Don’t be Silly
It’s Only a Lightbulb

@EyalItkin
Who Am I

❖ Eyal Itkin
❖ Vulnerability Researcher
❖ cp<rw Check Point Research
❖ Focusing on embedded devices & network protocols
❖ @EyalItkin
Motivation

- 2016 – I heard of a new gadget: smart lightbulbs
- I only need the light when I’m physically in the room
  - Why would I need an app to toggle the switch in the entrance?
- 2018 – More than 400,000 UK households use smart lightbulbs
- OK, there is surely some security problem in it
  - The response: Don’t be silly, it’s only a lightbulb, it’s fine
Prior Work

- A Lightbulb Worm? (@colinoflynn & @eyalr0) – BH USA 2016
- Iot Goes Nuclear: Creating a Zigbee Chain Reaction
  - Authors: Eyal Ronen, Colin O’Flynn, Adi Shamir and Achi-Or Weingarten
  - Website: https://eyalro.net/project/iotworm.html
  - War-flying demo: https://youtu.be/Ed1OjAuRARU
- With @eyalr0’s help, going to continue their research
So, what did they find?

- Attackers can remotely “steal” a lightbulb from a given ZigBee network, and force it to join their own network
- Attackers that share the same ZigBee network with a target lightbulb can send a malicious firmware update to it
- Even a regular lightbulb can be used to “steal” other lightbulbs
So, what did they find?

- Attackers can remotely “steal” a lightbulb from a given ZigBee network, and force it to join their own network.
- Attackers that share the same ZigBee network with a target lightbulb can send a malicious firmware update to it.
- Even a regular lightbulb can be used to “steal” other lightbulbs.

Fixed by the vendor.
Prior Work

Attacker

ZigBee (Radio)

Attacker Controlled

Ethernet

WiFi
Prior Work

Attacker

ZigBee Factory Reset

Malicious OTA Update

Attacker Controlled

ZigBee (Radio)

Ethernet

WiFi
New Goal: Infiltrate the Network

ZigBee Exploit

Attacker

Controlled

Ethernet

WiFi

Attacker

Exploit
New Goal: Infiltrate the Network

Attacker

ZigBee Exploit

Attacker Controlled

WiFi

Ethernet
Getting Started
ZigBee 101

▷ A suite of high level protocols for close proximity networks

▷ IEEE 802.15.4-based specification
  - low range / power radio (Not to be confused with IEEE 802.11 - WiFi)
  - Maximal Transmission Unit (MTU) of only **127 bytes**!

▷ ZigBee has a full network stack of its own
ZigBee 101

- **Physical (PHY) layer** – 2.4 GHz Radio
- **Medium Access Control (MAC)**
- **Network (NWK) Layer**
- **Application Sublayer (APS)**
- **ZigBee Cluster Library (ZCL)**
- **ZigBee Device Profile (ZDP)**

**Levels 1-2**
- **IEEE 802.15.4**
  - Physical (PHY) layer – 2.4 GHz Radio

**Levels 3**
- Network (NWK) Layer

**Levels 4**
- Application Sublayer (APS)

**Levels 5+**
- Some Application
- ZigBee Cluster Library (ZCL)
- ZigBee Device Profile (ZDP)
Meet our target

◊ Philips Hue smart lighting (now under “Signify”)
◊ Signify controls ~ 31% of the market in the UK
◊ We are going to focus specifically on the bridge
  ◊ Connected to both ZigBee (radio) and Ethernet
  ◊ Specifically, hardware version v2
Tearing down the bridge

Serial Debug

ZigBee “Modem”
ATSAMR21E18E

Main CPU
QCA4531-BL3A
Rooting the bridge

- The main CPU is of MIPS architecture
- The operating system is Linux (not an RTOS this time)
- @colinoflynn details in his blog how to root the bridge:
- Once finished, we get a root SSH connection 😊
ipbridge

- A single process that acts as the “brain”:
  - Parsing incoming ZigBee messages
  - Maintaining protocol state machines
  - ...

