Killing the Myth of Cisco IOS Diversity* :

Recent Advances in Reliable Shellcode Design

Ang Cui
Department of Computer Science
Columbia University
New York NY, 10027, USA
ang@cs.columbia.edu

Jatin Kataria
Department of Computer Science
Columbia University
New York NY, 10027, USA
jk3319@columbia.edu

Salvatore J. Stolfo
Department of Computer Science
Columbia University
New York NY, 10027, USA
sal@cs.columbia.edu

ABSTRACT

IOS firmware diversity, the unintended consequence of a complex firmware compilation process, has historically made reliable exploitation of Cisco routers difficult. With approximately 300,000 unique IOS images in existence, a new class of version-agnostic shellcode is needed in order to make the large-scale exploitation of Cisco IOS possible. We show that such attacks are now feasible by demonstrating two different reliable shellcodes which will operate correctly over many Cisco hardware platforms and all known IOS versions. We propose a novel two-phase attack strategy against Cisco routers and the use of offline analysis of existing IOS images to defeat IOS firmware diversity. Furthermore, we discuss a new IOS rootkit which hijacks *all* interrupt service routines within the router and its ability to use intercept and modify process-switched packets just before they are scheduled for transmission. This ability allows the attacker to use the payload of innocuous packets, like ICMP, as a covert command and control channel. The same mechanism can be used to stealthily exfiltrate data out of the router, using response packets generated by the router itself as the vehicle. We present the implementation and quantitative reliability measurements by testing both shellcode algorithms against a large collection of IOS images. As our experimental results show, the techniques proposed in this paper can reliably inject command and control capabilities into arbitrary IOS images in a version-agnostic manner. We believe that the technique presented in this paper overcomes an important hufdle in the large-scale, reliable rootkit execution within Cisco IOS. Thus, effective host-based defense for such routers is imperative for maintaining the integrity of our global communication infrastructures.

*Video demos of both IOS shellcodes and our stealthy exfiltration module can be found at [7].

1. INTRODUCTION

Over the past decade, Cisco IOS has been shown to be vulnerable to the same types of attacks that plague general purpose computers [13, 11]. Various exploitation techniques and proof-of-concept rootkits [14, 12] have been proposed. However, all current offensive techniques are impeded by an unintended security feature of IOS: diversity. As Felix "FX" Linder points out, Cisco IOS is not a homogenous collection of binaries, but a collection of approximately 300,000 diverse firmwares [12]. Although never intended as a defense against exploitation, this diversity makes the creation of reliable exploits and rootkits difficult.

Known proof-of-concept rootkits operate by patching specific locations within IOS. In the case of DIK [14], the rootkit intercepted a specific function responsible for checking password. The major drawback of this approach is that it relies on *a priori* knowledge of the location of this function. As previously noted, this knowledge is generally difficult to obtain with accuracy prior to attack. Therefore, any rootkit which depends on specific memory locations cannot be used reliably in large-scale attacks against the Internet substrate. Conversely, version-agnostic shellcode, combined with known vulnerabilities in IOS, makes such large-scale attacks against Cisco routers a feasible reality.

For reliable, large-scale payload execution in IOS to be feasible, we must construct attacks and shellcodes which are version and platform agnostic. Towards this end, we outline a two-stage attack methodology as follows:

Stage 1: Leverage some IOS invariant to compute a host fingerprint. Using computed information, inject stage-2 shellcode. Furthermore, exfiltrate host fingerprint back to attacker.

Stage 2: Persistent rootkit with covert command and control capability. The attacker will use exfiltrated fingerprint data to construct a version specific rootkit, which is loaded via the second-stage shellcode.

The attacker is at a disadvantage when attempting an online attack. However, since all IOS images can be obtained, and such images are not polymorphically mutated, an attacker can construct a large collection of version specific rootkits offline. If the attacker is able to simultaneously inject a simple rootkit and exfiltrate a host-environment fingerprint

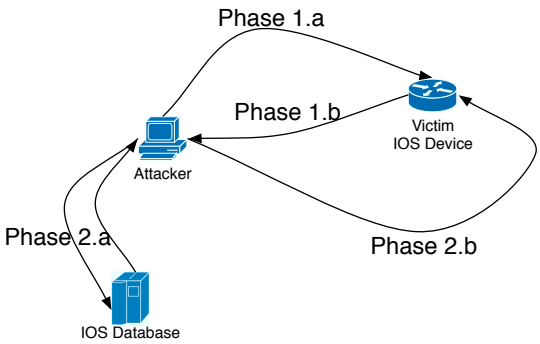

Figure 1: Timeline of two-stage attack against vulnerable IOS router of unknown hardware platform and firmware version. Attacker launches exploit with reliable shellcode (1.a). Shellcode installs rootkit and exfiltrates victim device's IOS fingerprint (1.b). The attacker finds exact IOS version from fingerprint by consulting offline database (2.a). The attacker then creates a version specific rootkit for victim, and uploads it using 1.b rootkit (2.b).

during the first phase of the attack, the attacker can then load a rootkit specifically parameterized for the exact IOS version of the victim router. Figure 1 shows the timeline of our proposed attack, which is intentionally broken into two phases to shift the advantage towards the attacker.

