CORE Security Technologies

Killing the myth of Cisco IOS rootkits: DIK (Da Ios rootKit)

Sebastian 'topo' Muñiz
March 2008

Abstract

Rootkits are very common in most popular Operating Systems like Windows, Linux, Unix and any variant of those but they are rarely seen in embedded OS's.
This is due to the fact that most of the time embedded OS's are closed source, hence internals of the OS are unknown and reverse engineering process is harder than usual.

In real life, it's very common that once an attacker takes control of a system he or she needs to maintain access to it so a rootkit is installed.
The rootkit seizes control of the entire system running on that hardware by hiding files, processes, network connections, allowing unauthorized users to act as system administrators, etc..

This paper demonstrates that a rootkit with those characteristics can be easily created and deployed for a closed source OS like IOS and run unnoticed by system administrators by surviving to most, if not all, of the security measures given by experts on the field.

As a proof of this, different ways to infect a target IOS will be shown like run-time patching and image binary patching.
To discuss the binary patching technique from a practical point of view, DIK (Da Ios rootKit) which is a set of python[1] scripts that provides a generic rootkit implementation for IOS will be introduced.

Introduction

The case of Cisco IOS (Internetwork Operating System) is special because it is probably the most widely deployed routing Operating System running on Internet and is part of mission critical networking operation on most, virtually any, organization.

Network devices are critical for every organization and sensitive data goes through them every second making them strategic locations for an attackers to place a rootkit to gather all kinds of information about the target.

System administrators should be prepared for this kind of threats because a serious information leak can happen before they realize that something is going on.

Security measures are usually taken to detect abnormal operations on Cisco devices but sometimes those measures may not be enough to detect advanced rootkits and may only unveil high-level rootkits (comparable to user-mode rootkits on Windows) like a TCL script (only recent versions of IOS support TCL as a scripting language) or device reconfiguration via startup-config file to alter routes, packet handling, etc.

Only a small percentage of the system administrators perform periodic security audits on the organization network infrastructure to detect a system compromise. These audits may be (but are not limited to): verifying router logs, checking external logs that were set by the router when a user logged in or changed the device configuration, or even downloading the IOS image to compare it's checksum with a previously calculated value from the original IOS image file. For any of these actions to take place, the system administrator implicitly relies on the IOS internal functions and if the device is compromised, the logging and sys log functions can be altered to cover the attacker's actions making the audit completely useless.

Knowing the enemy

Along the years Cisco created multiple hardwares (even using different CPU's architectures) with different software features sets (i.e., VoIP) to address the needs of their customers. This required multiple and unique IOS versions available because each one of them needed a separate build process for the specific feature set to run on the target hardware.

Another important thing is that IOS is not prepared to support additional modules or plug-ins to be loaded.

With all this in mind we jump into the conclusion that a generic solution for a rootkit might be too difficult if not impossible to implement.

It will be demonstrated that this can be easily solved with a generic method which will address the needs to maintain code for multiple architectures or programming the rootkit core in different assembly languages.

IOS Internals

Cisco IOS has a monolithic architecture which runs as a single image and all processes have access to each others memory.
No memory protection between processes is implemented which means that a bug in a process can (and will probably do) corrupt other processes and compromise system operations leading to a general failure.

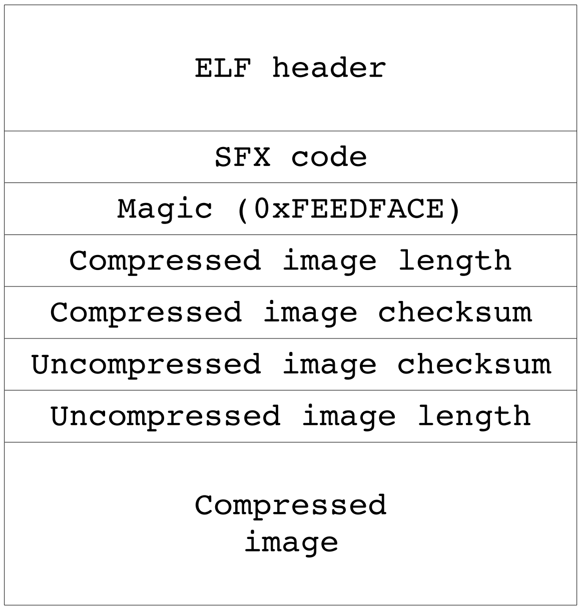
Another characteristic of the Cisco IOS is that the scheduler is not preemptive like it's counterparts on modern OS's. It has a 'run to completion' scheduler which means that when a process is scheduled, it runs until it decides to resign to this privilege and make a system call to allow other processes to run.

