

# REVERSE ENGINEERING AND EXPLOITING THE BT HOMEHUB 3.0B

Zachary Cutlip, Tactical Network Solutions, LLC  
zcutlip@tacnetsol.com

## ABSTRACT

This paper details the process by which I reverse engineered the firmware for the British Telecom HomeHub 3.0b and remotely exploited a vulnerability that yielded root access. The BT HomeHub 3.0b posed a greater challenge to reverse engineering and exploitation compared to many SOHO routers on the market today. This paper details the various strategies I pursued in search of an exploitable 0-day. Although some strategies were fruitful and some not, all were instructive.

*"Failure is instructive. The person who really thinks learns quite as much from his failures as from his successes."*

--John Dewey

## INTRODUCTION

SOHO wifi routers typically represent soft targets when it comes to vulnerability analysis, discovery, and exploitation. To say that low-hanging fruit is abundant is an understatement. Generally one can obtain a firmware update file from the vendor, unpack it, and within hours discover one or more common classes of vulnerability. With the right tools and skills it is even possible to develop a working, reliable exploit without ever possessing an instance of the target hardware.

A user<sup>1</sup> on the [www.kitz.co.uk](http://www.kitz.co.uk) contacted me directly to request help jailbreaking the British Telecom HomeHub 3.0b, and I accepted the challenge. Our motivation was to unlock configuration capability beyond the basic set of options allowed by BT. A particularly onerous restriction is the HomeHub only works with BT's broadband service.

I assumed I would find an easily exploited command injection vulnerability. I was wrong.

Our (the other forum participants and I) goal was to gain an easy root shell, authenticated or unauthenticated. We needed low level access to the device in order to develop a permanent modification for it. It turned out the classes of bugs I generally look for weren't readily apparent in this device. I took a breadth-first approach to analyzing this device's firmware. I started by analyzing the most obvious attack surface, the web interface, and then moved on to other aspects of the system. I did finally develop a working exploit. Before succeeding, I tried a variety of approaches that yielded little or no fruit. Everything I tried, though, resulted in valuable lessons learned.

## TARGET DEVICE

The subject of this paper's research is the BT HomeHub 3.0b (for brevity, HH3b). This is an ISP-issued customer premise equipment (CPE) purpose-built for BT's consumer broadband service.

---

<sup>1</sup> I am grateful to [kitz.co.uk](http://www.kitz.co.uk) forum user William Kirby for inviting me to help with this project and for sending me hardware to research.



Front of the BT HomeHub 3.0b



Back of the BT HomeHub 3.0b. Ports, from left to right: ADSL, BT broadband, four LAN ports, USB, power, reset button.

The HHH3b has the following technical details<sup>2</sup>:

CPU	Broadcom BCM6361 System-on-a-Chip Dual-core 400 MHz MIPS32 <sup>3</sup>
RAM	Hynix H5PS5162FFR-S6C, 64 MB DDR2-800 SDRAM <sup>4</sup>
Flash Storage	32 MB NAND Flash, TSOP48 Package <sup>5</sup>

<sup>2</sup> <http://forum.kitz.co.uk/index.php/topic,10161.msg213299.html#msg213299>

<sup>3</sup> [http://www.neufbox4.org/wiki/index.php?title=Neufbox\\_6#SoC\\_BCM6361](http://www.neufbox4.org/wiki/index.php?title=Neufbox_6#SoC_BCM6361)

<sup>4</sup> [http://www.hynix.com/inc/pdfDownload.jsp?path=/datasheet/pdf/graphics/H5PS5162FFR\(Rev.1.4\).pdf](http://www.hynix.com/inc/pdfDownload.jsp?path=/datasheet/pdf/graphics/H5PS5162FFR(Rev.1.4).pdf)

The HH3b has an ADSL jack as well as an Ethernet jack for connecting to the "BT Infinity" service. In addition, it has a USB port although there is no functionality exposed that uses this port<sup>6</sup>.

The HH3b is configured via a web interface, similar to many other SOHO routers. However, unlike many devices from Linksys, Netgear, and even Verizon, the HH3b exposes comparatively few configuration options to the user.

The screenshot shows the BT Home Hub web interface. At the top, there's a header with "BT Home Hub" and "Help | A-Z". Below this is a navigation bar with tabs: "Home", "Services", "Settings" (selected), and "Troubleshooting". Under "Settings", there are sub-tabs: "Wireless", "Broadband" (selected), "Home Network", "Port Forwarding", "System", and "Basic Settings". Below these, there's a row of links: "Internet" (selected), "Connection", "VPN", and "Dynamic DNS".

The main content area is titled "Internet Connection Configuration". It has a section "Connection information" with a table:

Connection time:	Disconnected
Data transmitted/received (GB):	Not available
Broadband username:	<input type="text" value="bthomehub@btbroadband.com"/>
Password:	<input type="password"/>

Below the password field is a "Show Characters" button.

There's another section "TCP/IP settings" with a table:

Broadband network IP address:	Not available
Default gateway:	Not available
Primary DNS:	Not available
Secondary DNS:	Not available

At the bottom right of the configuration area is a "Connect" button.

Relatively limited configuration options. For example there's no WAN configuration, only BT's PPPoE.

Of particular interest is the fact that the HH3b will only connect to broadband via PPPoE. Further, it will only accept PPPoE credentials issued to the customer by BT. This effectively ties the device to BT's service.

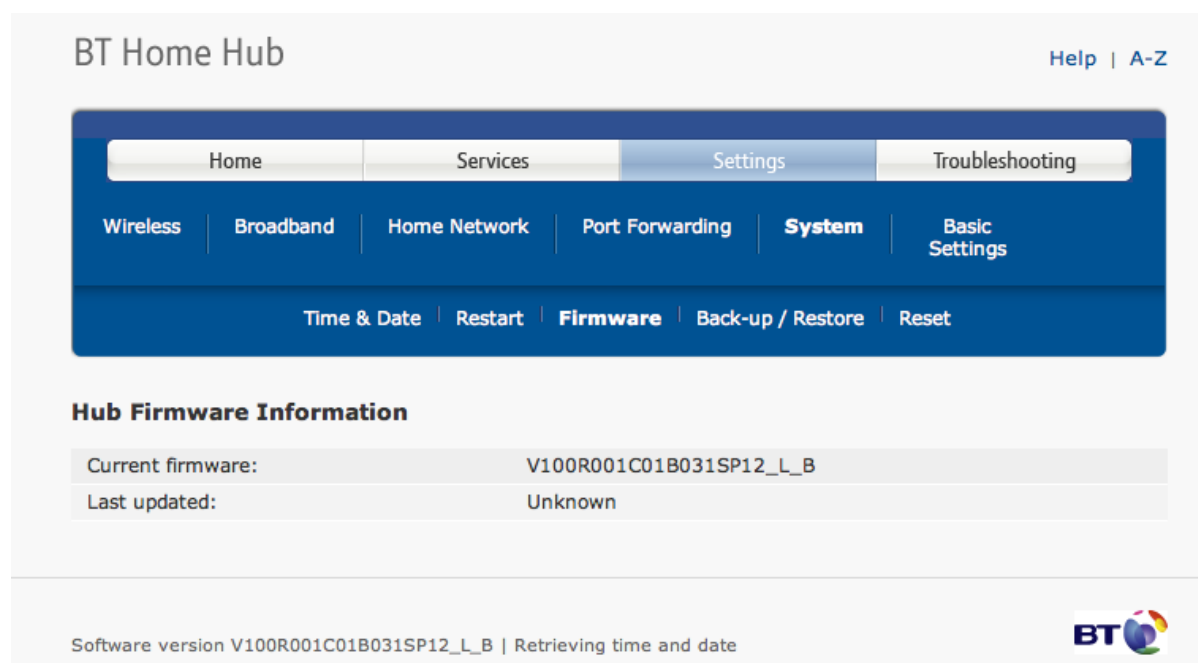
<sup>5</sup> [http://pdf1.alldatasheet.com/datasheet-pdf/view/94408/STMICROELECTRONICS/NAND256W3A2BN6/+7\\_4Q9UORIHdyRHOIpa/1XXyxeocP+uKxP6OXpaoV+/datasheet.pdf](http://pdf1.alldatasheet.com/datasheet-pdf/view/94408/STMICROELECTRONICS/NAND256W3A2BN6/+7_4Q9UORIHdyRHOIpa/1XXyxeocP+uKxP6OXpaoV+/datasheet.pdf)

<sup>6</sup> The USB port is useful during post-exploitation. The system mounts USB disks automatically, making it easy to load a debugger or other tools onto the device.