- `ipbridge` runs with `root` privileges 😊

- This is going to be the target process for our research
Looking for Vulnerabilities
Slow start

- Due to technical issues I couldn’t root the bridge
  - Didn’t have the right equipment
  - And the package we ordered got delayed (snail mail) 😞
- Meanwhile, started working on an old firmware version
  - Version from the original research (2016-2017)
Analyzing ipbridge

- At first glance, we saw something odd
- The code expects *strings* in the incoming message
  - The MTU is 127 bytes
  - The message should use *bits*
- What is going on here?
Things get complicated

◊ We forgot about the ZigBee “Modem”
  ◊ Uses Atmel BitCloud SDK to parse incoming messages
  ◊ Acts as a co-processor that handles the ZigBee lower network layers
◊ It converts the parsed messages to textual form and sends them over serial to the main CPU
◊ We don’t have the firmware for it – a black-box...
Living alongside a black-box

✧ The modem reduces the attack surface on the main CPU
  ✧ Complex parsing is offloaded to the Modem
  ✧ We don’t fully control messages sent to the main CPU

✧ Adds a huge uncertainty to everything we find
  ✧ Maybe the modem checks it?

✧ Let’s add it to the list of issues, and hope for the best …
Vulnerability Attempt #1

- The ZigBee Cluster Library (ZCL) manages configurations
  - Offers a `READ_ATTRIBUTE / WRITE_ATTRIBUTE` interface
  - Supports multiple types: `E_ZCL_BOOL (0x10)`
    - `E_ZCL_UINT8 (0x20)`
    - `E_ZCL_UINT32 (0x23)`
    - `E_ZCL_ARRAY (0x48)`
  - Hmm, variable-sized data type in an embedded device
  - This looks promising
Vulnerability Attempt #1

B1A0  move $a0, $s3
B1A4  jal **EI_zcl_read_1_byte** # 1-byte length field
B1A8  sw $v0, zcl_attribute.one_1f_data_two_1f_pdata($s1) # Store Word
B1AC  sb $v0, zcl_attribute.data_type_or_pdata_length($s1) # Store Byte
B1B0  jal **EI_malloc_with_mutex** # Jump And Link
B1B4  li $a0, 0x2B # '+' # buffer of 0x2B bytes (43 bytes)
B1B8  move $a0, $v0
B1BC  move $a1, $zero
B1C0  li $a2, 0x2B # '+' # Load Immediate
B1C4  jal memset # memset(pBuffer, 0, sizeof(buffer) = 0x2B)
B1C8  sw $v0, zcl_attribute.data_or_pdata($s1) # Store Word
B1CC  lw $a0, zcl_attribute.data_or_pdata($s1) # Load Word
B1D0  lbu $a1, zcl_attribute.data_type_or_pdata_length($s1) #
B1D0  # EI-DBG: Controlled (1-byte) memcpy into a
B1D4  move $a2, $s3
B1D8  sw $s0, 0x38+var_28($sp) # Store Word
B1DC  jal **EI_zcl_read_blob** # Jump And Link

# EI-DBG: Controlled (1-byte) memcpy into a
# EI-DBG: fixed size 0x2B heap buffer.
# EI-DBG: ==> Heap Buffer Overflow :)}
Vulnerability Attempt #1

◊ It isn’t a vulnerability until we have a PoC to trigger it
  ◊ We don’t have the latest firmware yet
  ◊ The Modem could block large ZCL arrays
  ◊ We don’t have radio equipment to transmit the attack and test it

◊ Finally, the package arrived
  ◊ We can root the bridge and extract the latest firmware
Array? String!

Yup, this firmware contains symbols!
Nope, No Vulnerability

◊ Latest firmware version dropped support for ZCL Arrays
  ◊ Supports ZCL Strings instead
  ◊ Sadly for us, strings are parsed correctly

◊ Time to search for other vulnerabilities

◊ Covered most of the firmware, and found nothing…
Nope, No Vulnerability

- Latest firmware version dropped support for ZCL Arrays
  - Supports ZCL Strings instead
  - Sadly for us, strings are parsed correctly
- Time to search for other vulnerabilities
- Covered most of the firmware, and found nothing...
- Who handles the incoming ZCL strings later on?
Vulnerability Attempt #2
Vulnerability Attempt #2

◊ Someone forgot to finish the migration from **Array to String**
  ◊ Should have been marked with **0x0F** for String
  ◊ However, internally they are still marked with **0x10** for Array

◊ The original vulnerability still exists in the code 😊
  ◊ Just buried a bit deeper, that’s all

◊ Time to start transmitting ZigBee messages
Commissioning?
Playing around with ZigBee

- Like the previous research, chose to use ATMEGA256RFR2-XPRO
  - Support sending/receiving ZigBee IEEE 802.15.4 radio frames
  - Should be computationally equivalent to a lightbulb
- Timing constraints dictate we use C code executed on the board
  - Our entire ZigBee code + exploit will be implemented in C
- The vulnerable flow is accessible during Commissioning
Commissioning?

