BlueBorne™

A New Class Of Airborne Attacks Compromising Any Bluetooth Enabled Linux/IoT Device

armis

BlueBorne™

Ben Seri, Head of Research
Gregory Vishnepolsky, Researcher

Agenda

- Airborne Attacks
- Brief Bluetooth background
- The BlueZ stack on Linux
- Remote Exploitation of Linux kernel BlueZ vulns
- DEMO

Modern Airborne Attacks

BROADPWN

BlueBorne™

Key Reinstallation Attack

GOOGLE PROJECT ZERO

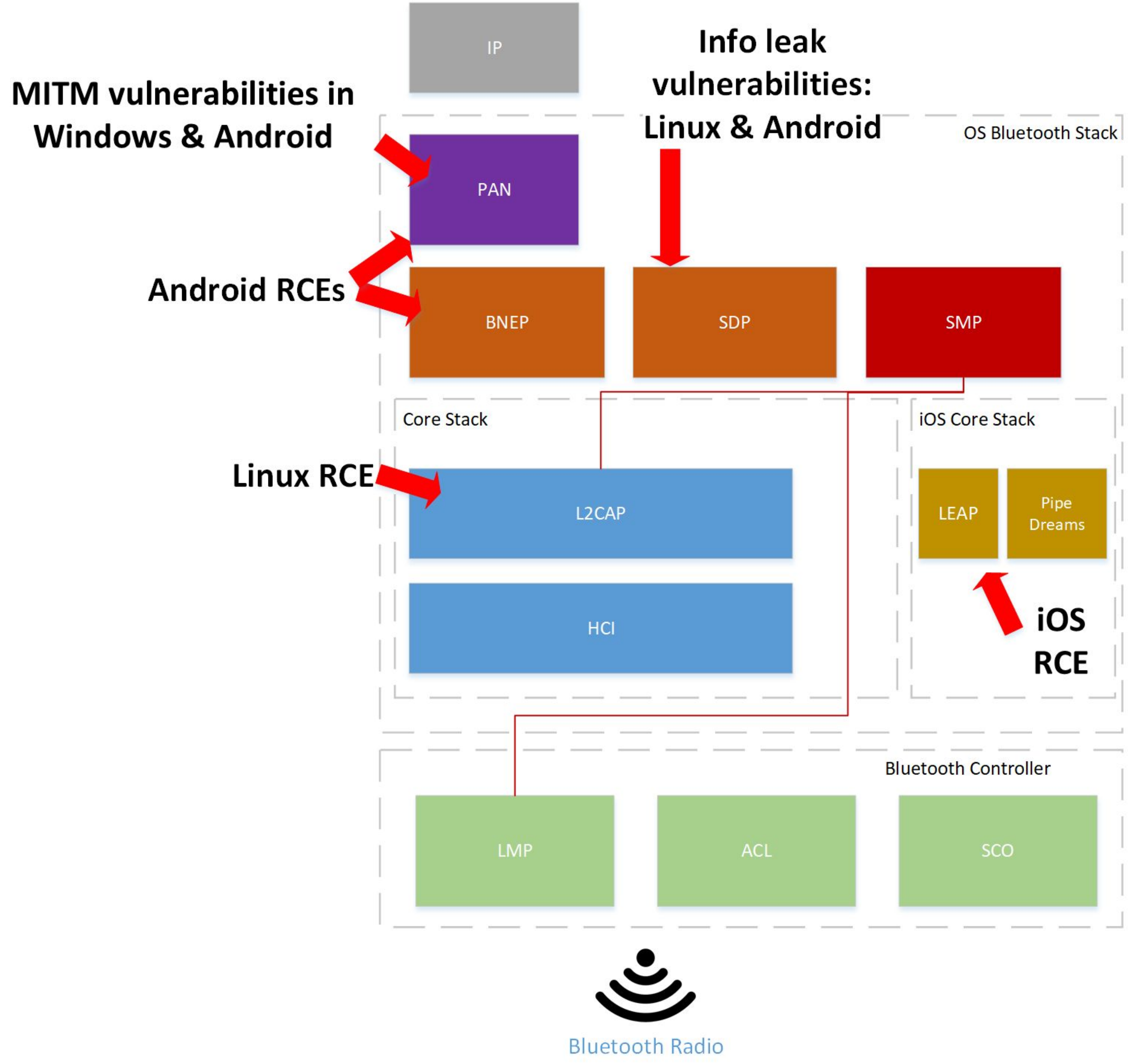
RCE on Broadcom Wifi FW

New Attack Vector Identified

- 5.3B Devices At Risk
- 8 Vulnerabilities (4 critical)
- Android, Windows, Linux, and iOS
- Most serious Bluetooth vulnerabilities to date
- No user interaction or authentication required
- Enables RCE, MiTM and Info leaks

BlueBorne™

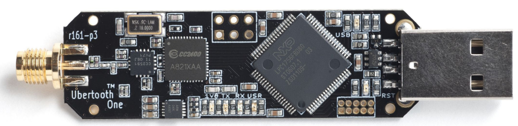
8 New Vulnerabilities



Demystifying Discoverability

Discoverability is not a prerequisite for establishing a connection

- Bluetooth devices transmit parts of their MAC address over the air (LAP)
- Sniffing a single packet enables brute force of the MAC (only 32 options)
- Open source tools allow attackers to find “undiscoverable” MACs (Ubertooth for example)



Ubertooth

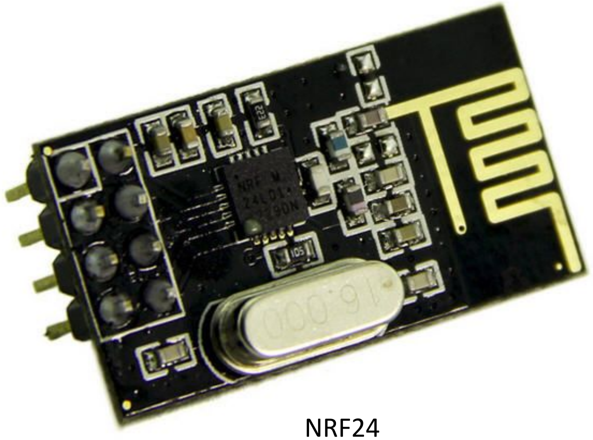
$100 range solution

LSB 68/72	54	0 - 2745 MSB
ACCESS CODE	HEADER	PAYLOAD

Figure 6.1: General Basic Rate packet format.

Demystifying Discoverability (continued)

- Sniffing BT packets can be done on standard BT Chips/Adapters with FW modification (not easy, but has been done before)
- Can easily be done using certain 2.4GHz transceiver ICs such as nRF24L01+
  - Needs to support RX on 1MHz wide channels, with GFSK modulation
  - Promisc sniffing “trick” by Travis Goodspeed
- $0.7 solution. Our code for nRF24 is on github
- https://github.com/armissecurity

