timing to Cache based Oracle

• Prior to our attack there was no known attacks against the fully patched pseudo constant time implementations
  • The timing is pseudo constant
From Timing to Cache based Oracle

• Prior to our attack there was no known attacks against the fully patched pseudo constant time implementations
  • The timing is pseudo constant
  • The overall memory access pattern is constant
From Timing to Cache based Oracle

• Prior to our attack there was no known attacks against the fully patched pseudo constant time implementations
  • The timing is pseudo constant
  • The overall memory access pattern is constant

• Our main observation
From Timing to Cache based Oracle

• Prior to our attack there was no known attacks against the fully patched pseudo constant time implementations
  • The timing is pseudo constant
  • The overall memory access pattern is constant

• Our main observation
  • The temporal memory access pattern is not constant
From Timing to Cache based Oracle

• Prior to our attack there was no known attacks against the fully patched pseudo constant time implementations
  • The timing is pseudo constant
  • The overall memory access pattern is constant

• Our main observation
  • The temporal memory access pattern is not constant
  • Using new variants of the PRIME+PROBE cache attack we were able to recreate the padding oracle
CBC_HMAC – Memory Access Long Pad

SQN || HDR  Payload fragment

MAC

Payload fragment  MAC tag  Padding

Decrypt

HDR  Ciphertext

MAC

HMAC-MD5, HMAC-SHA1, HMAC-SHA256

Encrypt

CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Padding

“00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Memory Access Long Pad

- **SQN || HDR**
- **Payload fragment**
- **MAC**
- **MAC tag**
- **Padding**
- **Decrypt**

Memory Accessed while decrypting

**MAC**: HMAC-MD5, HMAC-SHA1, HMAC-SHA256

**Encrypt**: CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

**Padding**: “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Memory Access Long Pad

- Memory Accessed while verifying

- HMAC-MD5, HMAC-SHA1, HMAC-SHA256
- CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
- “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Memory Access Short Pad

SQN || HDR

Payload fragment

MAC

Payload fragment

MAC tag

P

Decrypt

HDR

Ciphertext

MAC: HMAC-MD5, HMAC-SHA1, HMAC-SHA256

Encrypt: CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Padding: “00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Memory Access Short Pad

SQN || HDR | Payload fragment

MAC

Payload fragment | MAC tag | P

Decrypt

HDR | Ciphertext

Memory Accessed while decrypting

MAC

HMAC-MD5, HMAC-SHA1, HMAC-SHA256

Encrypt

CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Padding

“00” or “01 01” or “02 02 02” or …. or “FF FF….FF”
CBC_HMAC – Memory Access Short Pad

SQN || HDR | Payload fragment | MAC | Payload fragment | MAC tag | P

Decrypt

HDR | Ciphertext

Memory Accessed while verifying

MAC

HMAC-MD5, HMAC-SHA1, HMAC-SHA256

Encrypt

CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Padding

“00” or “01 01” or “02 02 02” or … or “FF FF….FF”
Our results
Our results

• Exploiting the *different temporal memory access* patterns we can recreate a *Lucky 13 attack variant*
Our results

• Exploiting the different temporal memory access patterns we can recreate a Lucky 13 attack variant
• PoC for 3 plaintext recovery attack variants
Our results

• Exploiting the different temporal memory access patterns we can recreate a Lucky 13 attack variant
• PoC for 3 plaintext recovery attack variants
  • Synchronized probe PRIME+PROBE on Amazon’s s2n
Our results

• Exploiting the **different temporal memory access** patterns we can recreate a **Lucky 13 attack variant**
• PoC for 3 plaintext recovery attack variants
  • Synchronized **probe** PRIME+PROBE on Amazon’s **s2n**
  • Synchronized **prime** PRIME+PROBE on wolfSSL, mbed TLS and GnuTLS
Our results

• Exploiting the different temporal memory access patterns we can recreate a Lucky 13 attack variant
• PoC for 3 plaintext recovery attack variants
  • Synchronized probe PRIME+PROBE on Amazon’s s2n
  • Synchronized prime PRIME+PROBE on wolfSSL, mbed TLS and GnuTLS
  • “PostFetch” cache attack on mbed TLS
Our results

• Exploiting the **different temporal memory access** patterns we can recreate a **Lucky 13 attack variant**
• PoC for 3 plaintext recovery attack variants
  • Synchronized **probe** PRIME+PROBE on Amazon’s s2n
  • Synchronized **prime** PRIME+PROBE on wolfSSL, mbed TLS and GnuTLS
  • “**PostFetch**” cache attack on mbed TLS
• Greedy Algorithm to **optimize** plaintext recovery
CBC_HMAC with SHA-384 Bugs
CBC_HMAC with SHA-384 Bugs

• Most widely used CBC_HMAC cipher suite
CBC_HMAC with SHA-384 Bugs

• Most widely used CBC_HMAC cipher suite
• All pseudo constant time countermeasures were vulnerable
CBC_HMAC with SHA-384 Bugs

• Most widely used CBC_HMAC cipher suite
• All *pseudo constant time* countermeasures were *vulnerable*
  • Dummy operation calculation wrongly based on SHA-1/256 specific hardcoded values
CBC_HMAC with SHA-384 Bugs

- **Most widely** used CBC_HMAC cipher suite
- **All pseudo constant time** countermeasures were **vulnerable**
  - Dummy operation calculation wrongly based on SHA-1/256 specific hardcoded values
  - Some implementations didn’t even protect SHA-1/256
CBC_HMAC with SHA-384 Bugs

