Rooting Routers Using Symbolic Execution

Mathy Vanhoef — @vanhoefm
IT-Defense, Stuttgart, Germany, 7 February 2019
Overview

Symbolic Execution

Handling Crypto

4-way handshake

Results
Overview

Symbolic Execution

Handling Crypto

4-way handshake

Results
Motivating Example

Try to reach the prize(#)!

```
+-----------+
| X      | # |
| +--- +--- +--- |
| | | |
| +-----------+
```

Enter 20 player moves using a sequence of 'w', 's', 'a' or 'd'

Input:
Motivating Example

Try to reach the prize(#)!

Enter 20 player moves using a sequence of 'w', 's', 'a' or 'd'

Input: dddddddssddww
Motivating Example

Try to reach the prize(#)!

```
+-----------+
<table>
<thead>
<tr>
<th>X</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+-----+
```

Enter 20 player moves using a sequence of 'w', 's', 'a' or 'd'

Input: wwww
How to explore all paths?

Try to reach the prize(#)!

Enter 20 player moves using a sequence of 'w', 's', 'a' or 'd'

Input:
How to explore all paths?

Try to reach the prize(#)!

Give as input math variables that can have any value

Input: $x_1 x_2 x_3 \ldots x_{20}$

= symbolic variables
How to explore all paths?

Try to reach the prize(#!)!

Can’t use normal compiler anymore!
Need **symbolic execution engine**

Running our maze: it first checks if all letters are ‘w’, ‘s’, ‘a’ or ‘d’

Input: $x_1x_2x_3 \ldots x_{20}$
How to explore all paths?

There is invalid input

All $x_i$ represent valid input
How to explore all paths?

There is invalid input

All $x_i$ represent valid input

Execution engine tracks conditions on $x_i$’s using a path constraint
How to explore all paths?

There is invalid input

All $x_i$ represent valid input

Game Over
How to explore all paths?

Path constraint: \( x_1 = 's' \) && ...

Path constraint: \( x_1 = 'd' \) && ...
void recv(data, len) {
    if (data[0] != 1) return
    if (data[1] != len) return
    int num = len/data[2]
    ...
Symbolic Execution

<table>
<thead>
<tr>
<th>data[0] != 1</th>
<th>data[0] == 1</th>
</tr>
</thead>
</table>
| void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return
  int num = len/data[2]
  ...
} | void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return
  int num = len/data[2]
  ...
} |
Symbolic Execution

\[
\text{data}[0] \neq 1 \quad \text{or} \quad \text{data}[0] = 1
\]

Continue execution:
\[
\text{if} \ (\text{data}[1] \neq \text{len})
\]

PC = Path Constraint
Symbolic Execution

\[ \text{data}[0] \neq 1 \]

\[ \text{data}[0] = 1 \land \text{data}[1] \neq \text{len} \]

\[ \text{data}[0] = 1 \land \text{data}[1] = \text{len} \]

Continue execution
Symbolic Execution

data[0] == 1 &&
data[1] == len

void recv(data, len) {
    if (data[0] != 1) 
        return
    if (data[1] != len) 
        return
    int num = len/data[2]
    ...

Yes! Bug detected!
Can data[2] equal zero under the current PC?
Implementations

We build upon KLEE
› Works on LLVM bytecode
› Actively maintained

Practical limitations:
› $|paths| = 2^{|if-statements|}$
› Infinite-length paths
› SMT query complexity
Overview

Symbolic Execution

Handling Crypto

4-way handshake

Results
Motivating Example

```c
void recv(data, len) {
    plain = decrypt(data, len)
    if (plain == NULL) return

    if (plain[0] == COMMAND)
        process_command(plain)
    else
        ...
}
```

Mark data as symbolic

Summarize crypto algo. (time consuming)

Analyze crypto algo. (time consuming)

Won’t reach this function!
Decrypted output = fresh symbolic variable
What does this give us?

WHAT? SORRY. I WAS USING THIS TIME TO THINK ABOUT SOMETHING USEFUL.

MAYBE YOUR BOSS CAN FILL YOU IN.

I WAS BRAIN-GOLFING.
What does this give us?

We can now detect misuse of crypto primitives!