Analysis of the HH3b is frustrated by the fact that there is no obvious way to interact with and analyze the device other than the limited web interface. There do not appear to be any listening services that will yield an interactive console. Further, there is no obvious hardware connection for UART or JTAG. In order to analyze and interact with the device, it is necessary to find and exploit a vulnerability and gain a root shell.

## FIRMWARE CHALLENGES

The first challenge this project presented was obtaining a firmware image for the target device. The HH3b differs from many other SOHO devices in that its firmware updates are pushed from the vendor, in this case the ISP: BT. There is no public facing web or ftp site from which a user may download a firmware update file or facility in the device's web interface with which to upload a firmware update file. There isn't even the ability for the user to prevent the device from being updated, or to command it to check for updates. The only thing visible to the end user regarding firmware is the version string, and a status message showing the time of the last firmware update.



The screenshot displays the BT Home Hub web interface. At the top, the title "BT Home Hub" is on the left, and "Help | A-Z" is on the right. Below the title is a navigation bar with four tabs: "Home", "Services", "Settings" (which is selected), and "Troubleshooting". Under the "Settings" tab, there is a sub-menu with "Wireless", "Broadband", "Home Network", "Port Forwarding", "System" (which is selected), and "Basic Settings". Below this sub-menu is another row of links: "Time & Date", "Restart", "Firmware" (which is selected), "Back-up / Restore", and "Reset". The main content area is titled "Hub Firmware Information" and contains a table with two rows: "Current firmware:" with the value "V100R001C01B031SP12\_L\_B" and "Last updated:" with the value "Unknown". At the bottom of the page, there is a footer that says "Software version V100R001C01B031SP12\_L\_B | Retrieving time and date" and the BT logo on the right.

Hub Firmware Information	
Current firmware:	V100R001C01B031SP12_L_B
Last updated:	Unknown

Firmware update page only shows firmware version and time of last update.

We need to dynamically or statically analyze the device's firmware in order to find a vulnerability and gain a root console. A root console is necessary in order to interact with and analyze the device. The circular dependency is obvious. Before I joined the project, a researcher by the name of 'asbokid' on the kitz.co.uk forum was able to gain a critical, initial beachhead. Asbokid removed the HH3b's flash storage chip, attached it to a flash reader and dumped its contents. The contents include the bootloader and several JFFS2 file systems, including the device's root file system.



**asbokid**  
 Reg Member  
 ☆☆☆  
 Posts: 819

**Re: BT Home Hub 3.0 - Type B**  
 « Reply #31 on: July 22, 2012, 03:55:07 AM »

Quote

A full 32MByte NAND flash dump from a **BT Home Hub 3.0b** firmware version **V100R001C01B031SP09\_L\_B\_t2011-06-01\_22\_39** is linked below [1]

The HOWTO in the *Google Docs* folder illustrates mounting and extracting the file systems from that NAND flash dump.

The flash dump contains two root file system images. They are identical JFFS2 images, the master and the slave. Both images contain a MIPS32 kernel, and the CFE bootloader (cferam.000).

There are also two smaller JFFS2 file systems in the dump, and the pre-CFE bootstrap code (times two), as well as the NVRAM area found at the end of the flash.

cheers, a

[1] <https://docs.google.com/folder/d/0B6wW18mYskvBMmNQTIhDeG5vT2c/edit>

« Last Edit: August 04, 2012, 02:24:24 AM by asbokid »

Report to moderator Logged

Kitz.co.uk forum user 'asbokid' post the firmware he was able to dump.

With the root file system extracted, we could begin analysis in earnest.

## COMMAND INJECTION

The first strategy I pursued was to locate a command injection vulnerability. This generally takes the form of a command that is generated and passed to the `system()` or `popen()` library calls. A format string plus one or more parameters is passed to `sprintf()` to generate the command string, which is then executed in a shell. If one of the additional parameters comes from user input and isn't properly sanitized, the user can provide invalid input that terminates the shell command and starts a new one.

The following is an example on a Trendnet device of such a vulnerability:

```
00415864 la      $a1, 0x420000
00415868 la      $t9, sprintf
0041586C addiu   $a1, (aPingClss - 0x420000) # "ping -c 1 %s > %s"
00415870 move   $a2, $s3
00415874 addiu   $a3, $s2, (aTmpPing_result - 0x420000) # "/tmp/ping_result.log"
00415878 jalr    $t9, sprintf
0041587C move   $a0, $s0
00415880 lw      $gp, 0x180+var_170($sp)
00415884 move   $a0, $s0
00415888 la      $t9, system
0041588C nop
00415890 jalr    $t9, system
```

An common `sprintf/system` pattern for command injection vulnerabilities.

In this case a user provided IP address is passed to `sprintf()` to generate a ping command. A user can provide input such that the following command is generated and executed:

```
ping -c 1 192.168.0.1;/sbin/telnetd& >/tmp/ping_result.log
```

This has the effect of pinging the provided IP address AND starting up the system's telnet server.

An easy way to discover such a vulnerability is to analyze strings in the system executables and libraries that appear to be printf-style format strings that contain shell redirect to `/dev/null` or some other file, such as `>/dev/null`. The above vulnerability contains `"ping -c 1 %s >/dev/null"` format string.

By performing cursory strings analysis using the `'strings'` and `'grep'` commands on the various system binaries, we find the `'cms'` executable which appears to be a rich target.

```
bthh3.0b-rootfs/bin (0) $ strings cms | grep 'dev\|/null' | grep '%s' | wc -l
193
```

There are lots of system commands in this executable that appear to get generated at run-time. It is not obvious which, if any, of these are generated from user input, and which, if any, fail to sanitize the user input. However, there are some promising leads. Most of the command strings in `'cms'` are iptables commands. A `grep` one-liner finds all format strings with shell redirects involving the iptables command:

```
$ strings cms | grep 'dev\|/null' | grep '%s' | grep iptables | wc -l
141
```

Some of these are likely from port forwarding rules input by the user through the web interface. Strings analysis on the web executable, which is the HTTP server, shows a reference to the `cms` executable.

Here is one iptables command that appears too good to be true:

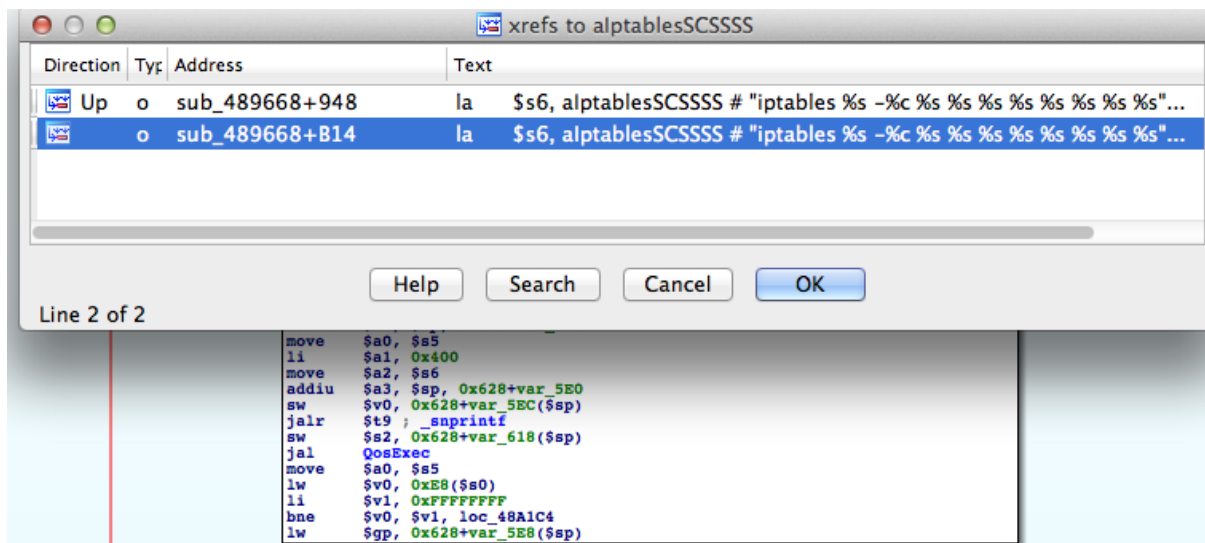
```
iptables %s -%c %s %s %s %s %s %s %s %s %s %s %s 2>/dev/null
```

The above command is generated from, among other things, nine input strings (indicated by the `%s` format codes). Surely at least one is from user input.

