Securely Implementing Network Protocols: Detecting and Preventing Logical Flaws

Mathy Vanhoef (KU Leuven)
Black Hat Webcast, 24 August 2017

@vanhoefm
Many protocols have been affected by logical bugs

<table>
<thead>
<tr>
<th>Design flaws</th>
<th>Implementation flaws</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLS</strong></td>
<td></td>
</tr>
<tr>
<td>BEAST\textsuperscript{11}</td>
<td>Early CCS attack\textsuperscript{5}</td>
</tr>
<tr>
<td>POODLE\textsuperscript{12}</td>
<td>FREAK\textsuperscript{8}</td>
</tr>
<tr>
<td>Lucky 13\textsuperscript{13}</td>
<td>Logjam\textsuperscript{10}</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td><strong>Wi-Fi</strong></td>
<td></td>
</tr>
<tr>
<td>WEP</td>
<td>Bad state machine\textsuperscript{4}</td>
</tr>
<tr>
<td>Protected setup\textsuperscript{7}</td>
<td>No downgrade check\textsuperscript{4}</td>
</tr>
<tr>
<td>Key reinstallations\textsuperscript{1}</td>
<td>Bad randomness\textsuperscript{6,7}</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td><strong>SSH</strong></td>
<td></td>
</tr>
<tr>
<td>CBC plaintext recovery\textsuperscript{2}</td>
<td>Bad state machine\textsuperscript{3}</td>
</tr>
</tbody>
</table>
Many protocols have been affected by **logical** bugs.

We focus on logical implementation flaws:

<table>
<thead>
<tr>
<th>Implementation flaws</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early CCS attack⁵</td>
</tr>
<tr>
<td>FREAK⁸</td>
</tr>
<tr>
<td>Logjam¹⁰</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Bad state machine⁴</td>
</tr>
<tr>
<td>No downgrade check⁴</td>
</tr>
<tr>
<td>Bad randomness⁶,⁷</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Bad state machine³</td>
</tr>
</tbody>
</table>
How were TLS flaws detected?

Several works audited state machines:

2014
• Kikuchi discovered the early CCS attack\(^5\)
• Manual inspection of CCS transitions in implementations

2015
• Beurdouche et al: manually define state machine of TLS\(^8\)
• Use state machine to \textit{generate invalid handshakes}

2016
• de Ruiter and Poll: \textit{extract state machine} automatically\(^9\)
• Manually inspect state machine for anomalies
Lesson: use model-based testing!

- Test if program behaves according to some abstract model
- Proved successful against TLS

- We applied model-based approach on the Wi-Fi handshake
- Our technique can be used to test other protocols!
Background: the Wi-Fi handshake

Main purposes:
- Network discovery
- Mutual authentication & negotiation of pairwise session key
- Securely select cipher to encrypt data frames

Short-term solution: reduced security so it could run on old hardware
Long-term solution based on modern cryptographic primitives

WPA-TKIP

AES-CCMP
Wi-Fi handshake (simplified)

Client

Select cipher

Access Point

Beacons: supported ciphers

Association Request: chosen cipher
Wi-Fi handshake (simplified)

Client

Beacons: supported ciphers

Select cipher

Association Request: chosen cipher

Msg1: ANonce

Session keys

Msg2: SNonce + chosen cipher

Session keys

Access Point
Wi-Fi handshake (simplified)

- **Client**
  - Select cipher
  - Session keys
  - verify supported ciphers

- **Access Point**
  - Beacons: supported ciphers
  - Association Request: chosen cipher
  - Msg1: ANonce
  - Msg2: SNonce + chosen cipher
  - Msg3: supported ciphers
  - verify chosen cipher
  - Msg4: ACK
Wi-Fi handshake (simplified)

- Beacons: supported ciphers
  
  - Select cipher
  - Association Request: chosen cipher
    
    - Msg1: ANonce
    - Msg2: SNonce + supported ciphers
    - Msg3: supported ciphers
    - Msg4: ACK

- Defined using EAPOL frames
**EAPOL frame layout**

- **802.1X Authentication**
  - Version: 802.1X-2004 (2)
  - Type: Key (3)
  - Length: 117
  - Key Descriptor Type: EAPOL RSN Key (2)
  - Key Information: 0x008a
  - Key Length: 16
  - Replay Counter: 0
  - WPA Key Nonce: 3e8e967dacd960324cac5b6aa721235bf57b949771c86798...
  - Key IV: 00000000000000000000000000000000
  - WPA Key RSC: 0000000000000000
  - WPA Key ID: 0000000000000000
  - WPA Key MIC: 00000000000000000000000000000000
  - WPA Key Data Length: 22
  - WPA Key Data: dd14000fac04592da88096c461da246c69001e877f3d
EAPOL frame layout