Timing side-channels

Decryption oracles
Overview

Symbolic Execution

Handling Crypto

4-way handshake

Results
The 4-way handshake

Used to connect to any protected Wi-Fi network

Mutual authentication

Negotiates fresh PTK: pairwise transient key
4-way handshake (simplified)

optional 802.1x authentication
4-way handshake (simplified)

PTK = Combine(shared secret, ANonce, SNonce)
4-way handshake (simplified)

optional 802.1x authentication

Msg1(r, ANonce)

Derive PTK

Msg2(r, SNonce)

Derive PTK
4-way handshake (simplified)
4-way handshake (simplified)

optional 802.1x authentication

Msg1(r, ANonce)

Derive PTK

Msg2(r, SNonce)

Derive PTK

Msg3(r+1; GTK)

Install PTK & GTK

Msg4(r+1)

Install PTK
4-way handshake (simplified)

- Optional 802.1x authentication
  - Msg1(r, ANonce)
    - Derive PTK
    - Msg2(r, SNonce)
    - Msg3(r+1; GTK)
    - Msg4(r+1)
  - Install PTK & GTK
    - Install PTK
    - Authenticated with a MAC
      - Encrypted data frames can now be exchanged
We focus on the client

Symbolic execution of

Intel’s iwd deamon  wpa_supplicant  kernel driver

How to get these working under KLEE?
Intel’s iwd

Avoid running full program under KLEE
› Would need to model Wi-Fi stack symbolically

Our approach
› iwd contains unit test for the 4-way handshake
› Reuse initialization code of unit test!
› Symbolically execute only receive function
wpa_supplicant

Unit test uses virtual Wi-Fi interface
› Would again need to simulate Wi-Fi stack…

Alternative approach:
› Write unit test that isolates 4-way handshake like iwd
› Then symbolically execute receive function!
› Need to modify code of wpa_supplicant (non-trivial)
MediaTek’s Driver

No unit tests & it’s a Linux driver
› Symbolically executing the Linux kernel?!

Inspired by previous cases
› Write unit test & simulate used kernel functions in userspace
› Verify that code is correctly simulated in userspace
› Again symbolically execute receive function!
Not all our unit tests have clean code

https://github.com/vanhoefm/woot2018
Overview

Symbolic Execution

Handling Crypto

4-way handshake

Results
Discovered Bugs I

Timing side-channels
› Authenticity tag not checked in constant time
› MediaTek and iwd are vulnerable

Denial-of-service in iwd
› Caused by integer underflow
› Leads to huge malloc that fails
Discovered Bugs II

Buffer overflow in MediaTek kernel module
› Occurs when copying the group key
› Remote code execution (details follow)

Flawed AES unwrap crypto primitive
› Also in MediaTek’s kernel driver
› Manually discovered
Decryption oracle in wpa_supplicant

Decryption oracle:
› Authenticity of Msg3 not checked
› But decrypts and processes data

→ Decrypt group key in Msg3 (details follow)
Rooting Routers:
Buffer overflow in MediaTek kernel module
MediaTek buffer overflow preconditions I

Triggered when the **client** processes Msg3

- Adversary needs password of network
- Examples: Wi-Fi at conferences, hotels, etc.
MediaTek buffer overflow preconditions II

Which clients use the MediaTek driver?
› Not part of Linux kernel source tree
› Used in repeater modes of routers

Our target:
› RT-AC51U running Padavan firmware
› Original firmware has no WPA2 repeater
Popularity of Padavan firmware

<table>
<thead>
<tr>
<th>Download repository</th>
<th>Size</th>
<th>Username</th>
<th>Downloads</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT-AC54U_3.4.3.9-099_base.trx</td>
<td>7.0 MB</td>
<td>padavan</td>
<td>37142</td>
<td>2016-03-05</td>
</tr>
<tr>
<td>RT-AC51U_3.4.3.9-099_full.trx</td>
<td>9.6 MB</td>
<td>padavan</td>
<td>51270</td>
<td>2016-03-05</td>
</tr>
<tr>
<td>RT-AC51U_3.4.3.9-099_base.trx</td>
<td>7.0 MB</td>
<td>padavan</td>
<td>5280</td>
<td>2016-03-05</td>
</tr>
<tr>
<td>RT-N11P_3.4.3.9-099_nano.trx</td>
<td>2.9 MB</td>
<td>padavan</td>
<td>5134</td>
<td>2016-03-05</td>
</tr>
<tr>
<td>RT-N11P_3.4.3.9-099_base.trx</td>
<td>4.1 MB</td>
<td>padavan</td>
<td>8045</td>
<td>2016-03-05</td>
</tr>
<tr>
<td>RT-N14U_3.4.3.9-099_full.trx</td>
<td>9.2 MB</td>
<td>padavan</td>
<td>13856</td>
<td>2016-03-05</td>
</tr>
</tbody>
</table>