• Most widely used CBC_HMAC cipher suite
• All pseudo constant time countermeasures were vulnerable
  • Dummy operation calculation wrongly based on SHA-1/256 specific hardcoded values
  • Some implementations didn’t even protect SHA-1/256
• Hard to test correctness of the pseudo constant time countermeasure
CBC_HMAC with SHA-384 Bugs

• Most widely used CBC_HMAC cipher suite
• All pseudo constant time countermeasures were vulnerable
  • Dummy operation calculation wrongly based on SHA-1/256 specific hardcoded values
  • Some implementations didn’t even protect SHA-1/256
• Hard to test correctness of the pseudo constant time countermeasure
  • All constant time countermeasures were secure
Disclosure
Disclosure

- wolfSSL switched to **full constant time** (release 3.15.4)
Disclosure

• wolfSSL switched to **full constant time** (release 3.15.4)
• mbed TLS released security advisory with CVEs 2018-0497 and 2018-0497 that were marked as “**high severity**”
  • Users urged to update to new version with interim fix
  • **Full constant time** solution is planned
Disclosure

• wolfSSL switched to full constant time (release 3.15.4)
• mbed TLS released security advisory with CVEs 2018-0497 and 2018-0497 that were marked as “high severity”
  • Users urged to update to new version with interim fix
  • Full constant time solution is planned
• Amazon s2n plans to disable CBC_HMAC by default and switch to the BoringSSL full constant time implementation
Disclosure

• wolfSSL switched to full constant time (release 3.15.4)
• mbed TLS released security advisory with CVEs 2018-0497 and 2018-0497 that were marked as “high severity”
  • Users urged to update to new version with interim fix
  • Full constant time solution is planned
• Amazon s2n plans to disable CBC_HMAC by default and switch to the BoringSSL full constant time implementation
• GnuTLS made several changes to address the bugs
  • We believe that the code is still vulnerable to variants of the attack
“PostFetch” Cache Attack
“PostFetch” Cache Attack

• We want to know what part of a short array was read
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
• Continuous reading near the end of the cache line will cause the next cache line to be prefetched
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
• Continuous reading near the end of the cache line will cause the next cache line to be prefetched
• Target our cache attack on the cache line storing the bytes after the array
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
• Continuous reading near the end of the cache line will cause the next cache line to be prefetched
• Target our cache attack on the cache line storing the bytes after the array

Cache Line 1  
Cache Line 2
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
• Continuous reading near the end of the cache line will cause the next cache line to be prefetched
• Target our cache attack on the cache line storing the bytes after the array
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
• Continuous reading near the end of the cache line will cause the next cache line to be prefetched
• Target our cache attack on the cache line storing the bytes after the array
“PostFetch” Cache Attack

• We want to know what part of a short array was read
• Differentiate between long and short access patterns inside a single cache line
• Continuous reading near the end of the cache line will cause the next cache line to be prefetched
• Target our cache attack on the cache line storing the bytes after the array
Synchronized probe PRIME+PROBE
Synchronized probe PRIME+PROBE

• We want to measure the time difference
Synchronized probe PRIME+PROBE

• We want to measure the time difference
  • E.g. between sending a message at $t_{send}$ and a memory access by the target at either $t_{send} + t_1$ or $t_{send} + t_2$
Synchronized probe PRIME+PROBE

- We want to measure the time difference
  - E.g. between sending a message at $t_{\text{send}}$ and a memory access by the target at either $t_{\text{send}} + t_1$ or $t_{\text{send}} + t_2$
  - We choose $t_{\text{probe}}$ such that $t_1 < t_{\text{probe}} < t_2$
Synchronized probe PRIME+PROBE

- We want to measure the time difference
- E.g. between sending a message at $t_{\text{send}}$ and a memory access by the target at either $t_{\text{send}} + t_1$ or $t_{\text{send}} + t_2$
- We choose $t_{\text{probe}}$ such that $t_1 < t_{\text{probe}} < t_2$
- We prime the memory before sending the message, and probe at $t_{\text{send}} + t_{\text{probe}}$
Synchronized probe PRIME+PROBE

• We want to measure the time difference
  • E.g. between sending a message at $t_{\text{send}}$ and a memory access by the target at either $t_{\text{send}} + t_1$ or $t_{\text{send}} + t_2$
  • We choose $t_{\text{probe}}$ such that $t_1 < t_{\text{probe}} < t_2$
  • We prime the memory before sending the message, and probe at $t_{\text{send}} + t_{\text{probe}}$
• We also use synchronized prime PRIME+PROBE
Conclusion

- All *pseudo* constant time implementations we reviewed
  - were *buggy* and still vulnerable to the original Lucky 13 attack.
  - were *vulnerable* to one or more of our 3 *novel cache attacks*
Conclusion

• All pseudo constant time implementations we reviewed
  • were buggy and still vulnerable to the original Lucky 13 attack.
  • were vulnerable to one or more of our 3 novel cache attacks
• Writing fully constant time code is hard but it is worth the effort
Conclusion

• All pseudo constant time implementations we reviewed
  • were buggy and still vulnerable to the original Lucky 13 attack.
  • were vulnerable to one or more of our 3 novel cache attacks
• Writing fully constant time code is hard but it is worth the effort

• Any questions?