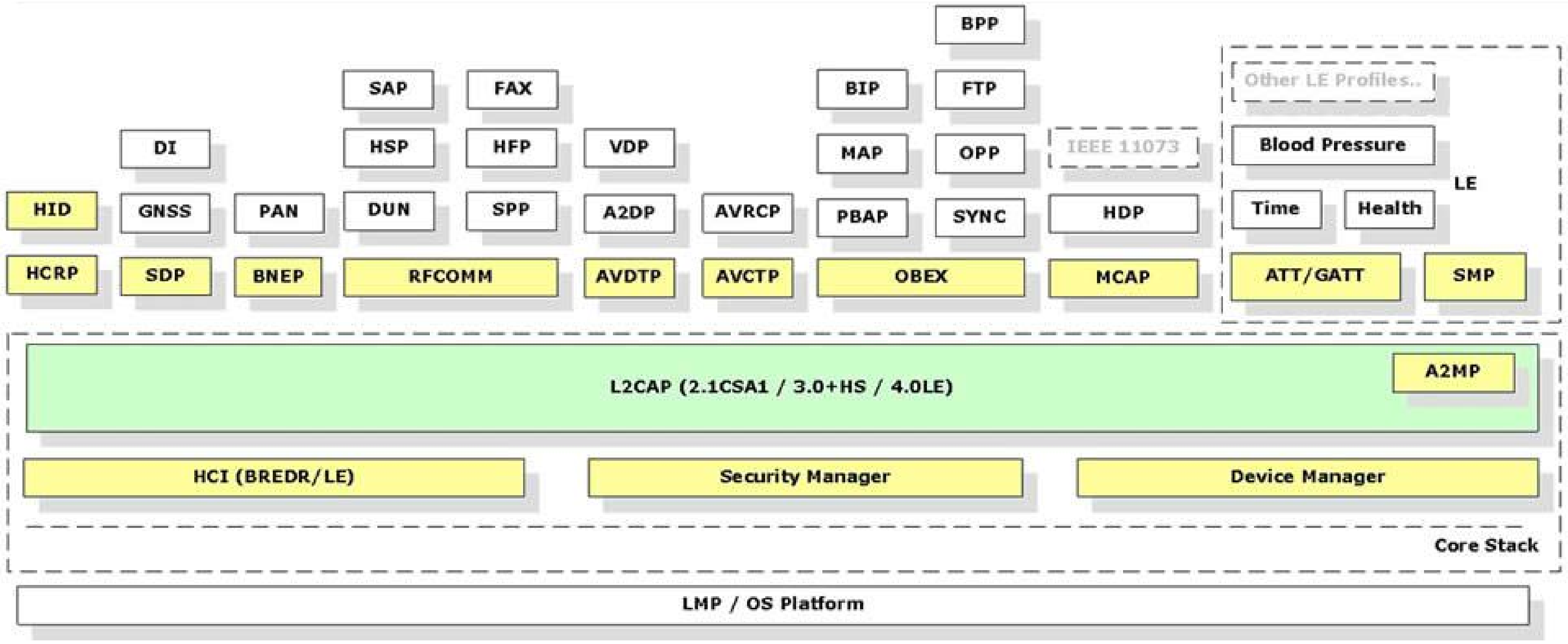


NRF24

Demystifying Discoverability (continued)

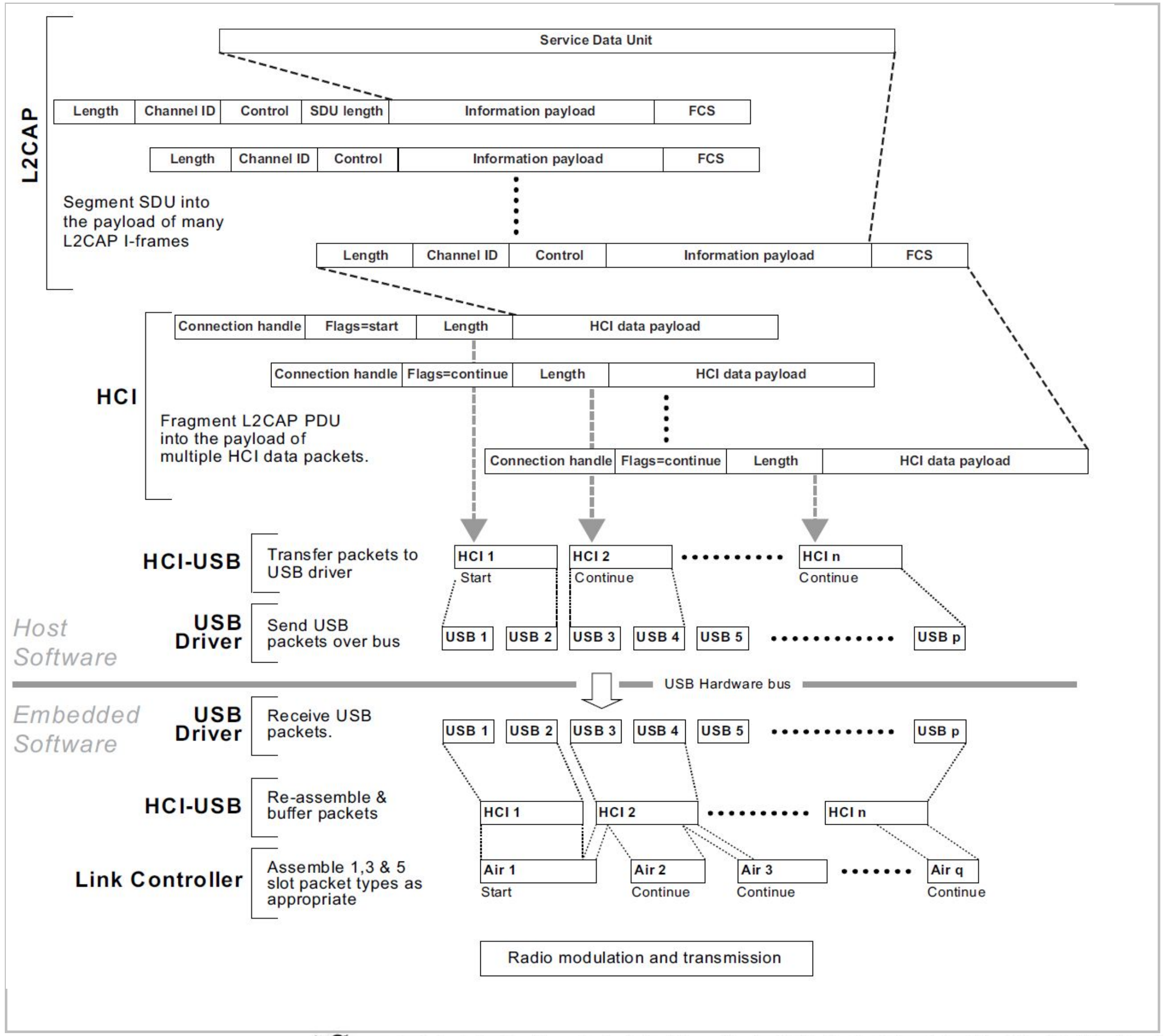
- A lot of OEMs use adjacent MACs for WiFi/Bluetooth
- Use WiFi monitor mode to find nearby Bluetooth devices
- Attacker positioned on the same network as victim can also use ARP cache

Untapped, Very Wide Attack Surface



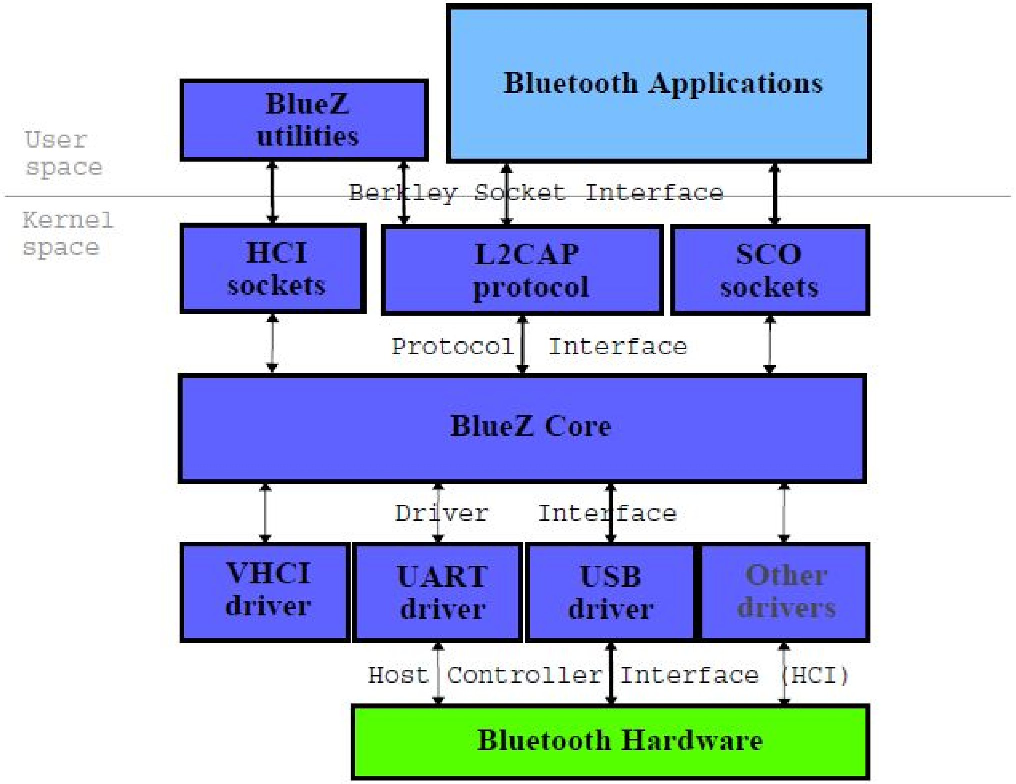
It’s complicated...