The two requirements of our first-stage shellcode, the need to reliably inject a basic second-stage rootkit, and the need to accurately fingerprint the victim device, can be satisfied simultaneously. Both shellcodes presented in this paper compute a set of critical memory locations within IOS's .text section. These memory addresses are used both as intercept points for our second-stage code, but also used to uniquely identify the exact micro-version of the victim's firmware. As figure 1 shows, this fingerprint data is exfiltrated back to the attacker and compared to a database of pre-computed fingerprints for all known IOS firmwares. As Section 8 shows, the fingerprints can be computed using simple linear-time algorithms and efficiently stored in a database. Pre-computing such fingerprints for all 300,000 IOS images should not take more than a few days on a typical desktop.

We present two different techniques for implementing this two-stage attack. The disassembling shellcode is discussed in Section 5. A novel interrupt hijack shellcode is discussed in Section 6. A stealthy exfiltration technique which modifies process-switched packets just before it is scheduled for transmission is discussed in Section 7. The intercept hijacking shellcode and the exfiltration mechanism built on top of it has several interesting advantages over existing rootkit techniques. First, the command and control protocol is built into the payload of incoming packets. No specific protocol is required, as long as the packet is punted to the router's control-plane. This allows the attacker to access the backdoor using a wide gamut of packet types, thus evading network-based intrusion detection systems. Hiding the rootkit inside interrupt handlers also allows it to execute *forever* without violating any watchdog timers. Furthermore, the CPU overhead of this shellcode will be distributed across a large number of random IOS processes. Unlike with shellcodes which take over a specific process, the network administrator can not detect unusual CPU spikes within any particular process using commands like *show proc cpu*, making it very difficult to detect by conventional means.

The remainder of this paper is organized as follows: Section 2 outlines the challenges of reliable IOS rootkit execution and provides motivation for the need for version-agnostic shellcodes. Section 3 presents a survey of advancements in Cisco IOS exploitation over the past decade and provides a timeline of public disclosures of significant vulnerabilities and exploitation techniques. Section 4 outlines a general two-stage attack strategy against unknown Cisco devices. Section 5 presents our first reliable IOS shellcode, a disassembling shellcode, which was first proposed by Felix Linder for PowerPC based Cisco devices. Section 6 presents our second reliable IOS shellcode. This shellcode hijacks all interrupt handler routines within the victim device, and is faster, stealthier and more reliable than our first shellcode. Experimental data, performance, overhead and reliability measurements are presented in Section 8. Potential defenses against our proposed shellcodes are discussed in Section 9. Concluding remarks are presented in Section 10. Lastly, the full source code of both shellcodes are listed in Appendix A.

Please note that the remainder of this paper will focus on MIPS-based Cisco IOS. All code examples will be shown in MIPS. However, the techniques presented can be applied to PowerPC, ARM and even x86-based systems.

2. MOTIVATION

Several recent studies demonstrate that there are vast numbers of unsecured, vulnerable embedded devices on the Internet [9], such devices are vulnerable to the same types of attacks as general purpose computers [3, 11], and can be systematically exploited in much the same way [1, 3, 5]. For example, various exploitable vulnerabilities [13, 12] and rootkits [14] have been found and disclosed for Cisco's flagship system, IOS. Cisco devices running IOS constitutes a significant portion of our global communication infrastructure, and are deployed within critical areas of our residential, commercial, financial, government, military and backbone networks.

Typical of the embedded security landscape, IOS is an aging system which does not employee standard protection schemes found within modern operating systems [14], and does not have any host-based anti-virus to speak of. In fact, not only is the installation of third-party anti-virus (which does not yet exist for IOS) not possible via any published OS interface, any attempt to do so will also violate the vendor's EULA and thus void existing support contracts.

Consider the availability of proof-of-concept exploits and rootkits, the wide gamut of high-value targets which can be compromised by the exploitation of devices like routers and firewalls, and the lack of host-based defenses within close-source embedded device firmwares. Such conditions should make the vast numbers of vulnerable embedded devices on the Internet highly attractive targets. Indeed, we have observed successful attempts to create botnets using Linux-

based home routers [4]. As Section 3 shows, the necessary techniques of exploiting Cisco IOS and installing root-kits on running Cisco routers are well understood. However, an obstacle still stands in the way of reliable large-scale exploitation of Cisco devices: *firmware diversity.*

As Felix Linder and others have pointed out [12], there are over 300,000 unique versions of Cisco IOS. Diverse hardware platforms, overlapping feature-sets, cryptography export laws, licensing agreements and varying compilation and build procedures all contribute to create an operating environment that is highly diverse. Although unintentional and not strictly a defense mechanism, this firmware diversity has made the deployment of reliable attacks and shellcodes difficult in practice. Therefore, in order for IOS exploitation to be feasible and practical, reliable shellcode that operate correctly across large populations of IOS versions is needed.

As Linder demonstrates [12], certain common features within Cisco routers can be used to improve the chances of reliable execution of IOS shellcode. The disassembling shellcode concept was proposed in the same work. Building off this insight, we first tested the reliability of the proposed disassembling shellcode. While this technique works smoothly across all versions of IOS for several major hardware platforms, it failed on all versions of IOS for several popular platforms, including the Cisco 2800 series routers. Furthermore, its computational complexity frequently triggered watchdog timer exceptions, which logs a clear trace of the shellcode. Section 5 discusses the reason for this failure, and several other drawbacks of this disassembling approach.