- The process of pairing and associating a new lightbulb
  - Classic Commissioning
  - Touchlink Commissioning

- The Philips Hue app initiates **Classic Commissioning**

- In theory, the ZigBee specs explains the entire process

- In practice, a lot of room for vendors to do as they wish
Analyzing the protocol

✧ No documented flow - What message is supposed to be sent? When?
✧ Can’t sniff a full conversation – Too many messages, sent too fast…
✧ We need the (Broadcast) transport crypto key, how do we get it?
  ✧ Not the first to tackle this issue - https://peeveeone.com/?p=166
✧ Analysis & implementation took a lot of effort, but it worked!
No documented function to send packets.

Can’t sniff a full connection.

We need the (Bro) patents, how do we get it?

Not the first to talk about it.

Analysis & implementation:

We open sourced our full .pcap of a classic commissioning:

github.com/CheckPointSW/Cyber-Research/tree/master/Vulnerability/Smart_Lightbulbs
Lessons Thus Far

- Without user interaction, the bridge won’t accept new lightbulbs
- We have ~1 minute to commission as many lightbulbs as we want
- The user will see the lightbulb in the app only after the “ZCL phase”

**Good News:** Managed to trigger the vulnerability during the ZCL phase!
Lessons The:

- Without user interaction, the bridge pushes lightbulbs
- We have ~1 minute to commission as
- The user will see the lightbulb in the absence

- **Good News:** Managed to trigger the vulnerability during the ZCL phase!

- No state machine check – send whatever response you like
- However, can only trigger the vulnerability **during** this phase
Let the Exploitation Begin
Vulnerability Recap

◊ We have a linear buffer overflow over the heap

◊ Our buffer size is limited to 70 controllable bytes

◊ ZCL is quite high in the ZigBee stack, and the initial MTU is only 127 bytes

◊ We don’t have any byte constraints on our payload

◊ The destination buffer is allocated on the heap

◊ Fixed size of 0x2B (43) bytes
The Heap

- The bridge uses uClibc – (old) embedded libc implementation
  - Chosen heap implementation is malloc-standard (dlmalloc)
- Much like glibc, but with less sanity checks
- All of our free buffers will fall into the range of the fastbins
  - Bin for each buffer size (multiple of 8) starting from 0x10
  - Each bin contains a singly-linked list of free()ed buffers
/dev/null fastbin

Code snippet from `free()`:

```c
if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
{
    set_fastchunks(av);
    // EI-DBG: Who checks that size >= 0x10?
    // EI_DBG: Lower size will get us indices -1 and -2
    fb = &((av->fastbins[fastbin_index(size)]));
    p->fd = *fb;
    *fb = p;
}
```

/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) (((((unsigned int)(sz)) >> 3) - 2)
/dev/null fastbin

```c
struct malloc_state {
    /* The maximum chunk size to be eligible for fastbin */
    size_t   max_fast;  /* low 2 bits used as flags */

    /* Fastbins */
    mfastbinptr fastbins[NFASTBINS];
}
```

- Index -1 ➔ Store the buffer on top `max_fast` – too risky
- Index -2 ➔ Store the buffer on an **unused** variable, creating a ghost linked list acting as /dev/null
Heap Overflow Plan