- Bluetooth Spec is 2822 pages long
- Some pages look like this
- Endless features and facilities
  (4 layers of fragmentation!)



BlueZ

- The Linux Bluetooth stack since 2001 (!)
- Talks to Bluetooth Controller HW devices
- Kernel side implementation of
  - H4, HCI event handling
  - ACL, SCO
  - L2CAP
- Userland implementation of higher layers
  - Bluetoothd daemon
  - Authentication, Pairing
  - SDP and BT services (HID, Audio, etc)
  - Runs as root

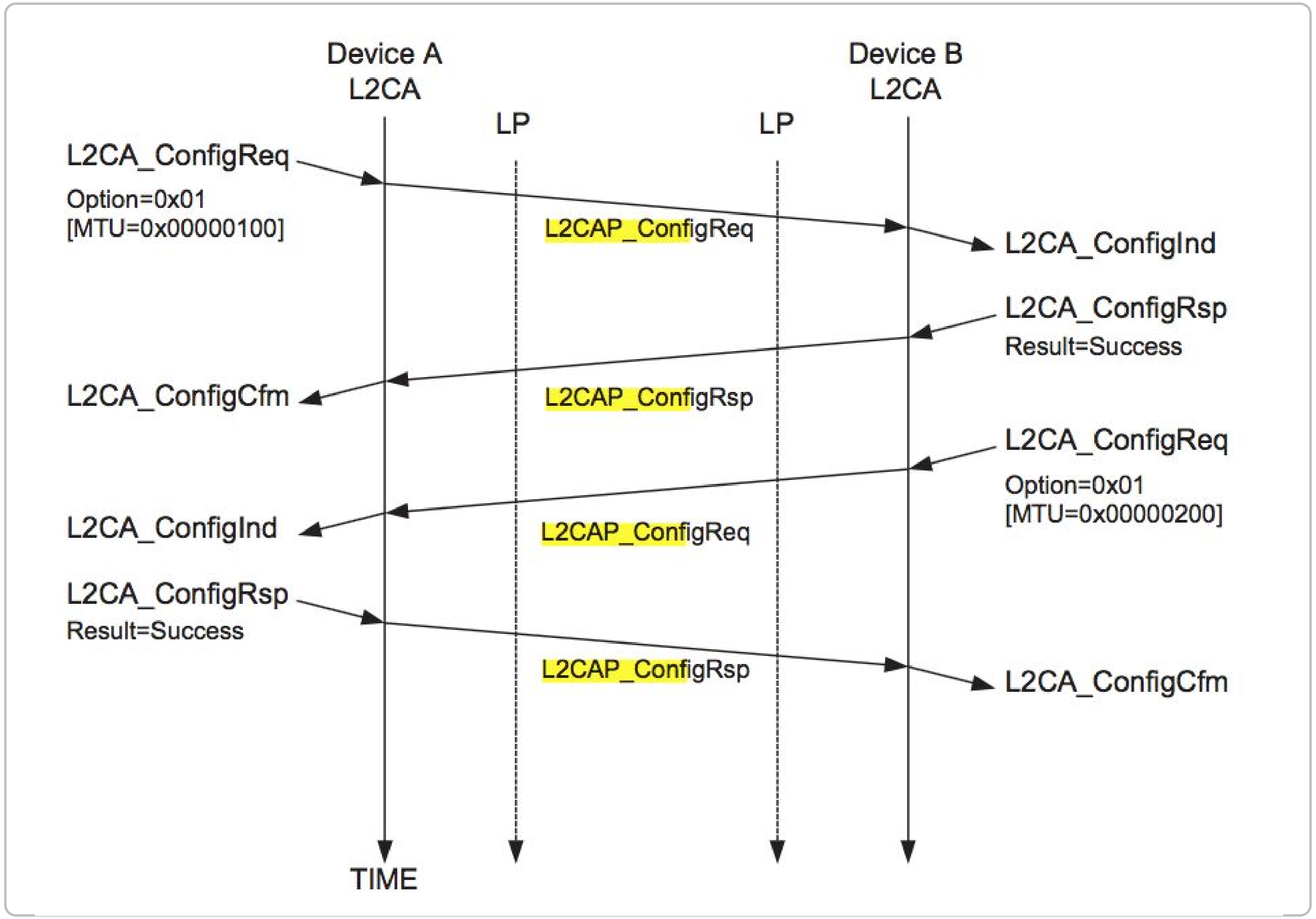


L2CAP

- The Bluetooth equivalent of TCP
  - Reliable connections over ACL packets
  - Listening “servers” on open “ports”
- Connecting to a port does not require authentication
  - Each service may request it later
- Lots of obscure QoS features == lots of code == attack surface
- L2CAP in BlueZ is implemented in Kernel