Sadly, some static analysis led to heartbreak, foul language, and the flipping over and setting fire to a coworker's car.

When we disassemble the `cms` executable, and look for cross-references to the above string (IDA gives this string a name: `'aIptablesSCSSSS'`), we find two. The function `sub_489688` references this string twice.



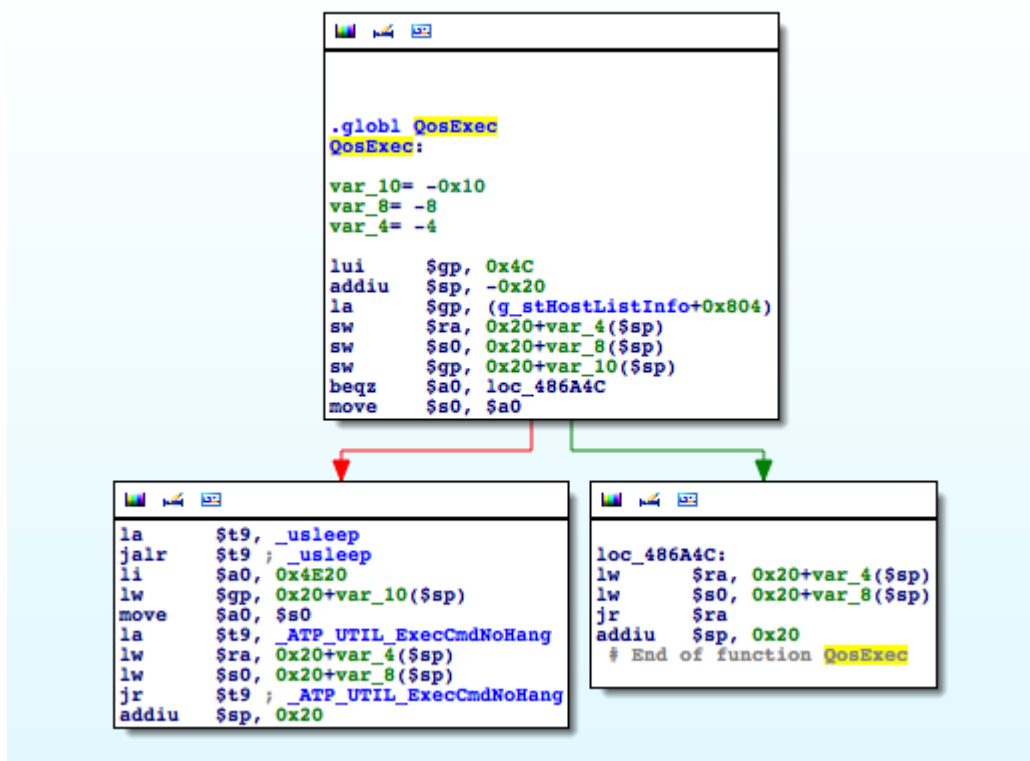


Only two direct references the 'alptablesSCSSSS' string.

In both cases, we're in for a trail of disappointment for the same two reasons. First, this format string is generated by the comparatively safe `snprintf()`. Assuming the programmer has provided the correct size of the destination buffer (or smaller), there is no opportunity for a buffer overflow from this bounds-checking function. A lack of a buffer overflow isn't the biggest issue; we're actually on the hunt for command injection, not stack smashing.

The real problem lies not in how the command is generated but how it gets executed. One of the reasons this and other format strings are so promising is their use of Bourne Shell style redirects such as `'2>/dev/null'` and the like. These appear to be commands that get passed as arguments to an invocation of `"/bin/sh -c"`. The most common two library functions that execute a command through an invocation of the shell are `system()` and `popen()`. It turns out that's not the only way to execute a command string.

Disassembly shows that the generated iptables command is passed as an argument to `QosExec()`. That function, in turn, passes the command string through to `ATP_Util_ExecCmdNoHang()`. Still no `system()` or `popen()` but no dead-end, either.



QosExec() is passed a command string which it passes into ATP\_UTIL\_ExecCmdNoHang().

We can disassemble libatputil.so to study how ATP\_Util\_ExecCmdNoHang() executes commands. To make a long story less long, it does the following:

- Parses the command string it is given into an array
- Opens whatever file is referenced by the shell redirect (e.g., >/dev/null) for writing (or reading)
- fork()
- dup2() the open file to whatever file descriptor is being redirected
- execv()'s the command
- parent process waits on child to terminate.
- 

Based on my analysis, all external commands are executed through this function or one of its variants. One interesting variant of this function differs from the others in that it does pass the command string as an argument to "sh -c" (not via system(), but rather fork() and exec()). This variant, called ATP\_Util\_ExecCmd() would be an ideal target for command injection, as it does not parse or, in any way, sanitize the command passed to it. However, my analysis to date has revealed only limited use of this function, none of which receives untrusted input.

There appears to be no obvious opportunity for command injection.

## EASY BUFFER OVERFLOWS

Short of an easy win via command injection, the most desirable (from an exploitation perspective) vulnerability is a stack-based buffer overflow resulting from the use of an unbounded string-handling function. Referring back to the above example where a ping command is generated from user input, the actual command string is generated using the unbounded `sprintf()` function. The destination buffer is allocated on the stack using a constant size. This combination of unsanitized user input, fixed size, stack-based allocation, and the use of unsafe string functions is a common pattern in embedded systems. This is despite the wide recognition, acceptance, and endorsement of safer patterns combining bounds checking functions with dynamic allocation of memory. Exploitation of such a bug was documented in Aleph One's seminal 1996 paper, *Smashing the Stack for Fun and Profit*<sup>7</sup>. In the years since, this knowledge has become canon.

A buffer overflow vulnerability can be both desirable and easy to exploit, particularly on embedded systems that lack the protection mechanisms of modern desktop and server operating systems. An exploitable buffer overflow can be more powerful than command injection since arbitrary code may be executed, rather than just code that already exists on the target.

"Safe" functions such as `memcpy()` and `strncpy()` may be exploitable too, but this requires the programmer to incorrectly specify the size of the destination buffer. Often, the miscalculation may only be off by a few bytes, drastically reducing the amount of potential memory corruption. Further, discovery of this class of vulnerability requires more work on the part of the researcher. Given my breadth-first approach, these more subtle bugs would have to wait until later. An easy win remained my hope and goal.

One way to begin the search for buffer overflows is simple strings analysis on the target's libraries and executables. References to functions linked from external libraries will exist as strings in the target binary.

Analysis of the 'web' executable, the program responsible for the HH3b's web interface reveals few uses of unsafe string handling functions:

```
bthh3.0b-rootfs (0) $ strings bin/web | grep strcpy | wc -l
```

```
0
```

```
bthh3.0b-rootfs (0) $ strings bin/web | grep sprintf | wc -l
```

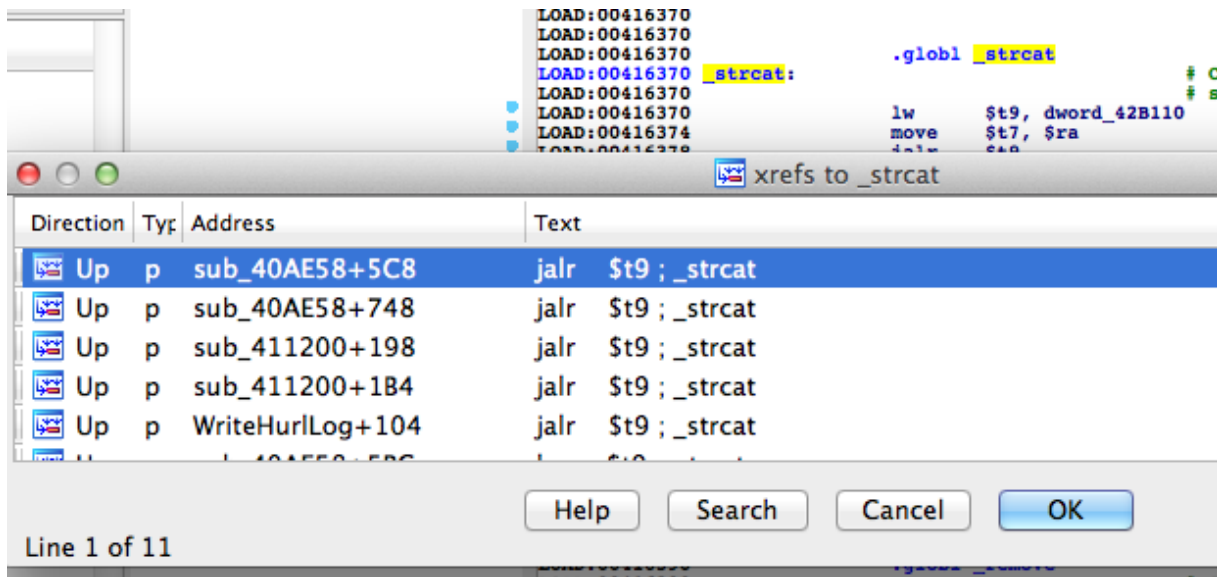
```
0
```

While there are no references to `strcpy()` or `sprintf()`. There is a reference to the unsafe `strcat()`.

