From Prey To Hunter[*]

Transforming Legacy Embedded Devices Into Exploitation Sensor Grids

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

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

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

ABSTRACT

Our global communication infrastructures are powered by large numbers of legacy embedded devices. Recent advances in offensive technologies targeting embedded systems have shown that the stealthy exploitation of high-value embedded devices such as router and firewalls is indeed feasible. However, little to no host-based defensive technology is available to monitor and protect these devices, leaving large numbers of critical devices defenseless against exploitation. We devised a method of augmenting legacy embedded devices, like Cisco routers, with host-based defenses in order to create a stealthy, embedded sensor-grid capable of monitoring and capturing real-world attacks against the devices which constitute the bulk of the Internet substrate. Using a software mechanism which we call the Symbiote, a white-list based code modification detector is automatically injected *in situ* into Cisco IOS, producing a fully functional router firmware capable of detecting and capturing successful attacks against itself for analysis. Using the Symbiote-protected router as the main component, we designed a sensor system which requires no modification to existing hardware, fully preserves the functionality of the original firmware, and detects unauthorized modification of memory within 450 ms. We believe that it is feasible to use the techniques described in this paper to inject monitoring and defensive capability into existing routers to create an *early attack warning system* to protect the Internet substrate.

1. INTRODUCTION

The Internet is a dynamically changing network of many different kinds of devices, predominantly general purpose hosts and servers connected by a large collection of specialized embedded devices. Embedded devices such as routers, switches and firewalls constitutes the Internet's communication substrate. Devices such as VoIP, IPTV, power management and physical access control units provide a myriad of other specialized services. Most host-based security technologies deployed today are designed primarily to protect general purpose servers and hosts, leaving vast numbers of embedded devices, the Internet substrate itself, undefended against exploitation.

We present a new embedded device defense system designed make the internet substrate a safer environment. We believe it is technically feasible to inject security functionality *in situ* into legacy embedded systems to:

1. Provide security features to protect these devices against exploitation and rootkitting.

2. Create a large scale sensor grid providing new detection capability to identify attacks against embedded devices that are currently unmonitored.

Recent studies suggest that large populations of vulnerable embedded devices on the Internet are ripe for exploitation [8]. However, examples of successful exploits against such devices are rarely observed in the wild, despite the availability of proof-of-concept malware, known vulnerabilities and high monetization potential. We posit that our inability to monitor embedded devices for malware installation is a factor in this phenomenon. When deployed throughout the Internet substrate, the sensor system discussed in this paper will provide visibility into black-box embedded devices, allowing us to capture and analyze exploitation of embedded devices in real-time.

As a first step to show feasibility, we demonstrate a general method of transforming existing legacy embedded devices into exploitation detection sensors. We use Cisco firmware and hardware as the main demonstrative platform in this paper. However, the techniques described are not specific to any particular operating system or vendor, and can be directly applied to many other types of embedded devices.

In order to detect and capture successful attacks against Cisco routers for analysis, we engineered a system which automatically injects generic whitelist-based anti-rootkit functionality into standard IOS firmwares. Once injected, the augmented router firmware can be loaded onto physical Cisco routers, essentially transforming such devices into highly interactive router honeypots. As Section 8 shows, the resulting devices are fully functional, and can be deployed into production environments.

[*]Please note that Figures 1 and 2, along with portions of Section 5 is taken from a companion paper [9], and are present here so that our exposition has the appropriate background and completeness.

The main challenge of creating an embedded device honeypot rests with the difficulties of injecting arbitrary detection code into proprietary, close-source, embedded devices with complex and undocumented operating systems. In order to overcome this challenge, we've created a software constructed called the Symbiote [9]. As Section 5 illustrates, the Symbiote, along with its payload, is injected *in situ* into an arbitrary host binary, in this case, Cisco IOS. The injection is achieved through a generic process which is agnostic to the operating environment of the host program. Figure 1 shows how a Symbiote is typically injected into a host program. In general, Symbiotes can inject arbitrary host-based defenses into black-box embedded device firmwares. For a full discussion of Symbiotes, please see [9]

The unique capabilities of the Symbiote construct allows us to overcome the complexities of injecting generic exploitation detection code into what is essentially an unknown black-box device. The original functionality of resulting Symbiote-injected embedded device firmware remains unchanged. A portion of the router's computational resources is diverted to a proof of concept Symbiote payload, which continuously monitors for unauthorized modifications to any static areas within the router's memory address space, a key side-effect of rootkit installation. As we demonstrate in Section 9, the portion of the CPU diverted to the Symbiote's payload is a configurable parameter, and directly effects the performance of the Symbiote payload; in this case, the detection latency of any unauthorized modification.