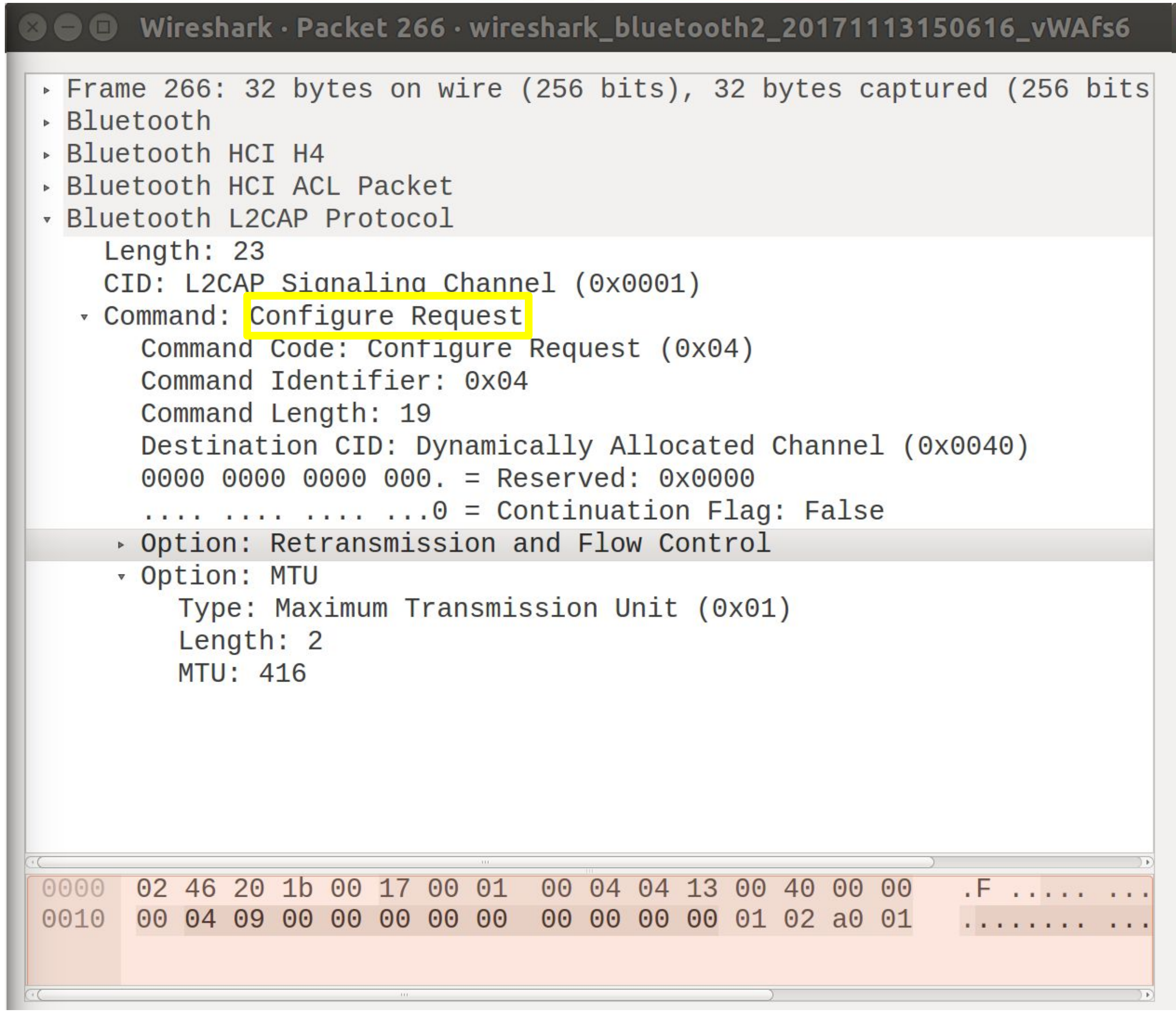
Mutual Configuration

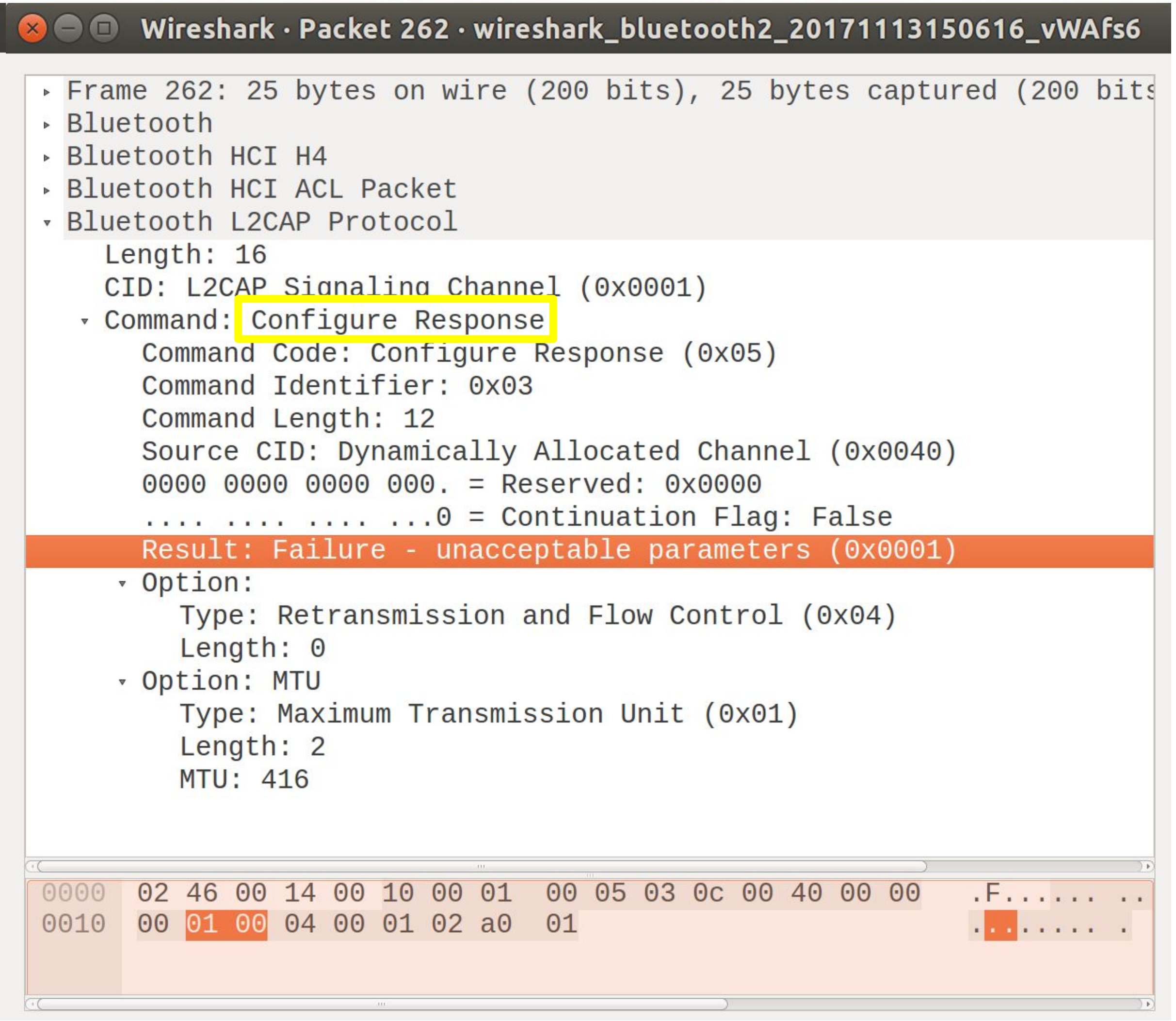
- Peers can negotiate parameters during connection establishment phase (e.g. MTU)
- Each side may send multiple ConfigReq and ConfigRsp packets
- The Result in ConfigRsp may also be **Unaccept** or **Pending**
- An Unaccepted ConfigRsp will be answered with a new ConfigReq
- Each new ConfigReq/Rsp will hold a reconstruction of the negotiated parameters



Excerpt from Bluetooth Spec, page 1902

Mutual Configuration Cont.





RCE in *l2cap_parse_conf_rsp* (CVE-2017-1000251)

```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);
        switch (type) {


        case L2CAP_CONF_MTU:
            // Validate MTU ...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;
        // ... Parsing and adding other various parameters
        }
```

`len` is the size of `rsp`. `data` is the output conf_req buffer, but it's size isn't given here...

Each param from `rsp` is added to the `data` buffer at `ptr` (and `ptr` is advanced). However, the bounds aren't checked...

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

RCE in *l2cap_parse_conf_rsp* on Linux v3.3-rc1+

```
switch (result) {
    case L2CAP_CONF_SUCCESS:
        // ...
        break;

    case L2CAP_CONF_PENDING:
        set_bit(CONF_REM_CONF_PEND, &chan->conf_state);
        if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
            char buf[64];
            len = l2cap_parse_conf_rsp(chan, rsp->data, len,
                                       buf, &result);
        // ...
        goto done;
```

The state of the connection needs to be *Pending*

The output data buffer buf is 64 bytes long on the stack :(

Excerpt from *l2cap_config_rsp* (net/bluetooth/l2cap_core.c)

Exploit Strategy

- Arrange ability to transmit arbitrary L2CAP_ConfRsp
- Overflow something significant on the stack (pointers)
    - Buffer must also be a valid L2CAP_ConfRsp
- Defeat possible mitigations
- Develop a write-what-where primitive
- Overwrite usermode-helper commands (to get root shell)
- Disable LSM (Linux Security Modules) if needed

Expected Stack Overflow Mitigations

- ASLR
- Stack canary (stack protector)
- FORTIFY_SOURCE (stack buffers are adjacent to the stack canary)
- NX-bit (DEP - Data is not executable, code is not writable)

Real World Kernel Configurations

- No KASLR (practically everywhere)
- Stack canaries enabled only in major linux distros
  - Not in default config
  - Almost never used in IoT devices
- Fortify source enabled, stack canary disabled (bad idea)
  - Samsung Tizen Watch
- No NX bit (wat?)
  - Amazon Echo (sad!)

Case Study #1 - Samsung Gear S3

- Kernel 3.18.14, Arm 64bit
- No KASLR
- No Stack canaries
- Fortify source enabled & stack canary disabled (bad combo)
    - First overflown qword is the frame pointer
- SMACK (access control)

Case Study #2 - Amazon Echo

- Kernel 2.6.37 (!), Arm 32bit
- No KASLR
- No Stack canaries
- No Fortify source
  - First overflown dword is the pointer to the output buffer (response)
- No NX Bit (!)
- No Access Control

Linux v2.6.32 Limited RCE Flow

```
switch (result) {
    ...
    case L2CAP_CONF_UNACCEPT:
      ...
      char req[64];
      if (len > sizeof(req) - sizeof(struct l2cap_conf_req)) {
          l2cap_send_disconn_req(conn, sk, ECONNRESET);
          goto done;
      }

     result = L2CAP_CONF_SUCCESS;
     len = l2cap_parse_conf_rsp(chan, rsp->data, len,
                                req, &result);
     ...
```

Input configurations are limited in length (60 bytes)

Output buffer (req) is still 64 bytes on the stack

Getting Out of Bounds

```
...
while (len >= L2CAP_CONF_OPT_SIZE) {
    len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);
    switch (type) {
    ...

    case L2CAP_CONF_RFC:
        if (olen == sizeof(rfc))
            memcpy(&rfc, (void *)val, olen);

        l2cap_add_conf_opt(&ptr, L2CAP_CONF_RFC, sizeof(rfc),
                                         (unsigned long)&rfc);
        break;
```

A zero length config element will result in a element added to the output response with it's default length

Excerpt from ***l2cap_parse_conf_rsp*** (net/bluetooth/l2cap_core.c)