Looking to improve reliability and performance, we constructed a different shellcode by leveraging a common invariant of not only Cisco IOS, but all embedded systems, *interrupt handler routines.* Hijacking interrupt handlers is advantageous for several reasons. First, such routines can be identified by a single 32-bit instruction, **eret**, or *exception return.* The search for a single **eret** instruction reduces the computational complexity of the first-stage shellcode. Whereas the disassembling shellcode frequently causes watchdog timer exceptions on busy routers (See Section 5), the interrupt-handler hijacking first-stage shellcode executes quickly enough to avoid such timer exceptions, even on heavily utilized routers. Second, there are approximately two dozen interrupt handler routines on any IOS image, all of which are clustered around a common memory range. By using offline analysis of large numbers of IOS images, we can safely reduce the memory range searched by the first-stage shellcode to a small fraction of IOS's .text section, further improving the efficiency of the shellcode (See Figures 8 and 9).

As our experimental data shows, the two proposed shellcodes, along with our proposed data exfiltration mechanism presented in Section 7, combined with available vulnerabilities of Cisco IOS makes the large-scale of Cisco routers feasible. Weaponizing the techniques presented in this paper to create worms which target routers is possible, and can seriously damage the Internet substrate. Therefore, the development of advanced host-based defense mechanisms to mitigate such techniques should now be considered a necessity. Section 9 discusses potential host-based defenses for Cisco IOS and other similar embedded devices.

3. RELATED WORK

A timeline of significant advancements in offensive technologies against Cisco IOS is listed below.

FX, 2003: FX analyzes several IOS vulnerabilities and various exploitation techniques [11].

Lynn, 2005: Lynn described several IOS shellcode and exploitation techniques, demonstrating VTY binding shellcode [13].

Lynn, 2005: Cisco and ISS Inc. files injunction against Michael Lynn [2].

Uppal, 2007: Uppal releases IOS Bind shellcode v1.0 [16].

Davis, 2007: Davis releases IOS FTP server remote exploit code [10].

Muniz: 2008 Muniz releases DIK (Da IOS rootKit) [14].

Futoransky: 2008 Futoransky presented DIK (Da IOS rootKit) [17].

FX, 2009: FX demonstrates IOS diversity, demonstrates reliable disassembling shellcode and reliable execution methods involving ROMMON [12].

Muniz and Ortega, 2011: Muniz and Ortega releases GDB support for the Dynamips IOS emulator, and demonstrates fuzzing attacks against IOS [15].

The techniques presented in this paper extend the above line of work by introducing novel methods of constructing reliable IOS shellcodes and stealthy exfiltration, making large-scale exploitation feasible across all IOS-based devices.

4. TWO-STAGE ATTACK STRATEGY

Sections 5 and 6 discusses two reliable shellcode techniques. Unlike existing IOS shellcodes, these two examples are designed to work in a two-phase attack. Figure 1 illustrates the attack process. In general, this attack first computes a series of memory locations which the second-stage shellcode will intercept to obtain minimal rootkit capability. This series of memory locations is also exfiltrated back to the attacker after the first-stage rootkit finishes execution. Using this information as a host fingerprint, the attacker queries a database of pre-computed fingerprints for all known IOS images to determine the exact micro-version of firmware running on the victim router. Once this is known, a version specific rootkit can be constructed automatically, then loaded onto the victim router via the rootkit installed by the first-stage shellcode.

Each shellcode computes a different set of features. In the case of the disassembling shellcode, a 2-tuple is computed; the address of an invariant string and the address of the password authentication function. In the case of the interrupt hijacking shellcode, a n-tuple is exfiltrated, containing a list of memory address for all interrupt handler routines on the victim device. Section 8 will discuss how accurately each feature-set can uniquely identify the micro-version of the victim IOS environment.

5. SHELLCODE #1: DISASM SHELLCODE

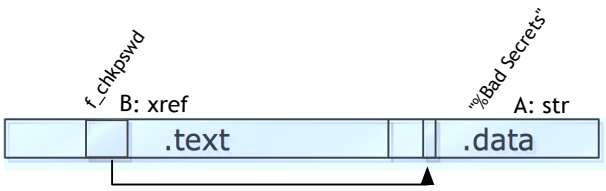

Figure 2: The disassembling shellcode first locates a known string (A), then locates a xref to this string (B). Once this xref location is found, the attacker can patch the function containing the xref. This shellcode requires two linear scans of IOS memory, one through the .data section, and a second one through the .text section.

First proposed by Felix Linder [12] for PowerPC-based routers, the disassembling shellcode scans the victim router's memory twice in order to locate and patch a target function based on some functional invariant, and works as follows:

A. Find String Addr: Scan through memory, looking for a specific string pattern. For example, '%Bad Secrets'.

B. Find String-Xref: With the string's memory location known, construct the instruction which loads this address. Rescan through memory, looking for code which references this string.

C. Patch Function: The data reference is located within the function we wish to find. Search within this function for the desired intercept point. For example, the function entry point, or a specific branch instruction.

Any function which prints a predictable string can be identified and patched in this manner. A particularly useful function is the credential verification function, which prints '%Bad Secrets' when the wrong enable password is entered 3 times.

Figure 3 shows the disassembly of this function. We can bypass password authentication by overwriting a single *move* instruction, highlighted in red.

As experimental results in Section 8 shows, this first-stage shellcode reliably disables password authentication for all versions of Cisco 7200 and 3600 IOS images tested. However, it failed for all Cisco 2800 series IOS images.