Disassembly reveals five cross-references to this function.

---

<sup>7</sup> <http://www.phrack.org/issues.html?issue=49&id=14#article>



Only five cross references to strcat() will be easy to analyze.

With only five uses of this function, it is relatively easy to run each of them to ground, identifying whether they are handling unsanitized user input. Each instance takes as input a string constant, or verifies the length of the input string is within safe limits. There appears to be no low-hanging buffer overflow opportunities in the web executable. This is unfortunate. The web interface is the most user-facing of the components running on the device, and, as such, presents the most attractive attack surface. With few exceptions, analysis of all the executables and libraries yield similar results. I'll discuss the exceptions later in this paper. There may well be opportunities to overflow a buffer, gaining control of execution, but they'll likely be more subtle bugs--if they exist.

## CONFIGURATION FILE

A device's configuration file can open some interesting attack vectors. Many embedded devices such as SOHO routers provide a means of backing up their configuration, allowing the user to restore the backup later. This makes it possible to replace the device in the event of hardware failure while saving the user the hassle of manual configuration.

Generally, these configuration backup files are obscured in some way to resist tampering. The system may place unwarranted trust in the configuration backup file, assuming that it remains unchanged from what was generated by trusted code. Tampering with a configuration file may be an opportunity to violate such assumptions.

The code that parses a supplied configuration file may be vulnerable to memory corruption bugs if it assumes well-formed data. Further, if the configuration file is parsed by a Bourne Shell script, there will almost certainly be the opportunity for command injection.

There may also be configuration options that, while not exposed in the user interface, are available in the configuration grammar. Such hidden configuration options may enable an undocumented backdoor account or enable a secondary administrative service such as telnet or SSH.

Sometimes the obfuscation is a custom compression or encryption mechanism. Other systems may use standard and well-vetted encryption mechanisms, and may even cryptographically sign the backup file.

Custom developed mechanisms can be easy to defeat, since they often are not mindful of cryptography principles. Sometimes their algorithms may be reverse engineered and re-implemented. Alternatively, with the help of a cross-compiler and the QEMU emulator, the necessary libraries from the target's firmware can be linked, and appropriate decryption functions called.

More robust implementations that use standardized encryption mechanisms may use per-device or per-user encryption and decryption keys<sup>8</sup>. If the BT HomeHub 3 uses unique keys for encryption and decryption, then the firmware extracted from one device likely won't provide sufficient information to decrypt another device's configuration file.

```
sub_681C:
var_158= -0x158
var_154= -0x154
var_150= -0x150
var_148= -0x148
var_138= -0x138
var_20= -0x20
var_14= -0x14
var_10= -0x10
var_C= -0xC
var_8= -8
var_4= -4
arg_10= 0x10
arg_14= 0x14

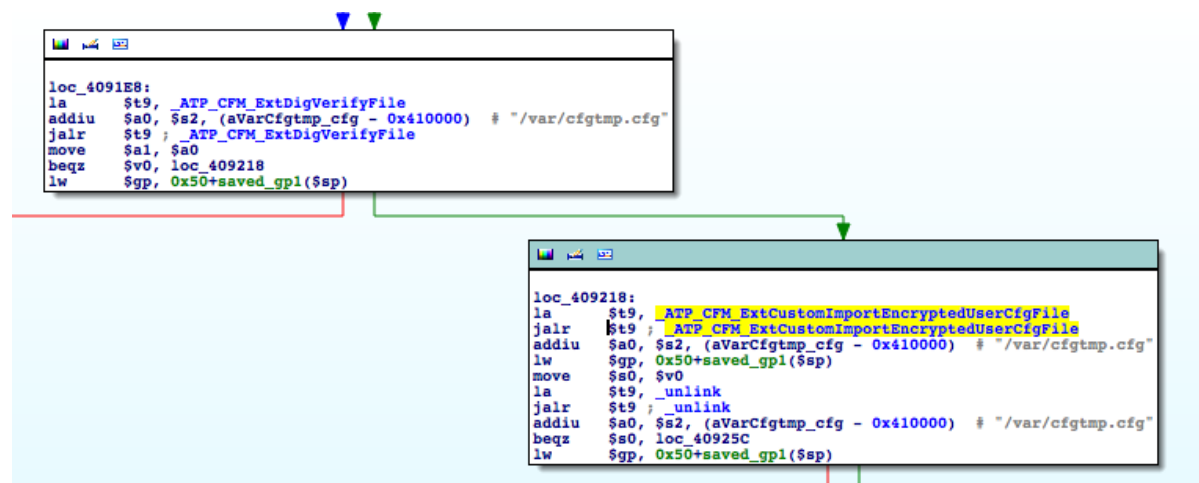
la      $gp, (data_buffer+0xFC4)
addu    $gp, $t9
addiu   $sp, -0x168
sw      $ra, 0x168+var_4($sp)
sw      $s3, 0x168+var_8($sp)
sw      $s2, 0x168+var_C($sp)
sw      $s1, 0x168+var_10($sp)
sw      $s0, 0x168+var_14($sp)
sw      $gp, 0x168+var_150($sp)
addiu   $s0, $sp, 0x168+var_148
la      $t9, _memcpy
sw      $a3, 0x168+var_20($sp)
move    $s2, $a0
move    $s1, $a1
move    $a0, $s0
move    $a1, $a2
jalr    $t9, _memcpy
li      $a2, 0x10
lw      $gp, 0x168+var_150($sp)
addiu   $s3, $sp, 0x168+var_138
la      $t9, _aes_setkey_dec
move    $a1, $s2
sll     $a2, $s1, 3
jalr    $t9, _aes_setkey_dec
move    $a0, $s3
lw      $gp, 0x168+var_150($sp)
lw      $a3, 0x168+var_20($sp)
lw      $v0, 0x168+arg_14($sp)
la      $t9, _aes_crypt_cbc
...
```

HH3b: This isn't the crypto you're looking for. Move along.

<sup>8</sup> Unique device keys would likely defeat the advantage of backing up a configuration from one device and restoring to another.

Static analysis revealed the use of AES and open source SSL functions, so I assumed the BT HH3b used an encryption scheme that would not be trivially defeated. Before moving on, though, I realized an easy test would be to back up from one device and restore to another. Restoration was successful and was a compelling sign that the firmware on any device can decrypt the backup file from any other device, given compatible firmware versions on the two devices.

Disassembly of the web server revealed how the backup file is unpacked and restored.



The config file's cryptographic signature is verified and stripped before the file is decrypted and restored.