Cisco IOS images are usually made of a 32-bits ELF file running on a hardware with a RISC processor (usually MIPS or PowerPC).
It's important to note that some twisted Cisco engineers modified some of the values from a standard ELF header so any tool trying to obtain information from the file will find lots of invalid values thus making initial diagnostic a little bit annoying.

IOS initial setup on memory

This image contains a SFX (self decompressing) header that unpacks the fully functional IOS code which will be relocated in memory during run-time.

Decompression and relocation involves several steps which must be understood because the compressed IOS image will be manipulated to unpack it, insert the backdoor into it and repack the image with the corrected checksums to bypass initializing tests that would forbid the modified image to run on the device.
An IOS compressed image has the following structure:


ELF header
SFX code
Magic (0xFEEDFACE)
Compressed image length
Compressed image checksum
Uncompressed image checksum
Uncompressed image length
Compressed image

Once the device powers on, it will start the ROM Monitor who will perform several steps to load the IOS image.
Those steps are:

- The ROM Monitor will load and position the self compressing image at its link address in memory (either from a flash boot or a netboot) as the ELF header specifies.

- The main routine in the compressed image then checks to ensure that enough memory is available for the decompression.

- The compressed image is checksummed and the result is compared against the value stored in the file to ensure that no corruption has occurred. Then it is moved to a higher memory location and the BSS section is initialized with zeros.

- Decompression process takes place. Once it finishes, the size of the decompressed image is compared against the value stored in the header to ensure that was completely successful. The decompressed image is also checksummed to ensure there was no corruption.

- A function call performs the image relocation in memory and after that the decompressed image entry point is called.

The beginning of the end

The rootkit will commonly hook/patch certain key (and usually low level) functions of the OS being compromised. These functions are strategic locations to intercept data of interest to the attacker. They could be grouped by their functionality:

- System Login
- Authentication and authorization
- File system access
- Networking operations
- Process handling
- Information displaying
- System Logs

In the case of a closed source OS like IOS the first we need to do is to identify the code that carry out those above mentioned functions. For that we will download the image running on the device that we want to infect. This can be done by configuring a server on an attacker controlled machine and issuing a copy command on the Cisco device command line.
With the target IOS image downloaded we can now proceed to the analysis phase.

Chasing the prey

Once we have the file, we must follow a few step to be able to detect the above mentioned functions:

1. Proceed to unpack the image using a script[2] called 'ciscoutils' which is part of DIK. Once the image is unpacked, we will checksum it to ensure there was no corruption.
   The decompression process is the same as for any zipped file so we can use any free unzip utility to do it and then perform the checksum without the decompression using the same script.

2. The decompressed image must be analyzed using IDA Pro[3] to obtain crucial information for the rootkit survival.
   This can take several minutes, even hours because uncompressed IOS image files take up several megabytes (specially the ones with advanced features sets).

3. We will note that once IDA finishes the analysis, it won't be successful because several functions and multiple string references will be missing.
   To address this problem we will use another script included with DIK called 'enhanced_analysis'[4] which uses IDAPython[5].
   The script will create separate segments for CODE and DATA and it will recognize every function and every string reference in the image.

Once the script finishes the image is ready to use and can be examined by the attacker to gain knowledge of the internals of the Cisco IOS.

Successful analysis of the image file is very important because it contains plenty of debugging strings to provide verbose information to the system administrator about the OS state and those debug strings will be used as starting point to detect the key functions of the OS and because we know for sure that the strings remain the same across multiple IOS versions.

Resistance is futile

Some of those interesting functions might not be located because of compiling issues it might not be possible to retrieve any string references or simply because they do not use any strings at all.

As we said before, the IOS contains plenty of strings, most of them with debugging information and others just output the commonly seen messages to the user terminal. These message can be located in functions close to the ones we are looking for and knowing that they will not be moved by the compiler, we can try to find these 'neighbour' functions and then identify the ones we are interested in hooking with the rootkit.

The location of neighbour functions is not necessarily immediate to the one we are looking for, there can be another function without any string references separating them but still this approach will succeed.