In general, this type of disassembling shellcode is suitable for finding *direct* data references, and will fail to find *indirect* references. Indirect references can be identified at the price of computational complexity. In the case of Cisco routers, this limit is a very practical one. A watchdog timer constantly monitors every process within IOS, terminating any process running for longer than several seconds.[1] As Figure 11 shows, our implementation of the disassembling shellcode frequently caused watchdog timer exceptions to be thrown, leaving clear evidence of the attack in the router's logs.

[1]The default watchdog timer value is 2 seconds.

```
text:829EB638                move    $a2, $zero
text:829EB63C                jal     sub_829EB50C
text:829EB640                nop
text:829EB644                bnez    $v0, loc_829EB66C
text:829EB648                li      $v0, 1
text:829EB64C
text:829EB64C loc_829EB64C:                          # CODE XREF: sub_829EB5C4+70|j
text:829EB64C                slti    $v0, $s0, 3
text:829EB650                bnez    $v0, loc_829EB60C
text:829EB654                move    $a0, $s5
text:829EB658                lui     $v1, 0x6396
text:829EB65C                addiu   $a0, $v1, aBadSecrets  # "\n%% Bad secrets\n"
text:829EB660
text:829EB660 loc_829EB660:                          # CODE XREF: sub_829EB5C4+2C|j
text:829EB660                jal     sub_806607AC
text:829EB664                nop
text:829EB668                move    $v0, $zero
text:829EB66C
text:829EB66C loc_829EB66C:                          # CODE XREF: sub_829EB5C4+80|j
text:829EB66C                lw      $ra, 0x90+var_8($sp)
text:829EB670                lw      $s5, 0x90+var_C($sp)
text:829EB674                lw      $s4, 0x90+var_10($sp)
text:829EB678                lw      $s3, 0x90+var_14($sp)
text:829EB67C                lw      $s2, 0x90+var_18($sp)
text:829EB680                lw      $s1, 0x90+var_1C($sp)
text:829EB684                lw      $s0, 0x90+var_20($sp)
text:829EB688                jr      $ra
text:829EB68C                addiu   $sp, 0x90
text:829EB68C  # End of function sub_829EB5C4
```

Figure 3: A disassembly of a typical f_chkpasswd. The string xref is the first highlighted block. The second highlighted block is the single instruction which can disable password authentication in IOS. While these addresses vary greatly, they can be reliably computed at exploitation time by the disassembling shellcode.

Once the first-stage completes execution, the attacker can connect to the victim router with level 15 privilege, bypassing authentication. The attacker can then identify the exact IOS version by a number of methods by using the router's administrative interface. While this backdoor gives the attacker persistent control of the device, it is not covert. Section 6 shows our interrupt hijack shellcode, which installs an equivalent backdoor through a covert channel, using payloads of IP packets punted[2] to the router's CPU. In our demonstration, we use a large collection of arbitrary UDP and ICMP packets to load complex rootkits into the router's memory.

The video demonstration of the disassembling shellcode running on a Cisco 7204 and 12.4T IOS can be found at [7].

6. SHELLCODE #2: INTERRUPT HIJACKER

As Section 5 showed, the disassembling shellcode can be used reliably, at least for several major hardware platforms, to locate and intercept a critical function which handles credential verification in IOS. However, this shellcode must search through large portions of the router's memory *twice* in order to identify the target string reference, and the target function. This required computation frequently triggered the router's watchdog timer, leaving evidence of the shellcode in the router's log. In general, we want to minimize the amount of computation required by the first-stage shellcode to evade the watchdog timer, and avoid any perceivable CPU spike or performance degradation.

6.1 First-stage shellcode

The interrupt hijacking shellcode performs a single scan through the router's .text section, locating and intercepting the end of *all* interrupt handler routines, and works as follows:

[2]A packet is punted to a router's CPU when it cannot be handled by its linecards, and must be inspected and process switched.

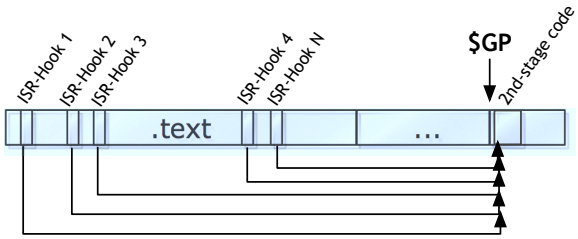

Figure 4: The interrupt hijack shellcode first locates all *eret* (exception return) instructions within IOS's .text section. The second-stage rootkit is then unpacked inside the $gp memory area (which is unused by IOS). All *eret* instructions, and thus all interrupt service routines are hooked to invoke the second-stage code. We now have reliable control of the CPU by intercepting all interrupt handlers of the victim router.

Unpack second-stage: The second-stage shellcode, which contains a basic rootkit, is unpacked and copied to the location pointed to by **$gp**, the general purpose register.

Locate ERET instructions: Scan through memory, looking for all [**eret**] instructions. All such addresses are stored and exfiltrated for offline fingerprinting (See Section 7).

Intercept all interrupt handler routines: Hijack all interrupt handler routines by replacing all **eret** instructions with the [**jr $gp**] instruction.