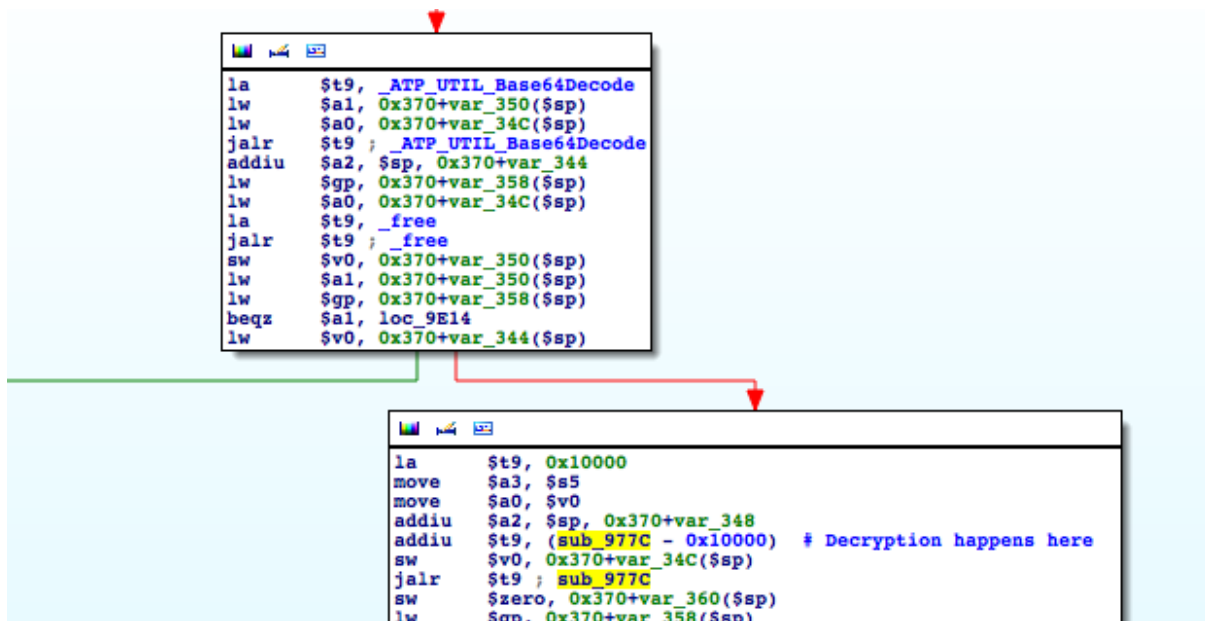
The backup file has a 128-byte cryptographic signature followed by base64-encoded data. The data is base64 decoded, AES decrypted, then decompressed using the standard Unix/Linux 'gunzip' command<sup>9</sup>. The resulting output is an XML-formatted text file.

We can easily decrypt the backup file using appropriate library functions from the target's firmware. This entails writing a simple C program that calls the verbosely named 'ATP\_CFM\_ExtCustomImportEncryptedUserCfgFile()' function. See Appendix A for this program's source code listing.

Reverse engineering this function reveals a straightforward API: it takes a single argument, a pointer to the name of the file to be decrypted. Here is a function signature:

```
ATP_CFM_ExtCustomImportEncryptedUserCfgFile(const char *file_to_decrypt);
```

<sup>9</sup> This is one of the few places that a command string is dynamically generated, then executed by 'sh -c'. Since it is generated from string constants and no user data, however, there is no opportunity for command injection.



Disassembly of ATP\_CFM\_ExtCustomImportEncryptedUserCfgFile().

The configuration backup is base64-decoded, then decrypted.

The 128-byte signature is removed in the signature checking stage. For our analysis, the actual verification of the file's integrity is not important, so we can skip that step and manually strip the signature. Using QEMU to run our test program on this backup file, sans signature, results in the decrypted XML found in Appendix C.

The first attempts to run the decrypting program in QEMU are telling, to the point of amusement. Interesting hints emerge as to how the system's cryptographic keys are stored.

Running the program, which dynamically links `libcmap_i.so`, yields errors from the dynamic linker. Our program is looking for oddly named functions such as `'PZMM_K_Fun2()'`. Disassembly of `libcmap_i.so`'s functions involved with encryption and decryption reveals references to many more functions with similar names, such as `PZMM_K_Fun3()` and `PZMM_K_Fun4()`, as well as `PZMM_I_Fun1-4`.

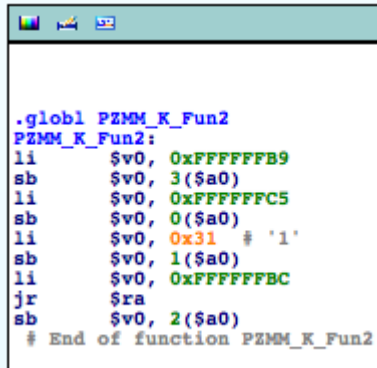
Direction	Type	Address	Text
Up	p	call_PZMM_K_Fun+50	jalr \$t9; _PZMM_K_Fun2
Up	o	call_PZMM_K_Fun+34	la \$t9, _PZMM_K_Fun2
D...	o	LOAD:00038B44	.word _PZMM_K_Fun2

A single call to PZMM\_K\_Fun2().



These functions are found in several other libraries, such as `libbhalapi.so` and `libsgapi.so`. This is unusual for a couple of reasons. First, exported functions from most of the libraries on this system conform to a predictable namespace and are organized based on purpose. `ATP_CFM_ProcessFile()`, for example, is found in `libcfdmapi.so`. `ATP_UTIL_ExecCmd()` is found in `libatputil.so`, and so forth. Second, `libcfdmapi.so` doesn't explicitly link the libraries it needs. Rather, it assumes some other library or executable earlier in the dependency chain has already linked them.

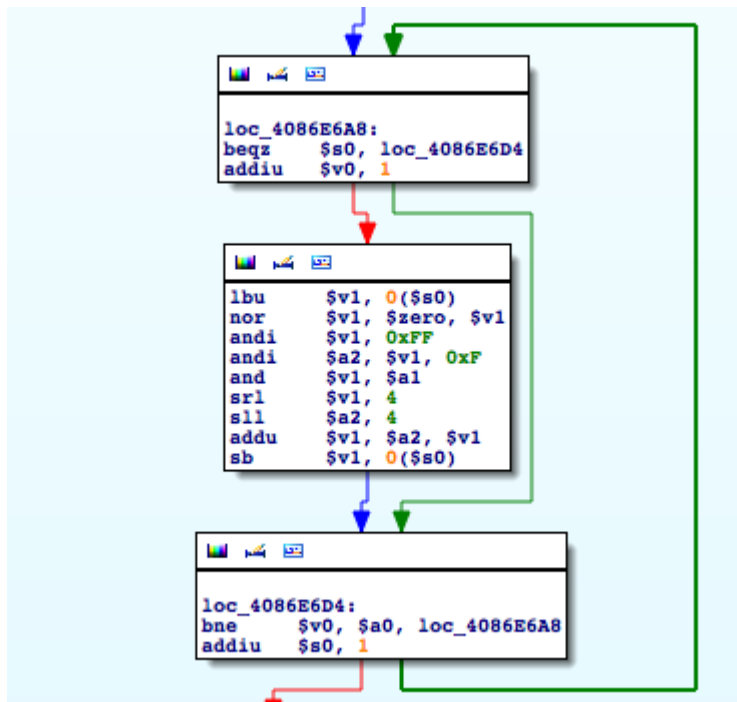
As it turns out the `PZMM_K_Fun*` functions assemble an AES key by loading byte constants into memory.

A screenshot of a text editor window showing assembly code for a function named `PZMM_K_Fun2`. The code is written in MIPS assembly and includes several instructions: `li $v0, 0xFFFFFFFFB9`, `sb $v0, 3($a0)`, `li $v0, 0xFFFFFFFFC5`, `sb $v0, 0($a0)`, `li $v0, 0x31 # '1'`, `sb $v0, 1($a0)`, `li $v0, 0xFFFFFFFFBC`, `jr $ra`, and `sb $v0, 2($a0)`. The function ends with a comment `# End of function PZMM_K_Fun2`. The window has a standard Linux-style title bar with icons for file operations and a terminal icon.

```
.globl PZMM_K_Fun2
PZMM_K_Fun2:
li    $v0, 0xFFFFFFFFB9
sb    $v0, 3($a0)
li    $v0, 0xFFFFFFFFC5
sb    $v0, 0($a0)
li    $v0, 0x31 # '1'
sb    $v0, 1($a0)
li    $v0, 0xFFFFFFFFBC
jr    $ra
sb    $v0, 2($a0)
# End of function PZMM_K_Fun2
```

`PZMM_K_Fun2()` loads a constant, 4-byte value into the buffer referenced by a pointer passed to it. Similar functions load twelve more bytes.

After assembling the 16-byte key buffer, it is de-obfuscated with a simple algorithm that, for each byte, performs a bitwise negation, and then swaps the low and high four bits.