Home sweet home

The rootkit location must be decided before any image patching takes place (whether it is on the file or at run-time) because the patches will jump to the rootkit code and they must know it's memory location.

Taking advantage of IOS memory management protection (or the lack of it) we will write the rootkit code on the DATA segment by sacrificing a debug string which will probably never be used.
Just in case that the system administrator decides to use some IOS feature that requires that string, we will put a NULL at the first character.

There are several ways to insert the rootkit code in the file and they are all well known for any Linux virus writer because it's mainly an standard ELF infection procedure[6][7].
No detailed explanation will be given about those techniques, only for the sake of clarity it will mentioned that overwriting an existing string resource in the file is the method we chose.

This method is the easiest in this case because IOS images contain very long strings that are rarely used and there is no need to modify the ELF header values because every section and segment remains the same.
In some cases the DATA segment permissions need to be changed to Read-Write-Execute.

In case the attacker wishes to create an additional section in the image file, it can be easily done with the PyElf[8] library specially created for this project.

Image manipulation must be done very carefully because it will be relocated after the decompression process and any invalid memory reference could lead to an exception resulting in a system crash.

We must store the memory address that points to the end of the rootkit code for further operations on the image.

Code voyeurism and fetishism

Once the key functions were found, we will discuss the rootkit insertion by binary patching the image. Once in control of the function it will take different actions based on the parameters passed at run-time.

Let's take for example the password checking function.
In this case the rootkit must take control at the beginning of the function to check if the rootkit password was entered. That means that some instructions (architecture dependent) will be overwritten at the prologue of the function and stored for further usage.

Due to the nature of the RISC architecture (despite of differences between MIPS and PowerPC) we must store the return address and then set a chunk of assembly code called 'trampoline' that will redirect the execution flow to a 'stub'.

The so called trampoline is responsible for saving the return address and jumping immediately (and unconditionally) to an attacker specific code which will ultimately call the function of the rootkit to validate the password and redirect execution flow again based on the result.

The location that the trampoline jumps to is called 'stub' and it is responsible of saving the return address of the caller (in this case the trampoline's address), calling the the rootkit function with the same arguments of the IOS legitimate function and process the result of the function call.
This result is needed to decide if it will return to the trampoline and continue the original execution flow by previously executing the instructions overwriting by the trampoline (in case that the password entered is not the rootkit password) or return directly to the trampoline's caller because no more password validation is needed (in case the password entered is the rootkit master password) which means that the attacker is logging in.

Some of those painful steps might not be necessary if the rootkit code was implemented in pure assembly but in the case of DIK it was implemented in plain C.
Those few lines of special assembly instructions called 'trampoline' and 'stub' were needed to fill the gap between a C function compiled (with position independent code) for the target architecture and extracted to be inserted 'as is' directly inside the IOS image.
The advantage of this method is that only one C code is maintained (with certain limitations, of course) instead of two codes that perform the same actions on different architectures (a MIPS code and a PowerPC code).

Functioning without the others functions

A function that performs password checking is useful to retrieve other user's password in plain text and if this information could be written somewhere (may be a hidden file on flash) then it would be of great interest for an attacker.

There are several functions besides the one mentioned above that a rootkit must hook/patch to take complete control of the system.
Those functions include equivalents of file handling functions like read/write, socket handling like send/recv and IOS functions that implement the CLI (Command Line Interface) commands entered that can alert the system administrator of unauthorized access.

No discussion will be made about all those functions but it must be said that some of them are already hooked by DIK. Operations like downloading the IOS image in a periodic manner by the system administrator to perform a checksum (like MD5, SHA1, etc.) as part of the security measures to detect modified images could be easily redirected to an external server that contains an unaltered image without any suspicion. It could even intercept the read() function calls asking for a chunk of the compressed image on flash (or any other media) and in that moment it decompress the infected chunk, patch it with the original bytes and re-compress it so it's returned intact (this is possible since the compression algorithm can work with chunks of bytes instead of the entire file).

At this moment the difference between a low level rootkit and a simple TCL script can be appreciated because such actions like the one mentioned before could never be taken by a higher level rootkit. Another advantage of the method is that no process is running to perform those actions.

Ready, steady, go

With the rootkit code in place, we are now ready to dump the newly patched IOS image, repack it with the original (self decompressing) file header and upload it to the target system.
All the patching techniques and the IOS dump is done by the main IDAPython script called 'image_patcher'.