The **eret**, or *exception return* instruction takes no operands, and is represented by the 32-bit value [0x42000018]. As the name suggests, eret is the last instruction in any interrupt handler routine, and returns the CPU context back to the previous state before the interrupt was serviced. Once intercepted, any interrupt serviced by the CPU will invoke our second-stage code, giving us persistent, perpetual control of the CPU to execute our second-stage rootkit.

6.2 Second-stage shellcode

The second-stage is essentially a simple code loader which continuously monitors the router's IOMEM range, looking for incoming packets with a specific format. The second-stage rootkit locates packet payloads marked with a 32-bit magic-number. Such packets contain a 4-byte target address value, a 1-byte flag and variable length data (up to the MTU of the network).

When such a packet is found, the second-stage either copies the variable length data to the 4-byte memory location as indicated by the packet, or jumps the PC to a specified location. In order to load such packets into the victim router's IOMEM, the attacker simply needs to craft IP packets which will be *punted* to the router's CPU. Any packets which must be inspected by the router's control-plane will suffice.[3] For demonstration purposes, we used a variety of UDP and ICMP packets. Such packets need not even be destined to the router's interface. Various malformed broadcast and multicast packets are automatically punted to CPU and copied to the router's IOMEM region (on the 7200 platform).

When the first-stage shellcode completes, the attacker has:

Host fingerprint: The list of *eret* addresses is exfiltrated to the attacker, and will uniquely identify the micro-version of the victim's IOS (See Section 8).

Perpetual CPU control: The second-stage code, copied to the global-scope memory area, is invoked each time an interrupt is serviced by IOS.

We now present a second-stage rootkit which monitors all incoming packet-data entries, or payloads of packets which have been *punted* to the router's control-plane for process switching, continuously scanning incoming packets for commands from the attacker. Using the second-stage rootkit presented below, the attacker can load and execute arbitrary code by crafting command and control packets in the payload of *any* IP packet which will be punted to the router's CPU. The attacker can stealthily assemble large programs within the router's memory by using a wide spectrum of different packet types, like ICMP, DNS, mDNS, etc.

Since we intercept *all* interrupt handlers, the second-stage code is invoked whenever *any* interrupt is serviced, including timer interrupts, interrupts from linecards, etc. Therefore, a very limited amount of computation (under a hundred instructions) can be done inside interrupt handlers without seriously impacting the router's stability and performance. Figure 5 illustrates a second-stage rootkit that is designed specifically for high-frequency execution within interrupt handlers. Each time the second-stage code is invoked, the rootkit scans through the linked-list of *packet data* entries located within IOMEM. Figure 6 shows a snapshot of this data structure in IOMEM. Each time the second-stage code is invoked, it scans through a fixed number of packet-data entries, looking for specially marked packets containing a 32-bit magic number. The number of packet data entries scanned at each iteration directly impacts the reliability of this method (See Section 8).

Once such an entry is found, the second-stage code does the following:

Parse OpCode: Parse the packet data entry, looking for a 1-byte opcode, along with a 4-byte target address value.

If OpCode = Load: The second-stage code will copy the content of the remainder of the packet-data entry to the 4-byte address indicated by the packet.

If OpCode = Run: The second-stage code will jump the PC to the target address indicated by the packet.

[3]Different router platforms have different packet handling capabilities, trying to reduce the number of packets that must be punted to CPU. However, packets destined to routing processes, like BGP, OSPF, along with ICMP and SNMP packets are typically punted to CPU.

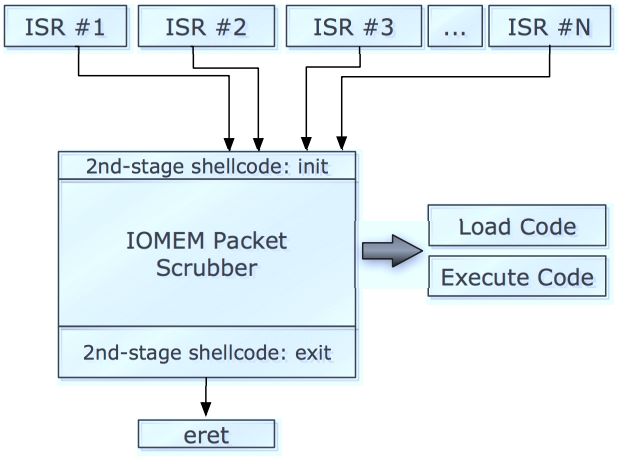

Figure 5: Interrupt hijack second-stage rootkit. Each time any ISR (interrupt service routine) is invoked, the rootkit will seek through the latest punted packets within IOMEM for specially crafted command and control packet payloads.

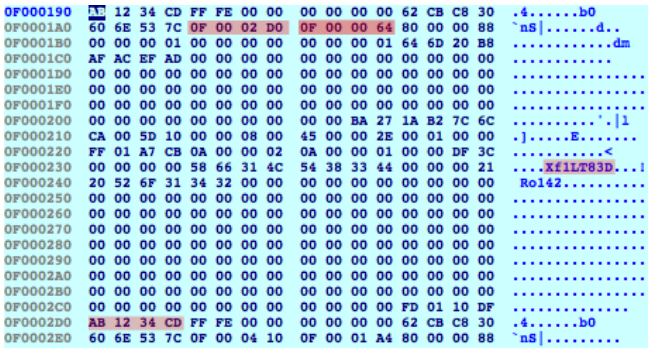

Figure 6: Highlighted words, left to right, top to bottom. 1: Pointer to previous packet data node. 2. Pointer to next packet data node. 3. Exfiltration request magic pattern. 4. Beginning of next packet data entry, pointed to by 2.