Deobfuscating the AES key after it has been assembled.

The series of PZMM\_I\_Fun\* functions similarly assemble a (constant!) AES initialization vector.

In order to decrypt the file it is simply a matter of using LD\_PRELOAD to ensure the libraries containing the obfuscated key-assembling functions are loaded. Further, the key and IV may be trivially obtained from either static analysis or by extracting them using a debugger.

A program calling the appropriate functions in the PolarSSL libraries may use the following key and IV declarations to decrypt the BT HomeHub 3.0b's encrypted configuration file:

```

unsigned char AES_DECRYPTION_KEY[]=

    {0x66,0x50,0x0a,0x68,0xa3,0xec,0x34,0x64

    0xb2,0x2c,0x28,0xf4,0xa6,0xee,0xa8,0x54};

unsigned char AES_IV[]=

    {0x54,0x7c,0x17,0xf2,0x23,0x93,0x94,0x23

    0x16,0x7e,0x7b,    0x09,0x4b,0xbc,0x17,0x0b};
  
```

The process of re-encrypting and re-signing of the configuration file is similar to decrypting. A short program written in C that links to libcfmapi.so and calls the right functions does the trick. See Appendix B for an example listing.

Perusing the decrypted backup configuration file, we don't find any settings that aren't exposed in the web interface<sup>10</sup>.

Perhaps there are optional XML elements that aren't present in this backup config, but would be restored if they were present. Doing some strings analysis dashes these hopes. The libcfmapi.so library appears to be the only library that references the <BackupFile> XML element. Strings analysis on this file reveals no XML elements that aren't also in the backup configuration file.

Before giving up on the configuration file as an attack vector, it's worth analyzing the system's default configuration file, '/etc/defaultcfg.xml'. This file's composition is similar to the configuration backup file and can be decrypted and re-encrypted in the same way<sup>11</sup>.

The decrypted default configuration reveals much more interesting configuration settings, such as remote administration credentials (presumably for use by the ISP) as well as the option to enable or disable a command-line interface. Additionally there are options to configure the LAN and WAN interfaces, including whether to use ADSL or Ethernet, and whether to use PPPoE.

Unfortunately this file appears to use a different XML namespace than the configuration backup file. Attempts to modify this file and restore it in lieu of the downloaded backup file do not result in any changes to the system's configuration. It may be possible to modify this file and have the system take it as its default configuration. Root access will be necessary to analyze such attack vectors, so that remains the goal, and that goal remains unmet.

## USERSPACE MIPS EMULATION WITH QEMU

Analysis of one binary, 'bcmupnp', shows some promising leads. This application stands apart from the other executables and libraries on the system for a couple of reasons. First, its function names and linked libraries don't use the same consistent naming discussed above. More importantly, however, there are abundant uses of unsafe string handling functions.

IDA Pro's analysis of this application wasn't particularly clean and many functions were orphaned with no references to them. It is likely that references to these functions are generated at run-time and they are called from a jump table or similar construct. There are a number of soft spots in this application. With no cross-references to those code paths, though, it can be difficult to know how to exercise them and to know if they are handling any untrusted data.

While static analysis alone could probably answer these questions, the opportunity to debug the application will save substantial effort. Clearly, we can't attach a debugger to the live system, so MIPS emulation via QEMU is necessary to run this application.

QEMU has two modes of operation. In computer mode, it boot and run a full operating system and all its individual programs. In user mode it can run a single application within the host operating system.

---

<sup>10</sup> The one semi-useful find is the PPPoE credentials specified in the <WANPPPConnection> element. While the web interface does expose this setting, it won't allow any PPPoE username that doesn't appear to be a valid BT Broadband account. One could manually edit this file, substituting another ISP's login credentials. However the goal of this exercise is to gain root, so this isn't a win.

<sup>11</sup> A decrypted listing of this file can be found at <http://pastebin.com/1GZ3c3dm>

Emulating in QEMU user mode is effortless for simple binaries that make few or no assumptions about the hardware they're running on. QEMU can generally run a device's busybox or CGI binaries with no trouble. The bcmupnp executable is more complicated in that it not only makes assumptions about the hardware that is present, but it is also responsible for managing some of that hardware. It is an application provided by the chipset manufacturer, Broadcom, to manage the system's wireless networking via Universal Plug and Play. As such, its dependencies on a specific physical configuration are considerable. QEMU is unable to satisfy those dependencies. There are tricks, though, that can get this application running.

The first problem encountered while trying to emulate bcmupnp was its queries for system configuration. This problem decomposes into three smaller ones. I would need to intercept the calls to `nvramp_get()` and supply answers that would satisfy the application. I didn't know the set of configuration parameters that would be queried. Even if I did know what would be queried from NVRAM, I could only guess at what would be reasonable values for those parameters. There was no way to inspect the live system's NVRAM for such values.

A solution to the first problem is relatively straightforward. I wrote and cross-compiled a shared library that provided a single function:

```
char *nvramp_get(char *nvramp_key);
```

This shared library can intercepts queries to NVRAM when loaded using the `LD_PRELOAD` environment variable.

To discover what configuration parameters would be queried, I simply had `nvramp_get()` print to standard output the `nvramp_key` string passed to it, and then return a NULL pointer in response. With knowledge of the NVRAM keys that would be queried, I built into the library a static list of what seemed like reasonable responses to the NVRAM queries. This was sufficient to convince the bcmupnp to run and idle in QEMU.

Emulation problems didn't end here though. Strings analysis indicates this application presents itself as a `WFADevice` to UPnP clients. Craig Heffner's miranda<sup>12</sup> UPnP analysis tool was useful in analyzing this application. When connected to the actual HH3b hardware, miranda discovers a `WFADevice` UPnP device in response to M-SEARCH enumeration. However the emulated bcmupnp refused to respond to M-SEARCH discovery requests.

After considerable effort debugging and reverse-engineering bcmupnp, I discovered a global structure, `WFADevice`, that is related to the presence of wireless hardware. If there are no wireless devices (under the application's narrow definition) then bcmupnp does not respond to M-SEARCH queries. Two fields in the `WFADevice` structure are reset to constant values 1 and 3, if there is no hardware to manage. It is unclear what the exact semantics of these particular fields are.

---

<sup>12</sup> <https://code.google.com/p/miranda-upnp/>

```

sub_40FD30:
var_4= -4
arg_0= 0

li      $gp, 0x1BDE0
addu    $gp, $t9
addiu   $sp, -8
sw      $fp, 8+var_4($sp)
move    $fp, $sp
sw      $a0, 8+arg_0($fp)
la      $v0, WFADevice
li      $v1, 1
sw      $v1, (WFADevice+0x2C - 0x421E90)($v0)
la      $v0, WFADevice
li      $v1, 3
sw      $v1, (WFADevice+0x30 - 0x421E90)($v0)
move    $v0, $zero
move    $sp, $fp
lw      $fp, 8+var_4($sp)
addiu   $sp, 8
jr      $ra
nop
# End of function sub_40FD30

```

When there is no wireless hardware to manage, two fields in the WFADevice structure are set to 1 and 3 respectively, which causes M-SEARCH requests to be ignored.