A monitoring system is constructed around the main component of our system, the Symbiote-injected IOS image. The Symbiote within the IOS firmware simultaneously performs checksums on all protected regions of the router's memory while periodically communicating with an external monitor via a covert channel. In the event of an unauthorized memory modification within the router, the Symbiote will raise an alarm to the external monitor, which then triggers the capture and analysis component of our system.

As Section 8 discusses, our monitoring system can be deployed in one of three ways; native deployment, emulated deployment, and shadow deployment. Due to the unique limitations of each deployment scenario, the capture and analysis mechanisms differ slightly. For example, when the Symbiote-injected firmware image is loaded into a physical Cisco router (native deployment), IOS's own core dump mechanism is used to capture the router's runtime state for analysis. This is less than ideal because, due to the hardware constraint of the Cisco device, we can not guarantee that the memory capture is performed atomically. Furthermore, since the core dump is generated by IOS's own (potentially compromised) code, the integrity of the output can not be fully trusted. In contrast, when the Symbiote-injected firmware is executed within Dynamips, a Cisco router emulator, on a general purpose computer (emulated deployment), the external monitor triggers a response which halts emulation of the compromised IOS image before initiating a full memory dump using the general purpose host computer. Thus, emulated deployment of our sensor can guarantee that the capture and analysis process can be done atomically without relying on potentially compromised code. Section 8 discusses the tradeoffs and advantages of all three deployments in detail.

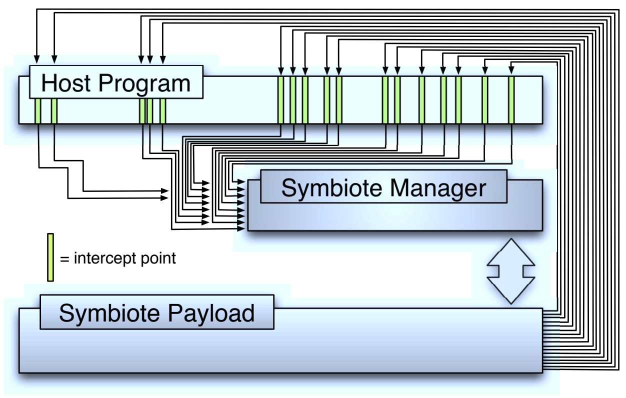

Figure 1: Logical overview of SEM injected into embedded device firmware. SEM maintains control of CPU by using large-scale randomized control-flow interception. SEM payload executes alongside original OS.

2. MOTIVATION

Several recent studies demonstrate that there are vast numbers of unsecured, vulnerable embedded devices on the internet [8], such devices are vulnerable to the same types of attacks as general purpose computers [3, 12], and can be systematically exploited in much the same way [1, 3, 5]. For example, various exploitable vulnerabilities [15, 13] and rootkits [16] have been found and disclosed for Cisco's router operating 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 employ standard protection schemes found within modern operating systems [16], 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 4 shows, the necessary techniques of exploiting Cisco IOS and installing root-kits on running Cisco routers are well understood. The works presented within academic and blackhat circles, combined with anecdotal evidence of the systematic exploitation of embedded network devices within the last decade suggests that real-world exploitation of Cisco routers is not only possible, but likely an undetected reality.

Documented cases of embedded device exploitation are still relatively rare. High-value embedded targets like enterprise networking equipment have seemingly eluded exploitation. It is possible that the exploitation of devices like Cisco routers is still beyond the technical capabilities of the blackhat community. However, it is far more plausible that stealthy, targeted attacks against high-value embedded devices have eluded detection due to our inability to gain visibility into the internals of such devices. It is quite possible routers have been successfully attacked and are compromised without anyone's knowledge except the UE sellers who offer them for sale.