We exploit this version
void RMTPParseEapolKeyData(pKeyData, KeyDataLen, MsgType) {
    UCHAR GTK[MAX_LEN_GTK];

    if (MsgType == PAIR_MSG3 || MsgType == GROUP_MSG_1) {
        PKDE_HDR *pKDE = find_tlv(pKeyData, KeyDataLen, WPA2GTK);
        GTK_KDE *pKdeGtk = (GTK_KDE*)pKDE->octet;
        UCHAR GTKLEN = pKDE->Len - 6;
        NdisMoveMemory(GTK, pKdeGtk->GTK, GTKLEN);
    }

    APCliInstallSharedKey(GTK, GTKLEN);
}
The vulnerable code (simplified)

```c
void RMTPParseEapolKeyData(pKeyData, KeyDataLen, MsgType) {
    UCHAR GTK[MAX_LEN_GTK];

    if (MsgType == PAIR_MSG3 || MsgType == GROUP_MSG_1) {
        PKDE_HDR *pKDE = find_tlv(pKeyData, KeyDataLen, WPA2GTK);
        GTK_KDE *pKdeGtk = (GTK_KDE*)pKDE->octet;
        UCHAR GTKLEN = pKDE->Len - 6;
        NdisMoveMemory(GTK, pKdeGtk->GTK, GTKLEN);
    }

    APCliInstallSharedKey(GTK, GTKLEN);
}
```

Len controlled by attacker

Destination buffer 32 bytes
Main exploitation steps

- Code execution in kernel
- Obtain a process context
- Inject shellcode in process
- Run injected shellcode
Main exploitation steps

• Code execution in kernel
• Obtain a process context
• Inject shellcode in process
• Run injected shellcode
Gaining kernel code execution

How to control return address & where to return?
› Kernel *doesn’t use stack canaries*
› Kernel stack has *no address randomization*
› And the kernel stack is *executable*

Return to shellcode on stack & done?
    Nope... our shellcode crashes
Problem: cache incoherency on MIPS

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>old stack data</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>old stack data</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Problem: cache incoherency on MIPS

Memory

...  
old stack data
...

Fetch

Data cache

...  
shellcode
...

Instruction cache

...  
<no cached entry>
...
Problem: cache incoherency on MIPS

Memory
...
old stack data
...

Data cache
...
shellcode
...

Instruction cache
...
old stack data
...

Fetch
Solution: flush cache after write

Flush

Memory

old stack data

Data cache

shellcode

Instruction cache

<no cached entry>
Solution: flush cache after write

Memory

...  

shellcode

...

Data cache

...

shellcode

...

Flush

Instruction cache

...

<no cached entry>

...

Fetch
Solution: flush cache after write
How to flush the cache?

Execute kernel function to flush cache

› Rely on Return Oriented Programming (ROP)
› Use mipsrop tool of Craig Heffner

MIPS ROP Finder activated, found 1292 controllable jumps between 0x00000000 and 0x00078FE8
Python>mipsrop.tails()

<table>
<thead>
<tr>
<th>Address</th>
<th>Action</th>
<th>Control Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0005E99C</td>
<td>move $t9,$a2</td>
<td>jr $a2</td>
</tr>
<tr>
<td>0x00061858</td>
<td>move $t9,$a2</td>
<td>jr $a2</td>
</tr>
<tr>
<td>0x00062D68</td>
<td>move $t9,$a2</td>
<td>jr $a2</td>
</tr>
</tbody>
</table>

Found 3 matching gadgets

→ Building ROP chain is tedious but doable
Main exploitation steps

- Code execution in kernel
- *Obtain a process context*
- Inject shellcode in process
- Run injected shellcode
Obtaining a process context

Code execution in kernel, let’s spawn a shell?
› Tricky when in interrupt context
› Easier in process context: access to address space

How to obtain a process context?
› System calls run in process context …
› … so intercept a close() system call
## Intercepting system calls

### System call table:

<table>
<thead>
<tr>
<th>System Call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_open</td>
<td></td>
</tr>
<tr>
<td>sys_read</td>
<td></td>
</tr>
<tr>
<td>sys_close</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- **sys_close**
- **normal code**
Intercepting system calls

System call table:
- `sys_open`
- `sys_read`
- `sys_close`
- ...

Interceptor

Attackers code
Jump to `sys_close`

Normal code

`sys_close`
Main exploitation steps

• Code execution in kernel
• Obtain a process context
• Inject shellcode in process
• Run injected shellcode
Hijacking a process

Kernel now executes in process context
› Hijack unimportant detect_link process
› Recognize by its predictable PID

Now easy to inject shellcode in process:
1. Call \texttt{mprotect} to mark process code writable
2. \textbf{Copy user space shellcode} to return address
3. Flush caches
Main exploitation steps

- Code execution in kernel
- Obtain a process context
- Inject shellcode in process
- Run injected shellcode
User space shellcode

When close() returns, shellcode is triggered
› It runs “\texttt{telnetd -p 1337 -l /bin/sh}” using execve
› Adversary can now connect to router

Important remarks:
› Original process is killed, but causes no problems
› Used telnetd to keep shellcode small
Running the full exploit

Multi-chain exploit. Space for shellcode?