The second-stage code is designed to execute with high frequency, but in small bursts. It will execute approximately 100 instructions each time it is invoked, which allows us to scan through several *dozen* packets before returning control of the CPU back to the interrupt handler, and thus the preempted IOS code.

Note that the head of the packet-data linked-list structure is located in a well-known address within the IOMEM region, which is mapped to the same virtual-memory address regardless of router model or IOS version [6], making this packet-scrubbing technique reliable across all IOS versions on many router platforms.

The intercept hijacking shellcode has several interesting advantages over existing rootkit techniques. First, the command and control protocol is built into the payload of incoming packets. No specific protocol is required, as long as the packet is punted to the router's control-plane. This allows the attacker to access the backdoor using a wide gamut of packet types, thus evading network-based intrusion detection systems. Hiding the rootkit inside interrupt handlers also allows it to execute *forever* without violating any watchdog timers. Furthermore, the CPU overhead of this shellcode will be distributed across a large number of random IOS processes. Unlike with shellcodes which take over a specific process, the network administrator cannot detect unusual CPU spikes within any particular process using commands like *show proc cpu*, making it very difficult to detect by conventional means.

The video demonstration of the interrupt hijack shellcode running on a Cisco 7204 router and 12.4T IOS can be found at [7].

7. STEALTHY DATA EXFILTRATION

After the first-stage shellcode completes, it yields a sequence of memory addresses where the *eret* instruction is located. As Section 8 shows, this data can serve as a host fingerprint, allowing the attacker to identify the exact micro-version of the victim's IOS firmware. Several known methods can be used to exfiltrate this fingerprint back to the attacker. Note that the entire memory sequence need not be transmitted, as a simple hash of the data will suffice. The attacker can carry out a VTY binding [16] to open a reverse shell back to the attacker, or simply use the console connection to generate an ICMP packet back to the attacker. Depending on which services are publicly accessible on the router, the attacker can inject the fingerprint data into the server response. For example, the HTTP server's default HTML can be modified.

These methods will most likely leave some detectable side-effect which can trigger standard network intrusion detection system. We present a new exfiltration technique which modifies the payload content of process-switched packets just prior to transmission. The data is exfiltrated using packets generated by router itself, thus making the detection of this covert channel more difficult.

Once a packet is punted to the router's control-plane, it is copied from the network interface hardware to the router's IOMEM region. For efficiency, when such a packet is process switched, the packet-data entry is not copied. Instead, the pointer to this data is simply moved from the router's RX queue to its TX queue. Once there, the packet is scheduled for transmission, then forwarded appropriately. If the attacker can *modify* the contents of the packet-data entry before it is transmitted, such payloads can be used as a vehicle for stealthy exfiltration. Figure 7 illustrates this exfiltration process.

This type of manipulation is highly time-sensitive, as the attacker will typically only have a few milliseconds after the packet's arrival to locate and manipulate its payload, before the packet is transmitted. However, since the second-stage rootkit is invoked with *every* interrupt, it can precisely in-

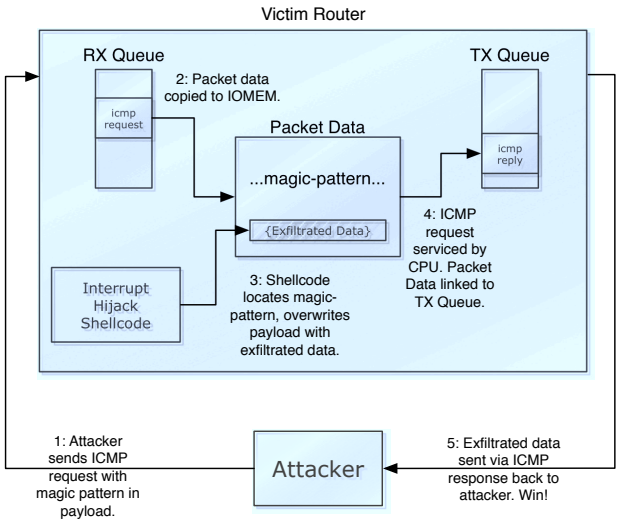

Figure 7: Data exfiltration through forwarded packet payload. 1: The attacker crafts a packet with a magic pattern in its payload indicating exfiltration request. 2: Packet payload is copied into a *packet data* structure. 3: Rootkit locates magic pattern, overwrites remaining packet with exfiltrated data. 4: Packet is process-switched. The packet data entry is linked to the TX queue. 5: The requested data is sent back to the attacker inside an ICMP response packet.

tercept the desired packet before it is placed on the TX queue, allowing the attacker to use the same covert command and control channel for data exfiltration. Section 8 discusses the performance of this exfiltration method. Due to the timing constraints of the interrupt hijack shellcode and various race conditions related to process-switching and CEF, not all exfiltration requests sent by the attacker will be processed. In practice, approximately 10% of exfiltration requests are answered by the rootkit when tested on an emulated 7204VXR/NPE-400 router.

The video demonstration of this exfiltration method can be found at [7].

8. EXPERIMENTAL DATA

The reliability of the disassembling shellcode, presented in Section 5 and the interrupt hijack shellcode, presented in Section 6, are shown in Table 1. Three major Cisco router platforms, the 7200, 3600 and 2800 series routers are tested. The two proposed shellcode algorithms are tested against 159 IOS images, ranging from IOS version 12.0 to 15.