Whether or not stealthy exploitation of embedded devices is a reality today, we can confidently anticipate that attacks against such defenseless, high-value targets is inevitable. Therefore, analysis and mitigation of embedded device exploitation is crucial to the integrity of the Internet substrate. We believe that accurate, real-time detection of such attacks is an important first step towards understanding the realities of the embedded security threat. Furthermore, we believe the ability to inject host-based security into existing legacy devices will be instrumental in mounting a realistic defense of existing embedded devices.

The Symbiote structure presented in this paper is designed specifically to abstract away the technical challenges of injecting third-party security into a diverse range of embedded devices. This device agnostic foundation allows us to look beyond specific hardware and firmware in order to create a general body of embedded defense methodologies which can be feasibly applied to all existing devices.

3. THREAT MODEL

We are interested in detecting, capturing and analyzing successful injection of rootkits into IOS at runtime. We assume that the attacker is technically sophisticated and has access to both zero-day vulnerabilities as well as a reliable rootkit which persistently alters the behavior of the victim device's OS, yielding covert root access to the attacker. We assume that the injected rootkit will patch specific portions of the router's code in order to create a hidden backdoor for the attacker. In other words, we assume that the rootkit will alter regions of memory within the router that is meant to be **static** during normal execution. Static sections within IOS firmware image typically include the .txt, .rodata, .firmware, .sdata and large portions of the .data sections. Furthermore, the boot-loader (rommon) section of the router, as well as all associated configuration files (startup configuration, running configuration, etc) can also be monitored for unauthorized modification.

While our current threat model encompasses all published IOS rootkitting techniques to date, it is probable that a covert backdoor can be created within an IOS router without modifying static regions of the router's memory. The proposed detection payload will not detect exploits which leave no persistent change within the victim device. However, the Symbiote-based injection scheme described in this paper can be extended to monitor for anomalies within dynamic sections of the target device, extending our whitelist-based detector into a full-blown host-based anomaly detector (See Section 10).

Furthermore, it is possible for sophisticated attacks to attempt to disable the Symbiote prior to the actual exploitation of the victim device. Since the Symbiote structure described in this paper is a software-based defense, absolute integrity of the Symbiote cannot be guaranteed. In the general case, Symbiotes can be fortified with the introduction of specialized hardware. However, such a solution is not feasible when considering the realm of legacy embedded devices. Instead, Section 6 illustrates a general method of increasing the computational complexity of a successful bypass of the Symbiote defense without relying on additional hardware.

4. RELATED WORK

Relatively little work has been done to detect and capture sophisticated attacks against embedded devices. However, such problems have been well studied for general purpose computers and operating systems. Numerous rootkit and malware detection and mitigation mechanisms have been proposed in the past but largely target general purpose computers. Commercial products from vendors like Symantec, Mcafee/Intel, Kapersky and Microsoft [2] all advertise some form of protection against kernel level rootkits. Kernel integrity validation and security posture assessment capability has been integrated into several Network Admission Control (NAC) systems. These commercial products largely depend on signature-based detection methods and can be subverted by well known methods [18, 19, 20]. Sophisticated detection and prevention strategies have been proposed by the research community. Virtualization-based strategies using hypervisors, VMM's and memory shadowing [17] have been applied to kernel-level rootkit detection. Others have proposed detection strategies using binary analysis [11], function hook monitoring [23] and hardware-assisted solutions to kernel integrity validation [22].

The above strategies may perform well within general purpose computers and well known operating systems but have not been adapted to operate within the unique characteristics and constraints of embedded device firmware. Effective prevention of binary exploitation of embedded devices requires a rethinking of detection strategies and deployment vehicles.

Our methodology transforms standard legacy embedded devices into exploitation detectors. This is similar to existing honeypot-based IDS strategies, which generally involves the use of intentionally vulnerable systems to log, capture and analyze attacks levied against it. Many honeypot-based systems have been proposed. Few focus on the use or protection of embedded devices. For example, Ghourabi *et al.* recently proposed the use of simulated honey routers to study protocol attacks against BGP [10].