- Header
- Replay counter
- …
- MIC
- Key data

encrypted

≈
Test generation rules:

- Test various edge cases, allows some creativity
- Are assumed to be independent (avoid state explosion)

A test case defines:
1. Messages to send & expected replies
2. Results in successful connection?
Executing test cases

For every test case

- Execute test case
- Check if connection successful
  - unexpected result
    - unexpected reply
      - Save failed test
        - All OK
          - Reset

Afterwards inspect **failed** test cases

- Experts determines impact and exploitability
Test generation rules

Test generation rules manipulating messages as a whole:
1. Drop a message
2. Inject/repeat a message

Test generation rules that modify fields in messages:
1. Bad EAPOL replay counter
2. Bad EAPOL header (e.g. message ID)
3. Bad EAPOL Message Integrity Check (MIC)
4. Mismatch in selected cipher suite
5. …
Evaluation

We tested 12 access points:

- Open source: OpenBSD, Linux’s Hostapd
- Leaked source: Broadcom, MediaTek (home routers)
- Closed source: Windows, Apple, …
- Professional equipment: Aerohive, Aironet

Discovered several issues!
Missing downgrade checks

1. MediaTek & Telenet don’t verify selected cipher in message 2
2. MediaTek also ignores supported ciphers in message 3

→ Trivial downgrade attack against MediaTek clients
Windows 7 targeted DoS

Client

Association Request
Association Request
Association Request
Association Rejected

AP

Association Request
Msg1
...

Client 2
Windows 7 targeted DoS

Client

AP

Client 2

PoC @

github.com/vanhoefm/blackhat17-pocs
Broadcom downgrade

Broadcom cannot distinguish message 2 and 4
- Can be abused to downgrade the AP to TKIP

Hence message 4 is essential in preventing downgrade attacks
- This highlights incorrect claims in the 802.11 standard:

“While Message 4 serves no cryptographic purpose, it serves as an acknowledgment to Message 3. It is required to ensure reliability and to inform the Authenticator that the Supplicant has installed the PTK and GTK and hence can receive encrypted frames.”
OpenBSD: client man-in-the-middle

Bug in state machine of AP → we also inspected client: State machine missing!

→ Man-in-the-middle against client
OpenBSD: client man-in-the-middle

Victim (client) <-> Adversary (Rogue AP)

Beacons with network info

Select network

Association Request
OpenBSD: client man-in-the-middle

Victim (client)  Adversary (Rogue AP)

Beacons with network info

Select network

Association Request

EAPOL-Key(\text{Group1}, \text{MIC}; \text{Encrypted}\{\text{GTK}\})

Verify with all-zero PTK
OpenBSD: client man-in-the-middle

Victim (client) -> Beacons with network info -> Adversary (Rogue AP)

Select network -> Association Request

EAPOL-Key(Group1, MIC; Encrypted{GTK})

Verify with all-zero PTK

EAPOL-Key(Group2, MIC)

Open 802.1x port
OpenBSD: client man-in-the-middle

Victim (client) → Adversary (Rogue AP)

Beacons with network info

Select network

Association Request

EAPOL-Key(Group1, MIC; Encrypted{GTK})

Verify with all-zero PTK

EAPOL-Key(Group2, MIC)

Open 802.1x port

← --- Victim sends and accepts plaintext data frames --- →
OpenBSD: client man-in-the-middle

PoC @

github.com/vanhoefm/blackhat17-pocs

--- Victim sends and accepts plaintext data frames ---

--- Select network ---
--- Associate client ---
--- EAPOL-Key (Group, MIC, encrypted {GTK}) ---
--- Verify with all-zero PTK ---

Victim (client)

Adversary (Rogue AP)

Beacons with network info
More results

See Black Hat & AsiaCCS paper:\n
- Benign irregularities → fingerprint
- Permanent DoS attack against Broadcom and OpenBSD
- DoS attack against Windows 10, Broadcom, Aerohive
- Inconsistent parsing of supported cipher suite list
- …
Current limitations:

- Amount of code coverage is unknown
- Only used well-formed (albeit invalid) packets
- Test generation rules applied independently

But already a promising technique

✔ Black-box testing mechanism: no source code needed
✔ Fairly simple handshake, but still several logical bugs!
Conclusion: avoiding logical bugs

What helps:
- Try to generalize known bugs (in your/other products)
- Model-based testing (e.g. this presentation)
- Write rigorous requirements (specification) and review them
- Detailed code reviews (e.g. by domain experts)

Does not help:
- Standard code review (only detects common mistakes)
- Traditional static or dynamic testing
Securely Implementing Network Protocols: Detecting and Preventing Logical Flaws

Mathy Vanhoef (KU Leuven)
Black Hat Webcast, 24 August 2017

@vanhoefm