The computational overhead of both shellcodes are shown in Figure 10 for a typical 7200 IOS 12.4 image. In some instances, the disassembling shellcode did not terminate in time, which triggered a watchdog timer exception to be thrown and logged (See Figure 11). The interrupt hijack shellcode consistently completed first-stage execution without triggering any watchdog timer exception.

	Hardware Platform	Sample Size	Reliability
xref	7200	76	100%
eret	7200	76	100%
xref	3600	52	100%
eret	3600	52	100%
xref	2800	31	0%
eret	2800	31	100%

Table 1: Reliability of the disassembling shellcode and interrupt hijack shellcode when tested on 159 IOS images.

	2	4	8	16	32	64
reliability	0%	0.67%	1.29%	4.67%	5.38%	10.10%

Table 2: Reliability of exfiltration mechanism when the number of packet-data nodes searched per invocation varies. Searching more than 64 nodes caused the test router to behave erratically.

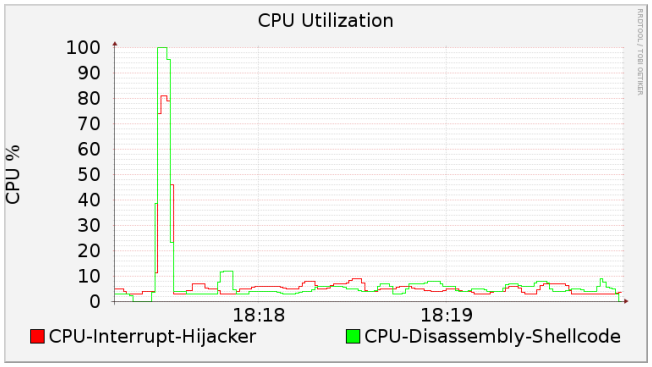

Figure 10: CPU utilization of 7204 router during the first-stage execution of both the disassembling and intercept hijack shellcodes. Note that the interrupt hijack shellcode is simpler, requires less CPU and thus avoids watchdog timer exceptions.

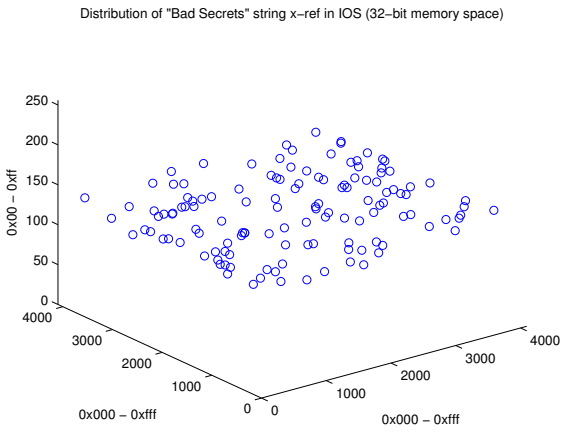

Figure 8: Distribution of the location of the password authentication function. This location varies greatly across the IOS .text segment, forcing the disassembling shellcode to search a large region.

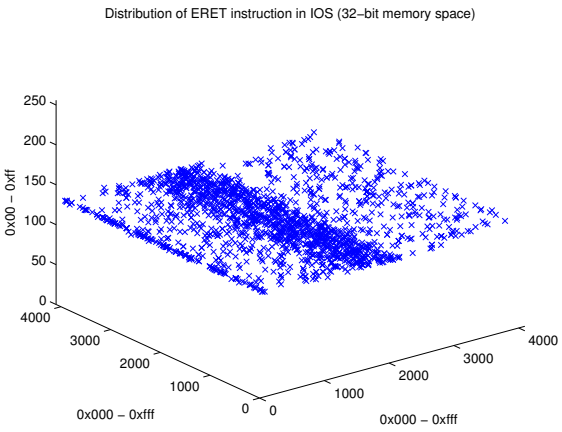

Figure 9: Distribution of the location of *eret* instructions over 162 IOS images. These locations mark the end of all interrupt service routines in IOS, and tend to be concentrated within a predictable region of IOS.

```
Router>
*May  1 16:22:56.599: %SYS-3-CPUHOG: Task is running for (2020)msecs,
 more than (2000)msecs (3/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:22:58.599: %SYS-3-CPUHOG: Task is running for (4020)msecs,
 more than (2000)msecs (3/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:23:00.603: %SYS-3-CPUHOG: Task is running for (6020)msecs,
 more than (2000)msecs (4/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:23:02.599: %SYS-3-CPUHOG: Task is running for (8012)msecs,
 more than (2000)msecs (5/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:23:03.103: %SYS-3-CPUYLD: Task ran for (8516)msecs, more t
han (2000)msecs (5/2),process = Exec
```

Figure 11: CPU intensive shellcodes will be caught by Cisco's watchdog timer, which terminates and logs all long running processes. The disassembling shellcode, although reliably bypasses password verification, consistently triggered the watchdog timer, generating the above logs, which give precise memory location of the shellcode.

Table 2 shows the reliability of the exfiltration mechanism presented in Section 7, as the number of packet-data nodes searched during each interrupt-driven invocation. The reliability rate is calculated by counting the number of exfiltration requests the rootkit successfully answered out of 150 ICMP requests. Searching more than 64 nodes at each invocation caused the router to behave erratically, sometimes leading to crashes.