- Modify only `size` and `ptr`

```
<table>
<thead>
<tr>
<th>size</th>
<th>User's data buffer</th>
<th>prev_size</th>
<th>size</th>
<th>ptr</th>
<th>User's free buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04 - 0x2C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>Some buffer content</th>
<th>garbage</th>
<th>0x01</th>
<th>ptr'</th>
<th>User's free buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04 - 0x2C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Goal: Confuse `malloc()` to “allocate” a buffer at an arbitrary address
Heap Shaping Strategy

- Overflowed a free `fastbin` buffer? – **Bingo!** this is what we aim at
- Overflowed a used buffer? – When `free()`ed it will go to `/dev/null`
- Overflowed a used buffer that lives forever? – oh well
- Overflowed a free large buffer? – We will probably crash soon 😞
- If done correctly, we will get the desired `Malloc-Where`
Heap Shaping Strategy

- Malloc-Where will grant us the ability to write on the GOT
  - Global-Offset-Table
  - A table full of function pointers used to execute library functions
- The GOT is at a fixed address 😊
- The modified fptr will jump to our shellcode*
- *Sounds easy on paper, way harder in real life
Shellcode: Theory vs Reality
Location, Location, Location

- We need to store a **binary** shellcode in a **fixed** global address
- The problem - we get **textual** messages from the ZigBee Modem
- Found only one good candidate for such a buffer
  - The ZigBee “phone book”
  - Array of ZigBee addresses seen / advertised thus far
  - Can hold up to 65 records of 16 bytes each (~1KB)
The “Neighbor Record”

- **Bytes 0x00-0x08**: Extended network address - Fully controlled.
- **Bytes 0x09-0x0A**: Short network address - Fully controlled.
- **Bytes 0x0A-0x10**: Misc fields - Uncontrolled.

- Oh, and about that

- The bridge is unstable when it gets > 20 records
  - This is going to be a **very** small shellcode
Initial plan

- It seems infeasible to restore the execution flow
- Instead, the shellcode will patch the binary with a backdoor
- After a crash, the daemon will restart with our backdoor

Problem:

- The patch and the file-path don’t fit in 10 consecutive bytes (each)
“Ideal” Shellcode

- Use the 10 consecutive bytes per record to build a decoder
- In mips16 a jump to the next record only costs us 2 bytes

Problems:

- We need to clear the cache before jumping to the unpacked shellcode
- If we sleep(), the watchdog kills the process
- We don’t have enough records to silence the watchdog
“Bold” Shellcode

◊ We will restore the execution flow, we have no choice
◊ This means we mprotect() and install the backdoor in RAM!
◊ A few days, and one hand-crafted shellcode later:
  ◊ The shellcode fully restores the execution (GOT, heap, everything)
  ◊ The shellcode costs us 16 records – well in budget
Connecting the dots

◊ The backdoor shellcode gives us an **Arbitrary-Write** primitive
◊ Our exploit fakes a “legitimate” lightbulb that will leverage it
◊ Used the **Arbitrary-Write** to write **Scout**’s loader to memory
◊ Upon execution, Scout loaded the full payload - **EternalBlue**

![Command output]

◊ Time for a demo 😊
Full Exploit Demo

Link: https://www.youtube.com/watch?v=4CWU0DA_bY
Coordinated Disclosure

◊ Vulnerability was reported to Signify on the 5\textsuperscript{th} of Nov 2019
   ◊ The vendor confirmed the vulnerability on the \textbf{same day!} (impressive)

◊ Signify issued a patch via an automatic update on Jan 2020
   ◊ Full details & Advisory in our blog post – \textbf{CVE-2020-6007}:
     ◊ \url{https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/}

◊ All products should have received the update by now
Conclusions

◊ Even with an MTU of 127 bytes, ZigBee vulns are exploitable

◊ Security mitigations only work when they are on-by-default
   ◊ Static binary for ipbridge, no stack canaries, writable GOT, ...
   ◊ ASLR for heap, stack and loaded libraries (thank you Linux)

◊ Smart devices are becoming popular by the minute, and yet, we can’t even trust our lightbulb...
Kudos

◊ Special thanks to everyone that helped make this research possible
  ◊ Eyal Ronen (@eyalr0) – Research idea & active guidance
  ◊ Colin O’Flynn (@colinoflynn) – Detailed writeups on rooting the bridge
  ◊ Peter – Publishing the ZigBee transport keys for the lightbulb
  ◊ Yaron Itkin – For the crucial hardware support along the way
    ◊ Thanks little brother 😊

◊ And finally, to the entire cp<r> team for their support
Until next time

@EyalItkin