When the WFADevice structure indicates there is no hardware to manage, bcmupnp will not respond to M-SEARCH queries. I patched the function that modifies this structure, such that it leaves the initialized values intact. The bcmupnp application would now run in QEMU and respond to M-SEARCH queries.

```

0040FD30
0040FD30
0040FD30
0040FD30 WFADevice_sub_40FD30:
0040FD30
0040FD30 var_4= -4
0040FD30 arg_0= 0
0040FD30
0040FD30 li      $gp, 0x1BDE0
0040FD38 addu   $gp, $t9
0040FD3C addiu  $sp, -8
0040FD40 sw     $fp, 8+var_4($sp)
0040FD44 move  $fp, $sp
0040FD48 sw     $a0, 8+arg_0($fp)
0040FD4C la     $v0, WFADevice
0040FD50 li     $v1, 1
0040FD54 nop
0040FD58 la     $v0, WFADevice
0040FD5C li     $v1, 3
0040FD60 nop
0040FD64 move  $v0, $zero
0040FD68 move  $sp, $fp
0040FD6C lw     $fp, 8+var_4($sp)
0040FD70 addiu $sp, 8
0040FD74 jr     $ra
0040FD78 nop
0040FD78 # End of function WFADevice_sub_40FD30
0040FD78

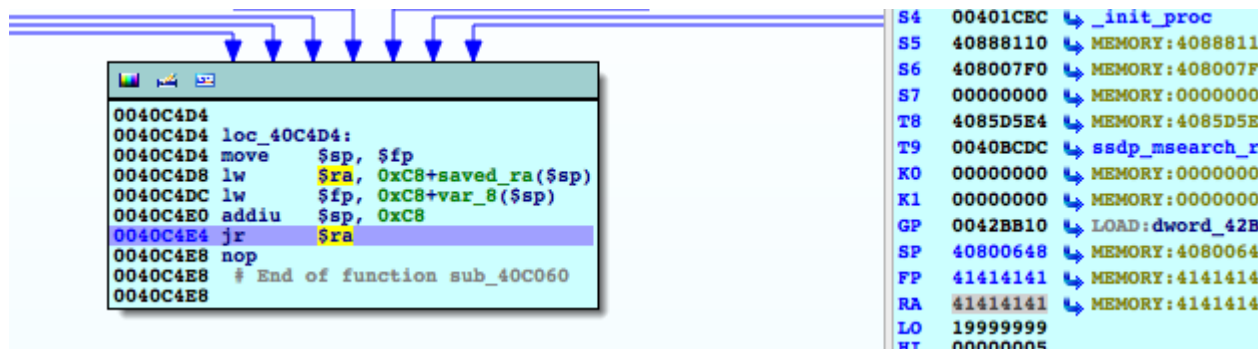
```

Binary patching this function preserves the existing value in the WFADevice structure

## A CRASH, CONTROLLING \$PC

With bcmupnp running in QEMU, I could start sending UPnP packets to it with Miranda in order to build an understanding of what stimuli exercise what code paths. I found several uses of unsafe string handling functions that can be trivially exercised by sending the appropriate UPnP data. Many of them either didn't deal with external data, or dealt with only small amounts such that overflowing a buffer wasn't possible.

However I did find a strcpy() that copies a UUID out of an M-SEARCH packet to the stack. This code path gets exercised when the M-SEARCH's search target (ST:) field is set to "uuid:". The value of the uuid search target is not sanitized or checked for length. It may contain arbitrary data, and is terminated with a null byte. I created a script in Python that generates an M-SEARCH UDP payload and sent it to the 239.255.255.250 multicast address on UDP port 1900. With IDA Pro's debugger attached to QEMU, a "uuid" consisting of an excessively long string of 'A's successfully produced a crash, controlling the \$pc register.



W00t! A crash, controlling the program counter.

From here successful exploitation involved developing a ROP chain to flush the CPU's data cache and to locate the call stack, whose addresses are randomized<sup>13</sup>.

I developed a complete exploit against this application entirely in QEMU MIPS emulation. I did initial testing and debugging of the exploit using QEMU user-mode emulation, chrooting into the BT HH3b's extracted root file system, and running the patched copy of bcmupnp. I replaced the MIPS bin/sh with a statically compiled x86 busybox<sup>14</sup>. Doing this, I was able to overflow the buffer and gain control of execution, resulting in a connect-back shell from my Ubuntu Linux development system.

With the exploit working in QEMU on Ubuntu, the only thing remaining was to find the actual base addresses of libraries containing the exploit's ROP gadgets. These addresses on the live system would obviously differ from the addresses when running on x86 Linux. QEMU's computer mode emulation helped with this problem. I copied the BT HH3b filesystem into a subdirectory of a running QEMU Debian MIPS system. Once again, I chrooted into the HH3b file system and ran the patched bcmupnp. Inspection of /proc/<pid>/maps revealed the base addresses of the libraries, allowing me to rebase the exploit. Once rebased, it worked against the live device on the first try.

Successful exploitation<sup>15</sup> of this buffer overflow yielded a fully privileged root connect-back shell<sup>16</sup>, enabling further analysis of the live device.

<sup>13</sup> While the mechanics of exploiting a stack-based buffer overflow on a MIPS CPU are interesting, they are not unique to this exploit. I encourage readers to view the MIPS-specific section of my presentation from Black Hat USA 2012 and DEF CON 20 entitled "From SQL Injection to MIPS overflows: Rooting SOHO Routers" for a detailed discussion of MIPS buffer overflows.

<http://www.youtube.com/watch?v=RGOjAF-NLY0>

<sup>14</sup> In user mode, QEMU passes system calls up to the host's kernel. The host's x86 kernel can't execve() a non-native bin/sh.

<sup>15</sup> Proof-of-concept exploit code is available for checkout from Github:

<https://github.com/zcutlip/exploit-poc/tree/master/BT/homehub3b>

<sup>16</sup> A video of the exploit in action is available here:

<https://vimeo.com/52954499>



```
4. zach@zolan: ~/code/wifi-reversing/exploit-poc/BT/homehub3b (ssh)
zach@zolan:~/code/wifi-reversing/exploit-poc/BT/homehub3b (1) $ ./hh3b_exploit.py
[+] bad char count: 1
[+] No bad bytes in decoder stub.
[+] Generating key.
Sending exploit
Waiting on callback server.
Listening on port 8080

Waiting for incoming connection.

Target has phoned home.

BusyBox v1.9.1 (2012-06-25 15:17:16 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cat /proc/version
Linux version 2.6.30 (qiufude@linux-whg66) (gcc version 4.4.2 (Buildroot 2010.02-git) ) #1
Mon Jun 25 15:11:19 CST 2012
# █
```

Root prompt. Victory, at last.

## FUTURE WORK

Considerable work on the BT HomeHub 3.0b remains. One of the first things I attempted with the root access I gained was to mount the JFF2 root filesystem read/write, and make some changes to facilitate instrumentation and to provide persistent access that would not require exploitation. These changes to the file system rendered the device inoperable upon reboot. Analysis of the firmware's boot loader revealed that an integrity check is performed across the filesystem before control of the boot process is transferred to the Linux kernel. If this integrity check fails, the boot process stops. At this point there is a web interface available at 192.168.1.1 that prompts the user to provide a restoration image.

Two immediate goals arise from the successful exploitation of this device. First is to obtain a firmware update image. A possible strategy will be to intercept an update from the device's automatic self-updating process. A second goal is to defeat the device's integrity check on the root file system. This could be done by changing a configuration setting in NVRAM, by patching the bootloader to skip the check, or by generating a hash of the filesystem that satisfies the check. These two goals will facilitate further analysis. They will reduce the risk associated with modifying and instrumenting the device.

The overarching goal is to unlock the device to expose configuration flexibility that, by default, is unavailable. Ideally users will be able to re-flash the device with a custom firmware such as OpenWRT.

## CONCLUSION

The American pragmatist philosopher, John Dewey, said that failure is instructive. Even though I succeeded in developing a root exploit, most of the time I invested in this project was spent failing. There are useful lessons to be taken from my failures and from my ultimate success. Sometimes what appears to be obviously vulnerable may not be. What appears on the surface to be a sure command injection can be misleading. Conversely, what may appear to be secure may not be. For example, it can be worthwhile to take a closer look at seemingly robust use of cryptography for implementation weaknesses. While most products in this category have easily exploited buffer overflows, not all do. Assuming you find a buffer overflow candidate, you may need to rely on emulation in order to develop an exploit. Advanced techniques such as binary patching, and function hooking can enable you to discover vulnerabilities and develop exploits entirely in emulation. Most importantly, though, when an easy win doesn't present itself, circle back and dig deeper. All targets are vulnerable, even if they don't at first seem so.

In the end I obtained an unauthenticated, remote root shell on this device in spite of the many challenges involved. Further, I can now develop exploits against a device I do not actually possess, confident they will work on the first try against their live target.