Figure 8 and 9 shows the distribution of features found by the disassembling shellcode and interrupt hijack shellcode respectively across 159 tested IOS images. Note that while the string reference tends to be more widely distributed, interrupt handler routines are typically found in a much smaller area. While the exact location of interrupt handlers still remain unpredictable, this concentration allows the interrupt hijack first-stage shellcode to search through a relatively small range of memory when compared to the disassembling shellcode.

9. DEFENSE

In order to categorically mitigate against the offensive techniques described in this paper, host-based defenses must be introduced into the router's firmware. Since persistent rootkits must modify portions of the router's code, a self-checksumming mechanism can be injected into IOS to detect and prevent unauthorized modification of IOS itself. This can be generalized to all regions of the router which should remain *static* during normal operation of the router, and can include large portions of the .data, ROMMON, and .text sections.

Such a defensive mechanism, called Symbiotic Embedded Machines, have been proposed by the authors to solve this problem [8]. We have shown that Symbiotes can be injected into Cisco IOS in a version-agnostic manner to provide continuous integrity validation capability to the host router. Our experimental results show that such Symbiotes can detect unauthorized modification to any static region of IOS in approximately 300ms. Symbiotic defenses of this type is the focus of ongoing research.

10. CONCLUSION

We present a two-stage attack strategy against Cisco IOS, as well as two unique multi-stage shellcodes capable of reliable execution within a large collection of IOS images on different hardware platforms. The disassembling shellcode, first proposed by Felix Linder [12] operates by scanning through the router's memory, looking for a string reference, allowing the attacker to disable authentication on the victim router. The interrupt hijack shellcode injects a second-stage shellcode capable of continuously monitoring incoming *punted* packets for specially crafted command and control packets from the attacker. The attacker can use this covert backdoor by sending a wide gamut of packet types, like ICMP and UDP, with specially crafted payloads. In both shellcodes, when the first-stage completes execution, a host fingerprint is computed and exfiltrated back to the attacker. Using this data, the attacker can accurately identify the exact micro-

version of IOS running on the host router. Using the second-stage rootkit, the attacker can then upload a version specific rootkit, which can be pre-made *a priori* for *all* IOS images, onto the victim router. This two-stage attack scenario allows the attacker to compromise any vulnerable IOS router as if the specific version of the firmware is known, bypassing the software diversity hurdle which has obstructed the reliable, large-scale rootkit execution within Cisco routers.

11. ACKNOWLEDGEMENTS

This work was partially supported by DARPA Contract, CRASH Program, SPARCHS, FA8750-10-2-0253.

12. REFERENCES

[1] kaiten.c IRC DDOS Bot. http://packetstormsecurity.nl/irc/kaiten.c.

[2] Injunction Against Michael Lynn. http://www.infowarrior.org/users/rforno/lynn-cisco.pdf.

[3] The End of Your Internet: Malware for Home Routers, 2008. http://data.nicenamecrew.com/papers/malwareforrouters/paper.txt.

[4] Network Bluepill. Dronebl.org, 2008. http://www.dronebl.org/blog/8.

[5] New worm can infect home modem/routers. APCMAG.com, 2009. http://apcmag.com/Content.aspx?id=3687.

[6] Vijay Bollapragada, Curtis Murphy, and Russ White. Inside cisco ios software architecture. Cisco Press, 2000. Demonstration of Hardware Trojans.

[7] Ang Cui. http://www.hacktory.cs.columbia.edu/ios-rootkit.

[8] Ang Cui and Salvatore J. Stolfo. Generic Rootkit Detection for Embedded Devices Using Parasitic Embedded Machines. Technical report, Columbia University, Department of Computer Science, 2010.

[9] Ang Cui and Salvatore J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *ACSAC*, pages 97–106. ACM, 2010.

[10] Andy Davis. Cisco ios ftp server remote exploit. In *http://www.securityfocus.com/archive/1/494868*, 2007.

[11] Felix "FX" Linder. Cisco Vulnerabilities. In *In BlackHat USA*, 2003.

[12] Felix "FX" Linder. Cisco IOS Router Exploitation. In *In BlackHat USA*, 2009.

[13] Michael Lynn. Cisco IOS Shellcode, 2005. In BlackHat USA.

[14] Sebastian Muniz. Killing the myth of Cisco IOS rootkits: DIK, 2008. In EUSecWest.

[15] Sebastian Muniz and Alfredo Ortega. Fuzzing and Debugging Cisco IOS, 2011. In Blackhat Europe.

[16] Varun Uppal. Cisco ios bind shellcode v1.0. In *http://www.exploit-db.com/exploits/13292/*, 2007.

[17] Ariel Futoransky. Viral Infections in Cisco IOS. In *In BlackHat USA*, 2009.

Target Platform	Tested IOS versions	Size
All MIPS	(12.0 - 12.4)	200 bytes

Table 3: MIPS-based disassembling rootkit statistics.

Target Platform	Tested IOS versions	Size
All MIPS	(12.0 - 12.4)	420 bytes

Table 4: MIPS-based interrupt hijack rootkit statistics.

APPENDIX

A. DISASSEMBLING SHELLCODE

Source code is available to reputable researchers upon formal request.

B. INTERRUPT HIJACKING SHELLCODE

Source code is available to reputable researchers upon formal request.