In general, honeypots can be native, emulated or simulated, and can involve a single machine or a vast network of simulated nodes. Many off-the-shelf honeypot systems exist for general purpose computers. However, such systems are not without flaws. For example, simulated honeypots disguises themselves as vulnerable systems but does not expose any actual vulnerabilities to the attacker. Therefore, minimizing false-positives in such systems is a challenge. Furthermore, simulated honeypots may catch indiscriminate exploitation attempts, but will rarely fool sophisticated attackers in highly targeted attacks. Thus, native and emulated honeypots which exposes real vulnerabilities to the attacker are much better suited for detecting sophisticated, targeted attacks.

Guards, originally proposed by Chang and Atallah [6], is another technology which uses mechanisms of action similar to Symbiotes. A Guard is a simple piece of security code which is injected into the protected software using binary rewriting techniques similar to our Symbiote system. Once injected, a guard will perform tamper-resistance functionality like self-checksumming and software repair. However, Guards have no mechanism to pause and resume its computation, the entire Guard routine must complete execution each time it is invoked. This limits the sophistication of what each Guard can realistically perform, especially when Guards are used in time sensitive software and real-time embedded devices.

Devices like Cisco routers are black-box systems utilizing large numbers of undocumented proprietary hardware components. The injection of new code into proprietary firmware and the emulation of specialized and undocumented hardware makes the creation native and emulated honeypots for embedded devices challenging. As Section 5 describes, the unique capabilities of the Symbiote construct allows us to overcome the above challenges in order to transform standard Cisco IOS firmware and hardware into highly believable native router honeypots.

5. MEET SYMBIOTE

For a full discussion of Symbiotic Embedded Machines, please see [9]. The Symbiote is a software construct that is injected *in situ* into a host program to provide the following four fundamental security properties.

1. The Symbiote has full visibility into the code and execution state of its host program, and can either passively monitor or actively react to the observed events at runtime.
2. The Symbiote executes along side the host software. In order for the host to function as before, it's injected Symbiote must execute, and vice versa.
3. The Symbiote is an autonomous entity which is hardened to defend against unauthorized modification or removal once it is injected into the host program.
4. No two instantiations of the same Symbiote are the same. Each time a Symbiote is created, its code is randomized and mutated, rendering signature based detection methods and attacks requiring predictable memory and code structures within the Symbiote ineffective.

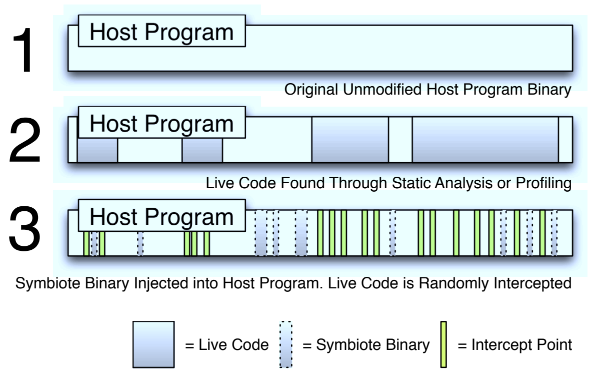
1
Host Program
Original Unmodified Host Program Binary
2
Host Program
Live Code Found Through Static Analysis or Profiling
3
Host Program
Symbiote Binary Injected into Host Program. Live Code is Randomly Intercepted
= Live Code
= Symbiote Binary
= Intercept Point

Figure 2: Symbiote Injection Process.

Figure 1 shows the three logical components of Symbiotes: Control-Flow Interceptors, Symbiotic Embedded Machine Manager (SEMM) and the Symbiote Payload. Together, all three components are injected *in situ* into the target embedded device firmware. Since the Symbiote is injected *in situ*, the size of the resulting firmware image is unchanged. For example, the current implementation of the Symbiote Manager, along with the rootkit detection payload requires only approximately 1600 bytes to be injected into IOS.