Getting Out of Bounds [2]

- Our input element:

0x04 == L2CAP_CONF_RFC

Type	Length
04	00

Zero len RFC

- Turns into an 11 byte output element

- For example: sending 30 zero-len-RFCs will overwrite data way *out of bounds*:
  - `30 * zero-len-rfcs = 60 (max)`
  - `30 * output-rfcs   = 330`

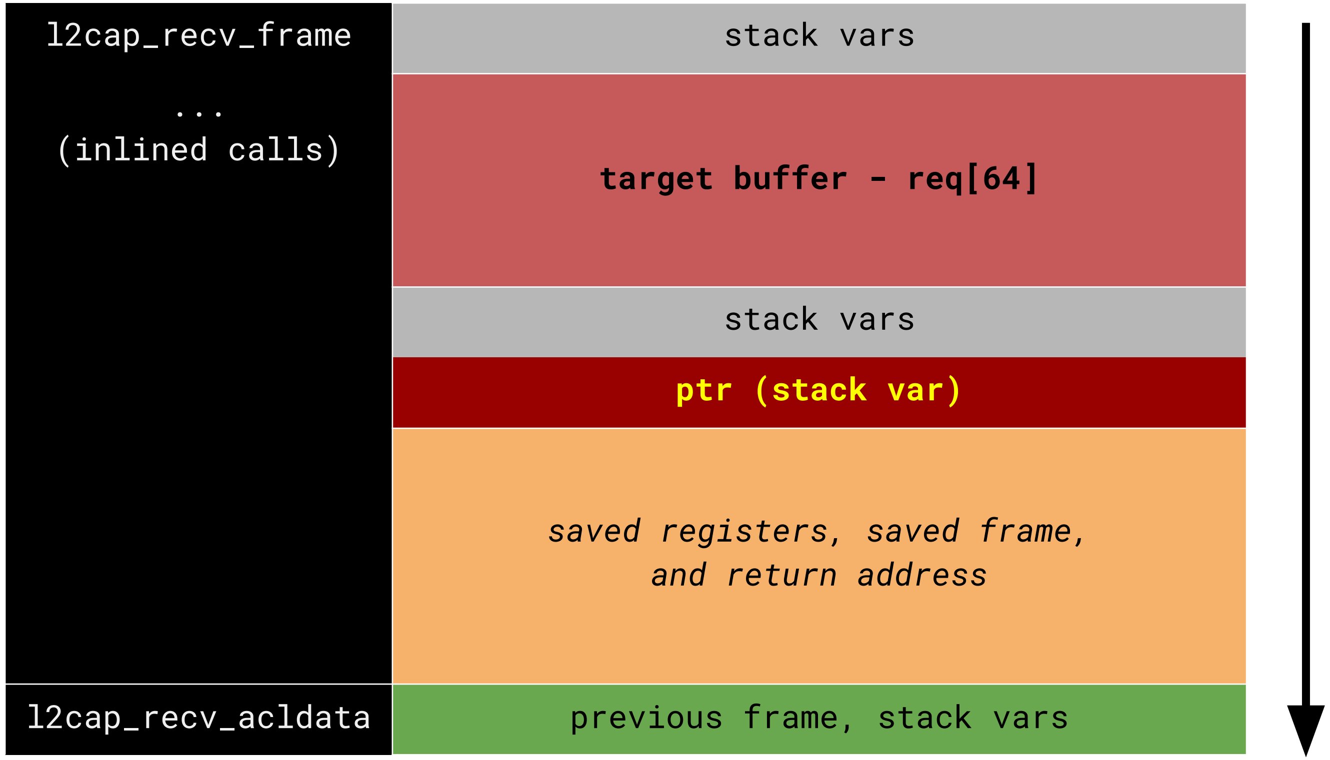
BDD8	req	04	09	XX	XX
BDDC		XX	XX	XX	XX
BDE0		XX	XX	XX	--
BDE4		--	--	--	--
BDE8		--	--	--	--
BDEC		--	--	--	--
BDF0		--	--	--	--
BDF4		--	--	--	--
BDF8		--	--	--	--
BDFC		--	--	--	--
...		...	...	...	...
BED4		*Out of bounds data*			

Uncontrolled data (9 bytes)

Stackframe from Amazon Echo v591448720

Analyzing the stack

- **ptr** points to the next element in the output buffer
- Sending 24 zero-len-RFCs will bring us to overwrite **ptr**:
  - `24 * zero-len-rfcs = 48`
  - `24 * output-rfcs = 264`
- After overwriting **ptr**, the **next** parsed element can be written **anywhere**



Stackframe from Amazon Echo v591448720

Creating a Write-What-Where Primitive

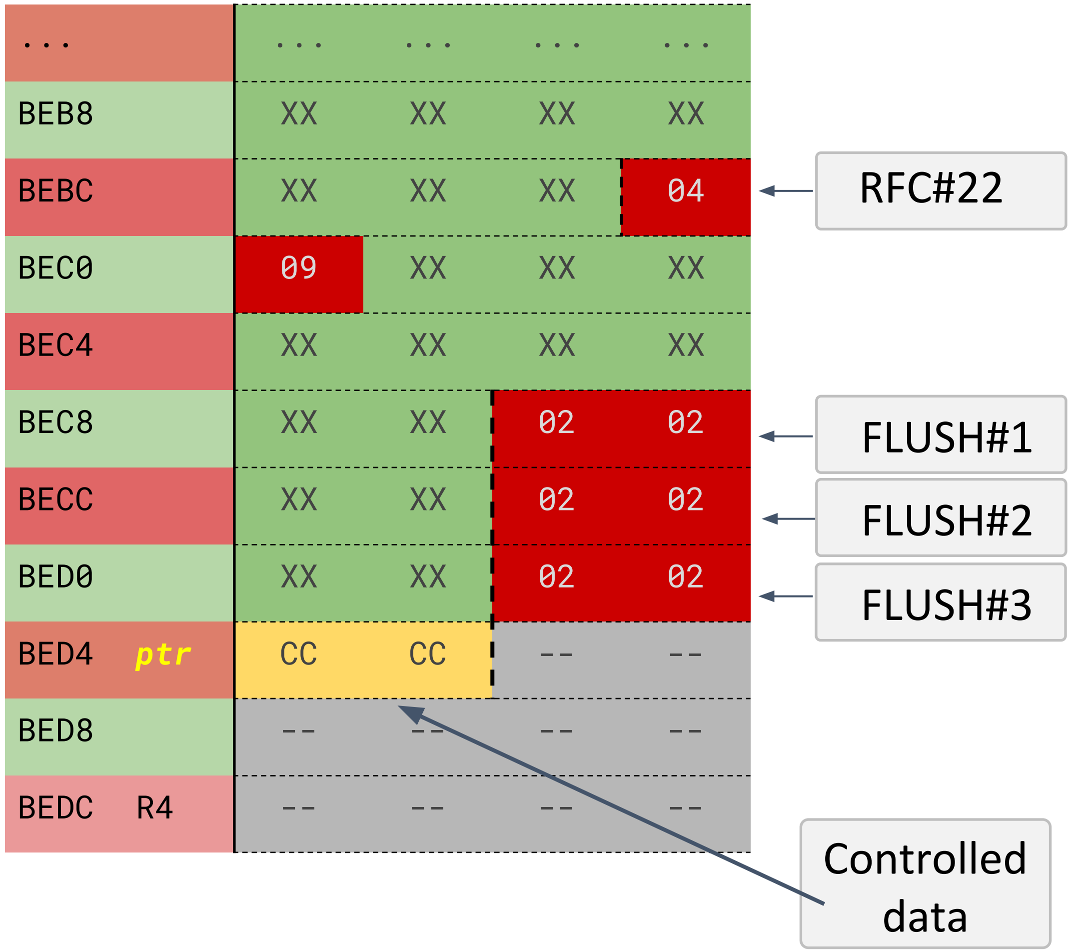
- 24 RFCs won't allow us to control `ptr` due to alignment
- We need other elements to align our overflow of the `ptr`
- Using FLUSH or MTU elements can enable proper alignment

BDD8	req	04	09	XX	XX
BDDC		XX	XX	XX	XX
BDE0		XX	XX	XX	04
BDE4		09	XX	XX	XX
...		...	...	...	...
...		...	...	...	...
BEC8		XX	XX	04	09
BECC		XX	XX	XX	XX
BED0		XX	XX	XX	XX
BED4	ptr	XX	04	09	XX
BED8		XX	XX	XX	XX
BEDC	R4	XX	XX	XX	XX

`ptr` is not controlled

Controlling Write-What-Where

- We send 22 empty RFCs
- Additional 2 empty FLUSH elements (for alignment)
- And lastly: We send a FLUSH element (#3) to control lower half of **ptr** (little endian)
- Now we can write an additional element anywhere on the stack!
- Reminder: **ptr** is where the next element is written to.