› For initial stage we have 250 bytes
› Handshake frame can transport ~2048 bytes
› We can even use null bytes!
Decryption
Oracle
Recall: decryption oracle in wpa_supplicant

Decryption oracle:
› Authenticity of Msg3 not checked
› Does **decrypt and process data**

How can this be abused to leak data?
On reception of Msg3 the receiver:
1. Decrypts the Key Data field
Background: process ordinary Msg3

On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements
Background: process ordinary Msg3

On reception of Msg3 the receiver:
1. Decrypts the Key Data field
2. Parses the type-length-values elements
On reception of Msg3 the receiver:

1. Decrypteds the Key Data field
2. Parses the type-length-values elements
On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements
On reception of Msg3 the receiver:
1. Decrypts the Key Data field
2. Parses the type-length-values elements
Background: process ordinary Msg3

On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements
3. Extracts and installs the group key (GTK)
How to turn parsing into an oracle?
Constructing an oracle

<table>
<thead>
<tr>
<th>header</th>
<th>221</th>
<th>38</th>
<th>$x_0 \ldots x_{37}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Len</td>
<td>GTK</td>
<td></td>
</tr>
</tbody>
</table>

Adversary knows type and length, but not GTK
Constructing an oracle

Adversary knows type and length, but not GTK.

1. Reduce length by two
Constructing an oracle

Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing
Constructing an oracle

Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing
Constructing an oracle

<table>
<thead>
<tr>
<th>header</th>
<th>221</th>
<th>36</th>
<th>$x_0 \ldots x_{35}$</th>
<th>$x_{36}$</th>
<th>$x_{37}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
<td>Len</td>
<td>GTK'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing
Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing only succeeds if \( x_{37} \) equals zero
Constructing an oracle

<table>
<thead>
<tr>
<th>header</th>
<th>221</th>
<th>36</th>
<th>$x_0 \ldots x_{35}$</th>
<th>$x_{36}$</th>
<th>$x_{37}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Len</td>
<td>GTK’</td>
<td>Type</td>
<td>Len</td>
<td></td>
</tr>
</tbody>
</table>

Adversary knows type and length, but not GTK.

1. Reduce length by two
2. **Parsing only succeeds if $x_{37}$ equals zero**
3. Keep flipping encrypted $x_{37}$ until parsing succeeds
Abusing the oracle in practice

1. Guess the last byte (in our example $x_{37}$)
2. XOR the ciphertext with the guessed value
3. **Correct guess**: decryption of $x_{37}$ is zero
   » Parsing succeeds & we get a reply
4. Wrong guess: decryption of $x_{37}$ is non-zero
   » Parsing fails, no reply

→ Keep guessing last byte until parsing succeeds
Practical aspects

Test against Debian 8 client:
› Adversary can guess a value every 14 seconds
› Decrypting 16-byte group key takes ~8 hours

Attack can be made faster by:
› Attacking several clients simultaneously
› Can brute-force the last 4 bytes
Conclusion

› Symbolic execution of protocols
› Simple simulation of crypto
› Root exploit & decryption oracle
› Interesting future work
Thank you!
Questions?
Backup slides
Example

```c
void recv(data, len) {
    plain = decrypt(data, len)
    if (plain == NULL) return
    if (plain[0] == COMMAND) {
        process_command(plain)
    } else {
        ...
    }
}
```

Mark data as symbolic

Create fresh symbolic variable

Can now analyze code that parses decrypted data
Other than handling decryption

Handling hash functions

› Output = fresh symbolic variable
› Also works for HMACs (Message Authentication Codes)

Tracking use of crypto primitives?

› Record relationship between input & output
› = Treat fresh variable as information flow taint
Detecting Crypto Misuse

**Timing side-channels**
› $\forall (paths)$: all bytes of MAC in path constraint?
› If not: comparison exits on first byte difference

**Decryption oracles**
› Behavior depends on unauth. decrypted data
› Decrypt data is in path constraint, but not in MAC
Exploit recap & lessons learned

Cache incoherence

Debug with infinite loops

First test ideas in C

io.netgarage.org
Although limitations remain, symbolic execution tools are now more usable & efficient.