The patched IOS image must be checksummed again because now that it's contents have changed then the old checksum values won't match.
The script 'ciscoutils' previously mentioned can be used to recalculate all the checksums and recreate the IOS self decompressing image ready to be used by the hardware.

Other ways of The Force

Image binary patching has been discussed in depth but run-time memory patching technique is also possible using the GDB[10] stub inside every IOS image.

The GDB stub is the debugging interface for Cisco developers which allows them to debug IOS processes. It also allows remote image diagnostic because it's capable of working over a Telnet session as well as over a Serial session establish on the console port.

This GDB stub is capable of working in three different ways:

- Process examination: Allows memory inspection and processor registers inspection but it cannot modify system values (memory of registers values).
  The system execution continues normally during debugging so 'examine' mode can be executed over a Telnet session.

- Process debugging: In the situations that a console port of the device is not accessible, process debug mode can be executed. It works by catching unhandled exceptions on the specified process, setting it in a special state where it will not be rescheduled and then running the process of the debugger to debug the failed process.
  The IOS system continues to run during process debugging so it is possible to debug a process over a Telnet session but certain restrictions apply. The scheduler, an interrupt service routine or any process needed for the debugging path (such as TCP/IP) cannot be debugged over this session.
  This debugging mode is capable of memory and processor registers modification so this is the best option for an attacker to remotely modify the device memory to insert the backdoor.

- Kernel debugging: If the attacker gains physical access to a console port he or she can execute the kernel debugger which is the preferred way to debug a router. In this mode, the entire device execution is stopped during the exception, freezing all system states.

Using the Telnet connection, a remote GDB instance can be executed to perform memory patching but certain precautions must be taken, like not writing the trampoline code before the rootkit code because if a patched function is invoked before the rootkit code is in place a memory access violation will be raised leading to a system crash.

An attacker might want to automate this run-time patching procedure for every system restart and it can be accomplished in a few different ways.

One possible way is to create a TCL script to execute at startup, engage a Telnet session with the local host and executing the process debugger to patch the device it is running on. In this case the script must contain the rootkit code inside with the memory locations to be modified which could have been previously obtained by the same analysis phase that the image binary patching procedure.

Conclusions

A reliable and generic method for Cisco IOS image infection can be implemented either via binary image modification or via run-time code patching.

To face this kind of threat the only possibility available today is CIR[11] created by Felix 'FX' Lindner from Recurity Labs and presented early this year when he talked about developments on IOS forensics[12].
It's also important make a special mention on this because this is the ONLY serious (and possible) way to perform forensics on a Cisco device and still it might not be enough if the rootkit controls the core-dump generation routines.

Unless every system administrator plans on using advanced forensics methods on every device on their networks, they should take serious security measures and try to keep the devices updated to minimize the risk.
Even this may not be enough to detect an advanced rootkit already deployed in the system.

References

[1] A free python interpreter for Windows called ActivePython can be obtained at:
http://www.activestate.com/Products/activepython/features.plex

[2] The script called 'ciscoutils.py' can be used to decompress, checksum and recreate a compressed file. Usage information could be obtained by executing the script without parameters of with --help.

[3] IDA Pro is disassembler and debugger that can be obtained at
http://www.hex-rays.com/idapro/

[4] The script called 'enhanced_analysis.py' can be used to completely analyze the file thus allowing other scripts to successfully patch the IOS image to insert the backdoor.

[5] IDAPython is a plug-in for IDA Pro to allow python scripts to be executed in the context of IDA and to access all of its functions. It can be downloaded from http://d-dome.net/idapython

[6] 'The ELF virus writing HOWTO' at
http://www.linuxsecurity.com/resource_files/documentation/virus-writing-HOWTO/_html/index.html

[7] Daniel Hodson presentation at RUXCON 2004 is available at
http://www.ruxcon.org.au/files/2004/11-daniel_hodson.ppt

[8] PyElf is a simple library for easy ELF file manipulation. Refer to file pyelf.py for usage help.

[9] The script called 'image_patcher.py' is responsible of functions locating through strings, binary patching and image dumping to disk.

[10] GDB is The GNU Debugger Project and information about it can be obtained from http://sourceware.org/gdb/

[11] CIR (Cisco Information Retrieval) is accessible at
http://cir.recurity-labs.com/

[12] 'Developments in IOS Forensics' paper can be obtained at
http://www.recurity-labs.com/content/pub/RecurityLabs_Developments_in_IOS_Forensics.pdf