Controlling Write-What-Where [2]

...		...	...	...	...
BEB8		XX	XX	XX	XX
BEBC		XX	XX	XX	04
BEC0		09	XX	XX	XX
BEC4		XX	XX	XX	XX
BEC8		XX	XX	02	02
BECC		XX	XX	02	02
BED0		XX	XX	02	02
BED4	ptr	FC	BE	--	--
BED8		--	--	--	--
BEDC	R4	--	--	--	--

end of stackframe

arbitrary ptr value

ptr is now here

1. Element overflowing *ptr*

BED4	ptr	FC	BE	--	--
BED8		--	--	--	--
BEDC	R4	--	--	--	--
BEE0	R5	--	--	--	--
BEE4	R6	--	--	--	--
BEE8	R7	--	--	--	--
BEEC	R8	--	--	--	--
BEF0	R9	--	--	--	--
BEF4	R10	--	--	--	--
BEF8	R11	--	--	--	--
BEFC	LR	02	02	XX	XX

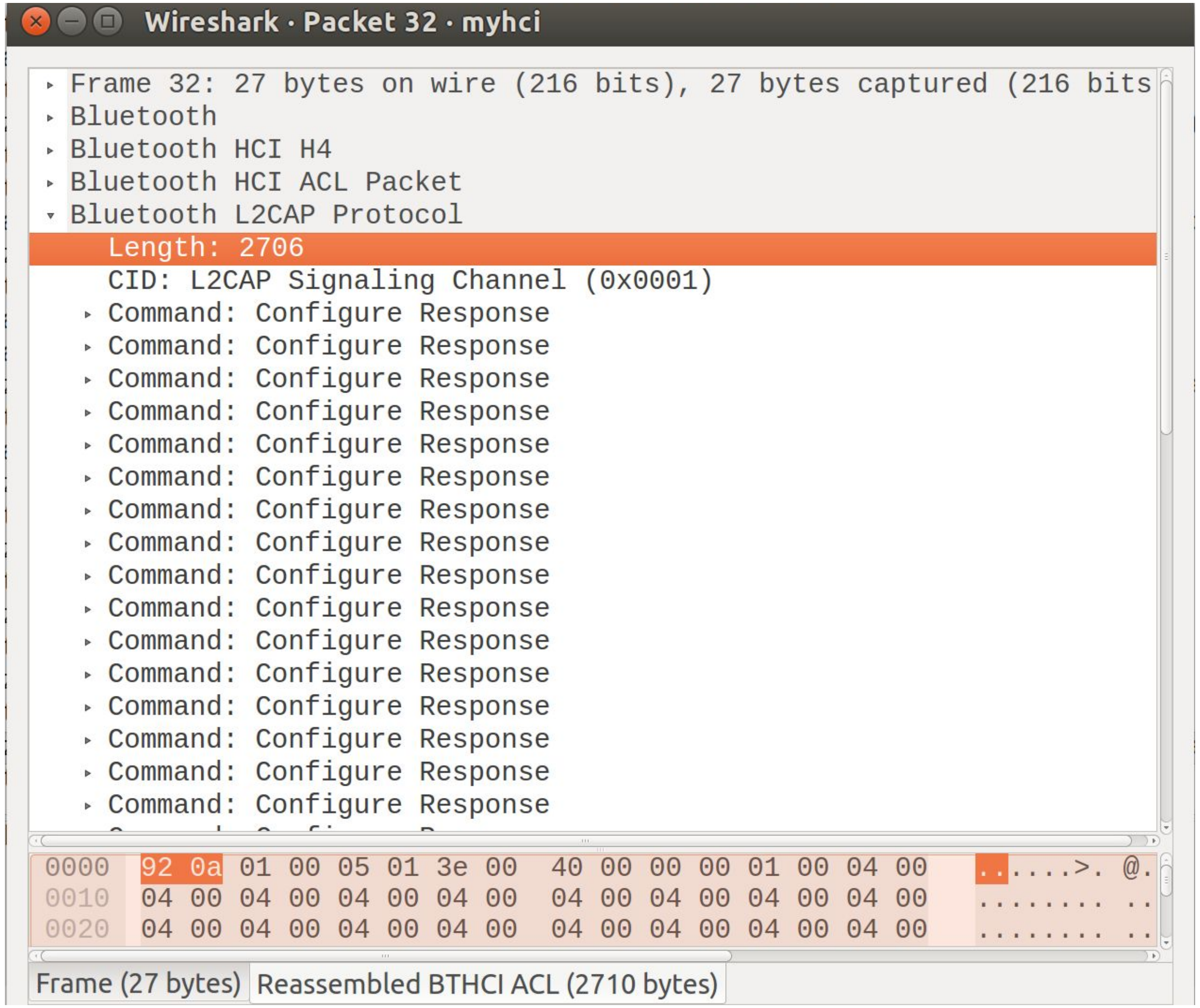
Control of LR

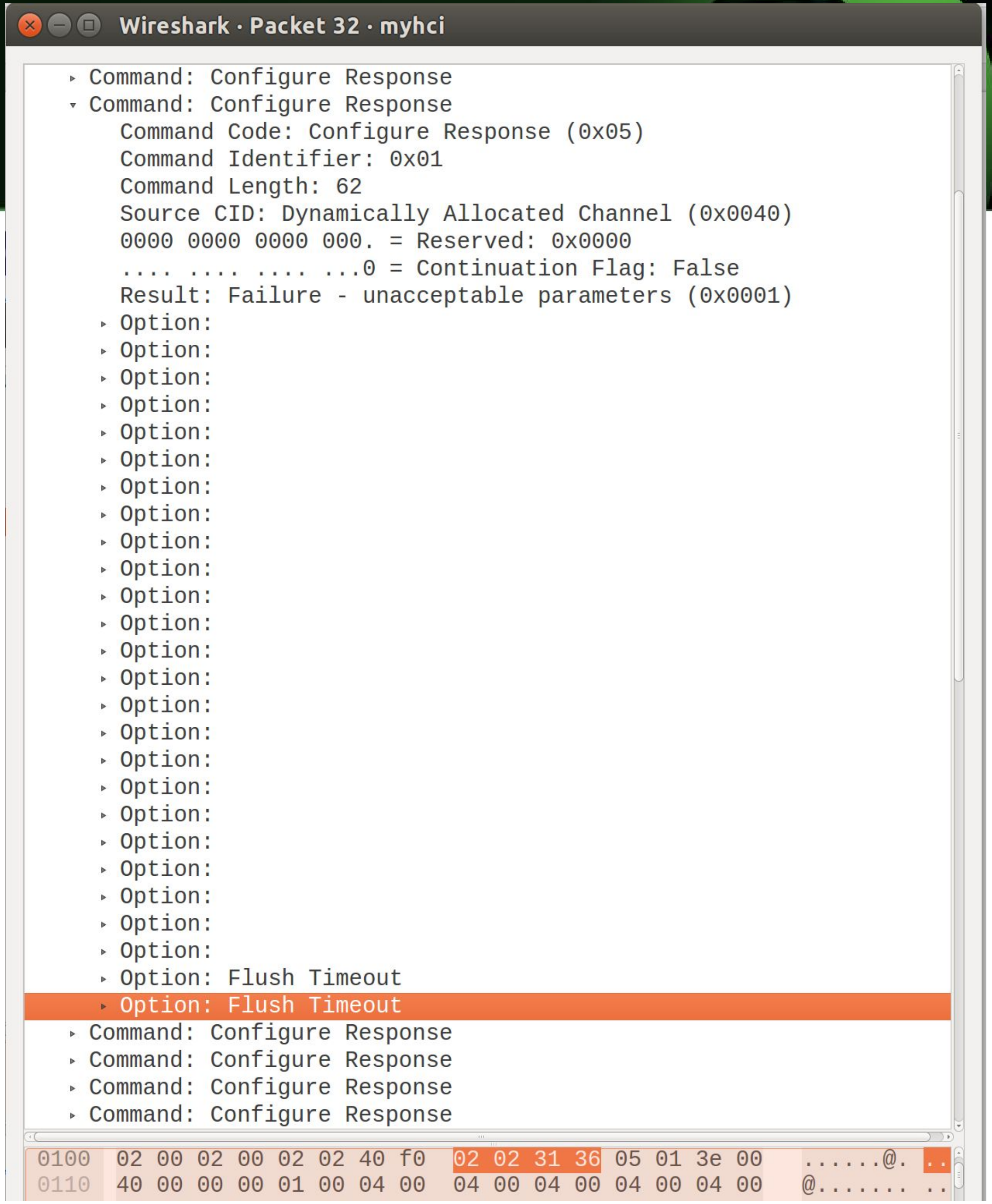
2. Next element written

Packaging an Attack Buffer

- Each *ConfRsp* command we send allows a Write-What-Where of 2 bytes anywhere on the stack
- Conveniently, L2CAP allows packing multiple commands into 1 packet. This allows sending multiple *ConfRsp*'s at once (essential for overflowing all 4 bytes of LR)
- We'll use that to write a shellcode somewhere on the stack, word by word, and then point LR there (No NX-bit)