## Appendix A

A program that links the HH3b's libcfmapi.so to decrypt a user's configuration backup file.

```
#include <stdlib.h>
#include <stdio.h>

int restorebackup(const char *tmp_cfg_name,const char *xml_cfg_name);
int ATP_CFM_ExtCustomImportEncryptedUserCfgFile(const char *tmp_cfg_name);
int main(int argc, char **argv)
{
    int ret;
    if(argc < 3)
    {
        printf("specify temp config file name.\n");
        exit(1);
    }
    ret=restorebackup(argv[1],argv[2]);
    return ret;
}

int restorebackup(const char *tmp_cfg_name,const char *xml_cfg_name)
{
    int ret=0;

    //ret = ATP_CFM_ExtDigVerifyFile(tmp_cfg_name,tmp_cfg_name);
    if(ret != 0)
    {
        printf("Verify File failed.\n");
        return ret;
    }
    ret = ATP_CFM_ExtCustomImportEncryptedUserCfgFile(tmp_cfg_name);
    return ret;
}
```

## Appendix B

A program that links the HH3b's libcfmapi.so to encrypt and sign a user's configuration backup file.

```
#include <stdlib.h>
#include <stdio.h>

int encode_config(const char*xml_cfg_name,const char *encrypted_cfg_name);
int ATP_CFM_ExtCustomExportEncryptedUserCfgFile(encrypted_cfg_name);

int main(int argc, char **argv)
{
    int ret;
    if(argc < 2)
    {
        printf("specify encrypted config file name.\n");
    }
}
```

```

        exit(1);
    }
    ret=encode_config(NULL,argv[1]);
    return ret;
}

int encode_config(const char*xml_cfg_name,const char *encrypted_cfg_name)
{
    int ret=0;
    ret = ATP_CFM_ExtCustomExportEncryptedUserCfgFile(encrypted_cfg_name);
    if(ret != 0)
    {
        printf("Compressing, encrypting, and encoding failed.\n");
        return ret;
    }
    ret = ATP_CFM_ExtDigSignFile(encrypted_cfg_name);
    return ret;
}

```

## Appendix C

### Example of a decrypted configuration backup file

```

<?xml version="1.0"?>
<BackupFile>
  <Time>
    <Enable>1</Enable>
    <NTPServer1>ntp.homehub.btopenworld.com</NTPServer1>
    <NTPServer2/>
    <NTPServer3/>
  </Time>
  <UserInterface>
    <X_Web>
      <UserInfo instance="1">
        <Userpassword encrypted="true">AeRhtYDXyTPKe9uBjnSWuQ==</Userpassword>
        <PasswordHint/>
      </UserInfo>
    </X_Web>
  </UserInterface>
  <LANDevice instance="1">
    <LANHostConfigManagement>
      <DHCPSEnable>1</DHCPSEnable>
      <MinAddress>192.168.10.64</MinAddress>
      <MaxAddress>192.168.10.253</MaxAddress>
      <DHCPLeaseTime>86400</DHCPLeaseTime>
      <IPInterface instance="1">
        <IPInterfaceIPAddress>192.168.10.254</IPInterfaceIPAddress>
        <IPInterfaceSubnetMask>255.255.255.0</IPInterfaceSubnetMask>
      </IPInterface>
    </LANHostConfigManagement>
    <WLANConfiguration instance="1">
      <X_InterfaceType>802.11b/g/n</X_InterfaceType>
      <Enable>1</Enable>
      <Channel>6</Channel>
      <AutoChannelEnable>1</AutoChannelEnable>
      <SSID>BTHub3-FQH9</SSID>
      <BeaconType>WPAand11i</BeaconType>
      <WEPKeyIndex>1</WEPKeyIndex>
      <BasicEncryptionModes>WEPEncryption</BasicEncryptionModes>
      <BasicAuthenticationMode>None</BasicAuthenticationMode>
      <WPAEncryptionModes>TKIPEncryption</WPAEncryptionModes>
      <WPAAuthenticationMode>PSKAuthentication</WPAAuthenticationMode>
      <IEEE11iEncryptionModes>AESEncryption</IEEE11iEncryptionModes>
    </WLANConfiguration>
  </LANDevice>
</BackupFile>

```

```

<IEEE80211AuthenticationMode>PSKAuthentication</IEEE80211AuthenticationMode>
<X_MixedEncryptionModes>TKIPandAESEncryption</X_MixedEncryptionModes>
<X_MixedAuthenticationMode>PSKAuthentication</X_MixedAuthenticationMode>
<WPS>
  <Enable>1</Enable>
  <ConfigMethodsEnabled>PushButton</ConfigMethodsEnabled>
</WPS>
<WEPKey instance="1">
  <WEPKey encrypted="true">B2ZSGQpEUq5K1eORXApvKw==</WEPKey>
</WEPKey>
<WEPKey instance="2">
  <WEPKey encrypted="true">CmBgin5slZgIh3N8HJ2LTQ==</WEPKey>
</WEPKey>
<WEPKey instance="3">
  <WEPKey encrypted="true">CmBgin5slZgIh3N8HJ2LTQ==</WEPKey>
</WEPKey>
<WEPKey instance="4">
  <WEPKey encrypted="true">CmBgin5slZgIh3N8HJ2LTQ==</WEPKey>
</WEPKey>
<PreSharedKey instance="1">
  <PreSharedKey encrypted="true">B2ZSGQpEUq5K1eORXApvKw==</PreSharedKey>
</PreSharedKey>
</WLANConfiguration>
<Hosts>
  <Host instance="1">
    <IPAddress>192.168.10.66</IPAddress>
    <AddressSource>DHCP</AddressSource>
    <MACAddress>7C:C3:A1:89:9D:38</MACAddress>
    <HostName>endor</HostName>
    <InterfaceType>Ethernet</InterfaceType>
  </Host>
  <Host instance="2">
    <IPAddress>192.168.10.68</IPAddress>
    <AddressSource>DHCP</AddressSource>
    <MACAddress>00:0C:29:AB:5B:F5</MACAddress>
    <HostName>zolan</HostName>
    <InterfaceType>Ethernet</InterfaceType>
  </Host>
  <Host instance="3">
    <IPAddress>192.168.10.64</IPAddress>
    <AddressSource>DHCP</AddressSource>
    <MACAddress>C8:2A:14:38:D1:81</MACAddress>
    <HostName>endor</HostName>
    <InterfaceType>Ethernet</InterfaceType>
  </Host>
  <Host instance="4">
    <IPAddress>192.168.10.65</IPAddress>
    <AddressSource>Static</AddressSource>
    <MACAddress>00:0C:29:AB:5B:FF</MACAddress>
    <HostName>Unknown-00-0C-29-AB-5B-FF</HostName>
    <InterfaceType>Ethernet</InterfaceType>
  </Host>
</Hosts>
</LANDevice>
<WANDevice instance="1">
  <WANConnectionDevice instance="1">
    <WANPPPConnection instance="1">
      <Username>foo@btbroadband.com</Username>
      <Password/>
      <X_DMZ>
        <Enabled>0</Enabled>
        <DeviceMAC/>
      </X_DMZ>
    </WANPPPConnection>
  </WANConnectionDevice>
</WANDevice>
<X_PortForwarding>
  <RuleSet instance="1">
    <Name>myapp</Name>
    <MAC/>
    <IP>127.0.0.1</IP>
  </RuleSet>
</X_PortForwarding>

```

```

        <Name/>
    </RuleSet>
    <Customgame instance="1">
        <Name>myapp</Name>
        <Ports instance="1">
            <Protocol>TCP</Protocol>
            <TranslatePort>22</TranslatePort>
            <PortStart>22</PortStart>
            <PortEnd>22</PortEnd>
        </Ports>
    </Customgame>
</X_PortForwarding>
<X_AccessControl>
    <Enable>0</Enable>
</X_AccessControl>
<X_PowerSave>
    <Enable>0</Enable>
    <PSWindow>
        <PSWindowStart>00:00</PSWindowStart>
        <PSWindowStop>07:00</PSWindowStop>
    </PSWindow>
</X_PowerSave>
<X_PortClamping>
    <Enabled>0</Enabled>
</X_PortClamping>
<X_DDNS>
    <Enabled>0</Enabled>
    <ServiceName/>
    <Username/>
    <Password/>
</X_DDNS>
<UpnP>
    <Enabled>1</Enabled>
    <ExtendedSec>1</ExtendedSec>
</UpnP>
<X_Firewall>
    <Mode>Disabled</Mode>
</X_Firewall>
<X_FixedIP instance="1">
    <IP/>
    <MAC/>
</X_FixedIP>
<X_FixedIP instance="2">
    <IP/>
    <MAC/>
</X_FixedIP>
<X_FixedIP instance="3">
    <IP/>
    <MAC/>
</X_FixedIP>
<X_FixedIP instance="4">
    <IP/>
    <MAC/>
</X_FixedIP>
<X_FixedIP instance="5">
    <IP/>
    <MAC/>
</X_FixedIP>
</BackupFile>

```