Packaging Attack Buffer





Usermode helpers

- We’ve got Kernel mode code exec. We want a root shell.
- *orderly_poweroff* function runs a command in userspace that is supposed to shut the machine down gracefully
- *poweroff_cmd* is a global (writeable) string in kernel memory that holds that command.
- Our payload writes a bash connectback to *poweroff_cmd*, and then calls *orderly_poweroff*

Exploit Recap (Amazon Echo)

- Begin an L2CAP connection, with a high MTU
- Inject a crafted packet with multiple ConfRsp's:
  - Each ConfRsp writes 2 bytes of payload to an unused area on the stack
  - The last 2 ConfRsp's point the LR to the payload
- The payload is a shellcode that overwrites poweroff_cmd and causes a bash command to be executed.
  - Finally, it restores execution (jumps back)

Linux 3.18.14 RCE flow

- Performed on Gear S3 Smartwatch
  - No limitation to size of *ConfRsp* on this newer kernel
  - NX-bit enabled, arm-64, FORTIFY_SOURCE
- But no stack canary… Therefore, we overflow LR directly
- Point LR to a stack pivot, executing a ROP chain from our *ConfRsp*.
- ROP performs the same usermode helpers trick

Defeating modern mitigations

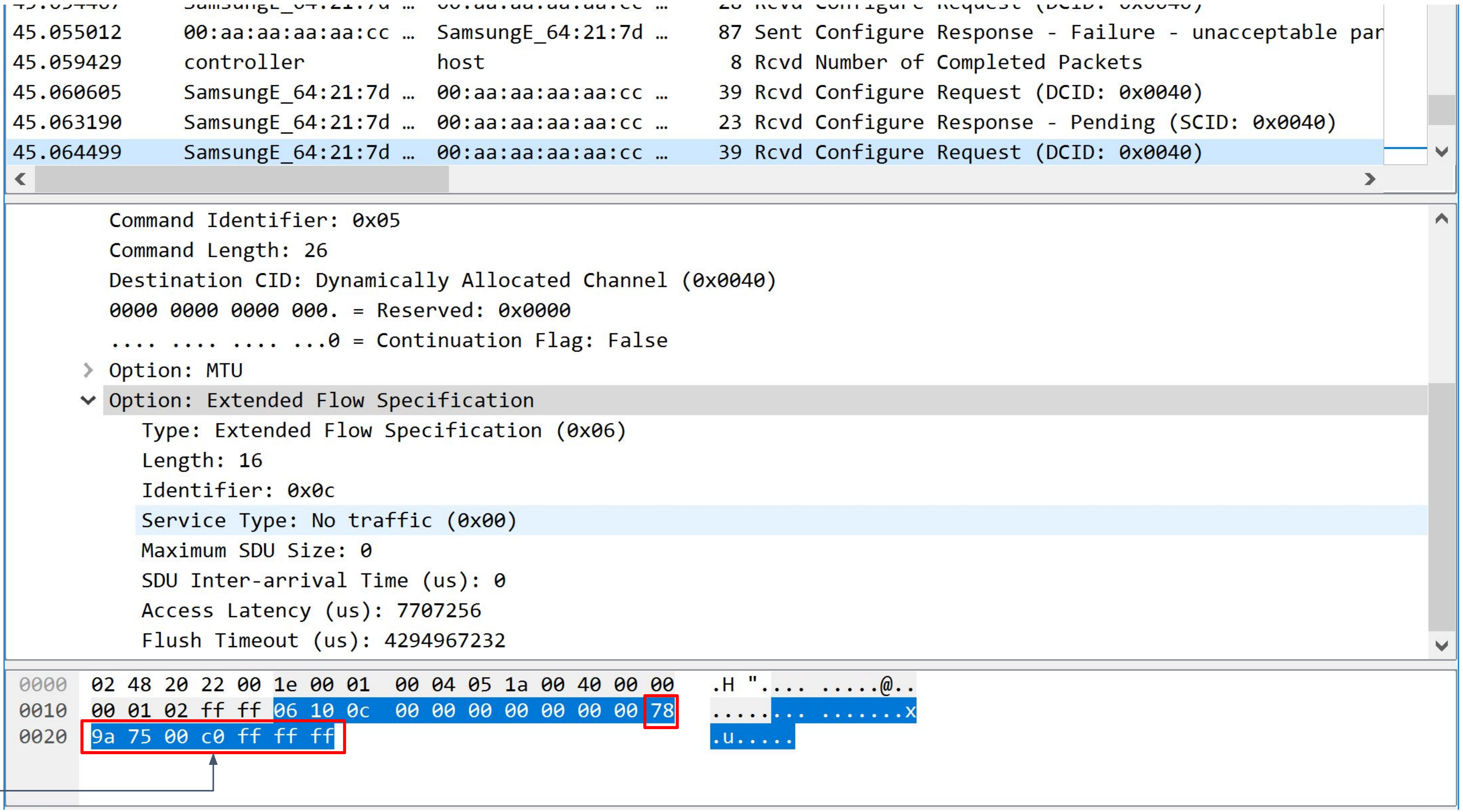
- On major Linux distros, kernel stack canaries are enabled. Some enable KASLR. However:

```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                                void *data, u16 *result) {
  ...
  struct l2cap_conf_efs efs;  // <- Uninitialized
  ...
  while (len >= L2CAP_CONF_OPT_SIZE) {
   ...
     case L2CAP_CONF_EFS:
          if (olen == sizeof(efs))
              memcpy(&efs, (void *)val, olen);
          ...
          l2cap_add_conf_opt(&ptr, L2CAP_CONF_EFS, sizeof(efs),
                                          (unsigned long) &efs);
```

olen is attacker controlled, this memcpy can be avoided

Uninitialized *efs* (16 bytes from stack) will be leaked to attacker

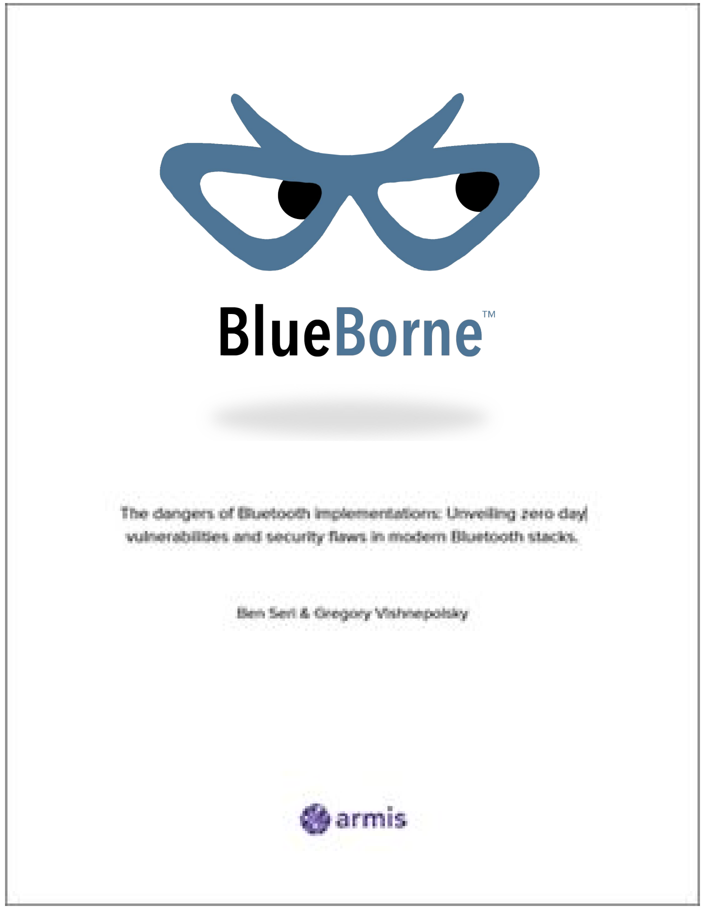
Defeating modern mitigations [2]



pointer from the stack

BlueBorne References

- BlueBorne Linux Exploit (https://github.com/ArmisSecurity)
- BlueBorne Linux Exploit Blog (https://armis.com/armis-labs)
- BlueBorne Technical White Paper (https://armis.com/blueborne)

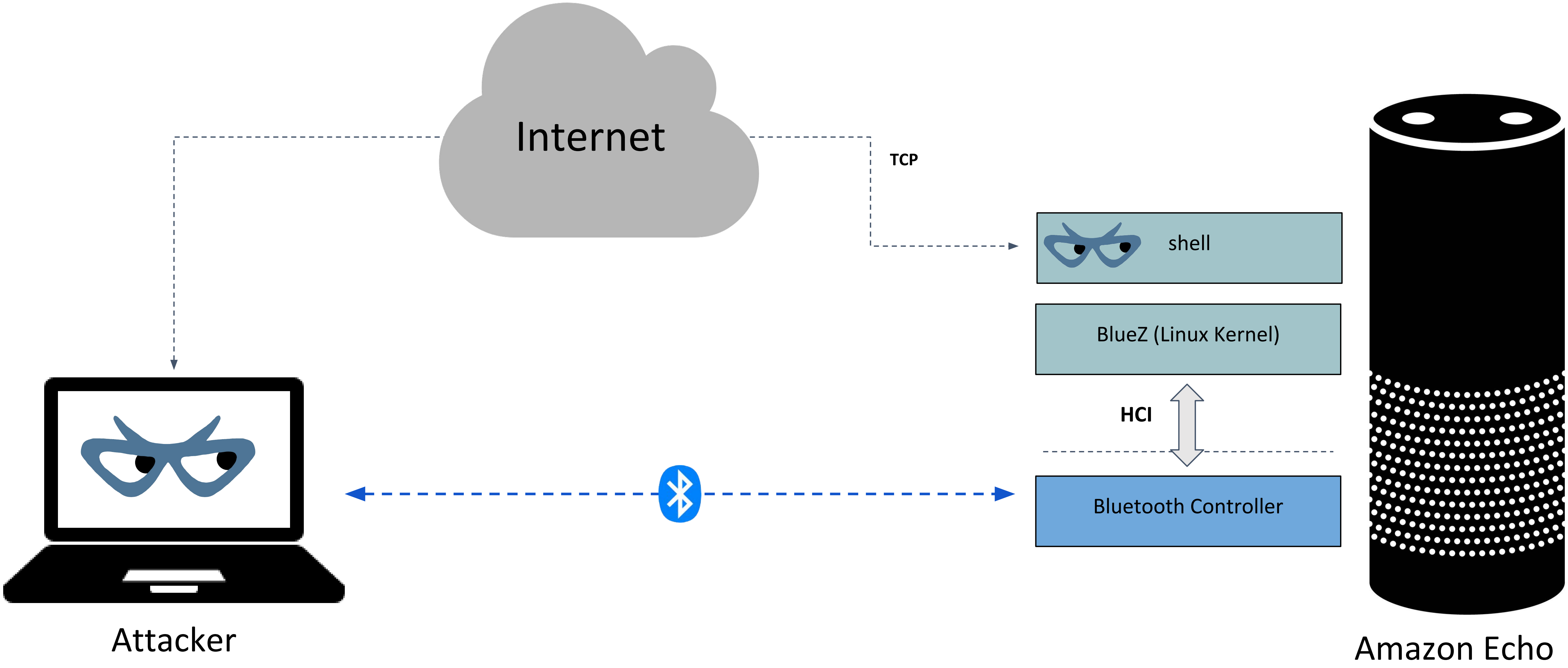

BlueBorne™

The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks.

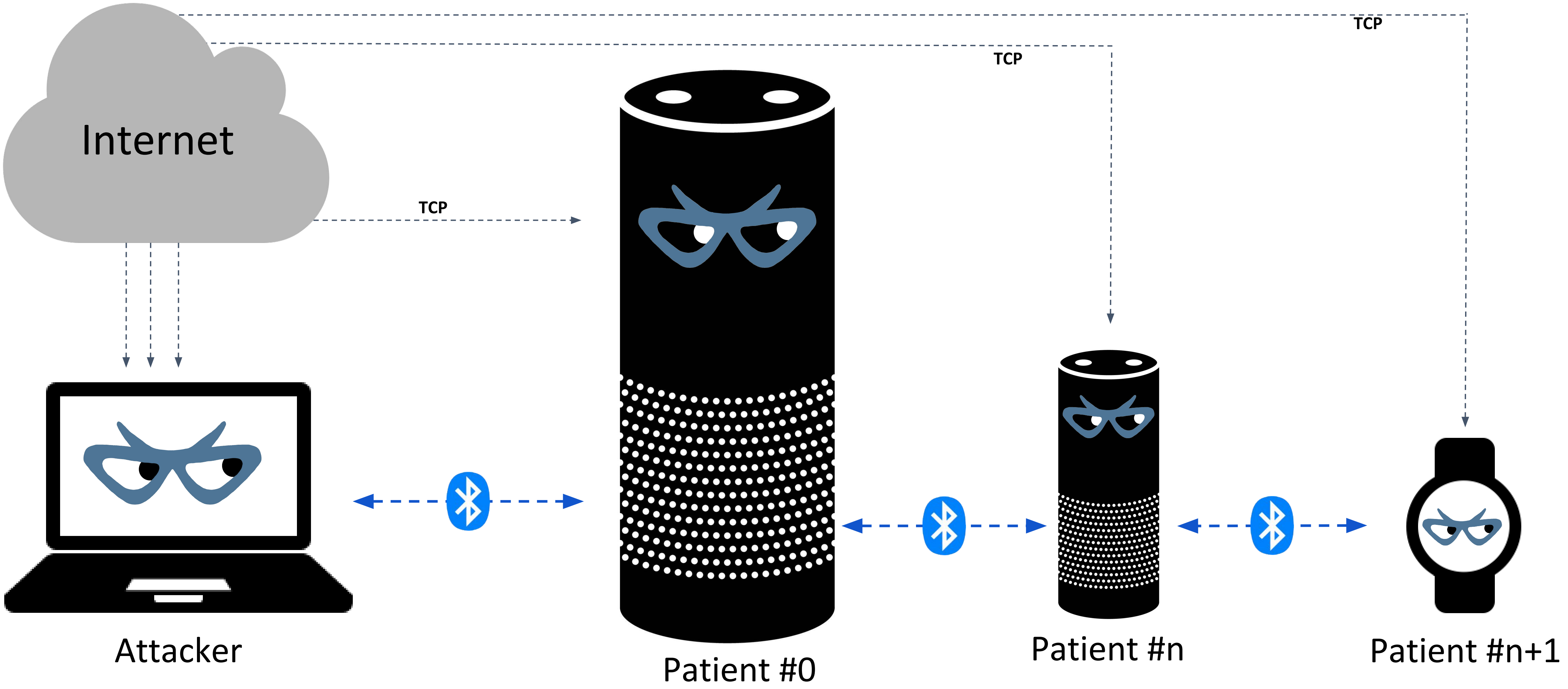
Ben Seri & Gregory Vishnepolsky

armis

Demo


Internet
TCP
shell
BlueZ (Linux Kernel)
HCI
Bluetooth Controller
Attacker
Amazon Echo

Spreading the attack



Key Takeaways

- Bluetooth implementations are complex and underexamined
- Mitigations in Linux devices (especially IoT) are lagging behind
- Security mechanisms should be monitoring Bluetooth, and other wireless protocols as well

QUESTIONS