Figure 2 illustrates the three step Symbiote injection process. First, analysis is performed on the original host program in order to determine areas of live code, or code that will be run with high probability at runtime. Second, intercept points are chosen randomly from the host program. Lastly, the Symbiote Manager, Symbiote payload and a large number of control-flow intercepts are injected into the host program binary, yielding a Symbiote protected host program.

The Symbiote randomly intercepts a large number of functions as a means to divert periodically and consistently a small portion of the device's CPU cycles to execute its payload. This approach allows the Symbiote to remain agnostic to operating system specifics while executing its payload alongside the original OS. The Symbiote payload has full access to the internals of the original OS but is not constrained by it. This allows the payload to carry out functionality which might not be possible under the original OS. In the case of Cisco IOS for example, a process watchdog timer will forcibly terminate any process which executes for more than several seconds. However, since the Symbiote payload executes in time-slices randomly distributed throughout many unrelated processes, the Symbiote payload can execute indefinitely, circumventing the watchdog timer entirely.

Stealth is a byproduct of the SEM structure. In the case of IOS, no diagnostic tool available within the OS (short of a full memory dump) can detect the presence of the SEM payload because it manipulates no OS specific structure and is effectively invisible to the OS. The impact of the SEM payload is further hidden by the fact that CPU utilization of the payload is not reported within any single process under IOS and is distributed randomly across a large number of unrelated processes.

Once the Symbiote Manager gains control of the CPU, it allocates a certain number of cycles for the execution of its Symbiote payload (in this case, a checksuming mechanism). After the payload completes its execution burst, control of the CPU is returned to the Symbiote Manager, which in turn resumes the execution of the original host program.

The Symbiote Manager acts as a job scheduler, treating the entire host program as one process, and its Symbiote payload as the other. Traditional scheduling strategies can be used to determine the proper CPU resource distribution between the Symbiote and its host program. In general, this involves the optimization of both the *frequency* of context switches as well as the *duration* of the Symbiote payload's execution bursts.

The proposed Symbiote payload detects unauthorized code modification through the computation of checksums over static regions of memory. Therefore, a delay exists between the time of the code modification and its detection. In general we refer to the time between the occurrence of an unauthorized event and its detection as the *detection latency.* Intuitively, the amount of CPU resources diverted to the Symbiote payload should be inversely proportional to the detection latency, and thus directly proportional to the performance of our detector. In the case of Cisco IOS, and many other embedded systems, an over allocation of CPU resources to the Symbiote can adversely affect the performance of the protected host device. In practice, we have found that it is beneficial to frequently interleave the host program's execution with short Symbiote payload execution bursts. This allows the Symbiote payload to compute at acceptable rates while minimizing the impact on the real-time nature of Cisco routers.

The Symbiote scheduling problem is arguably simple as it involves only two "tasks". However, performing such a task safely in an OS agnostic manner on embedded systems presents several interesting complexities. A full discussion of potential Symbiote scheduling algorithms is out of the scope of this paper. However, in the case of our IOS exploitation detection Symbiote, the performance and overhead characteristics of several scheduling strategies are discussed in Section 9.

6. SELF-MONITORING SYMBIOTES

We must consider ways to protect the Symbiote itself against attack and removal. The polymorphic nature of the Symbiote and its payload makes signature-based attacks against it ineffective. To further raise the bar, multiple Symbiotes within a protected host program can be configured in a self-monitoring monitor arrangement. As proposed by Stolfo, Greenbaum and Sethumadhavan [21], a network of monitors can be constructed, such that an alarm will be raised if any subset of monitors are compromised or deactivated, or if any critical condition monitored by the system is violated. Consider Figure 3, which shows three independent Symbiotes arranged in a full-mesh monitoring network. In this arrangement, each Symbiote monitors a specific critical condition, i.e., the output of their Symbiote Payload, while simultaneously monitoring the operational status of the other two Symbiotes within the network. If one or more of the Symbiotes are corrupted or disabled, the remaining Symbiotes within the network will raise an alarm. Similarly, if all three Symbiotes are simultaneously deactivated, an external sensor can also detect this event and raise an alarm. Note that the three Symbiotes shown in Figure 3 need not be located within the same host router. Large networks of embedded device sensors can be collectively protected in this mutually defensive arrangement. Using Symbiotes in this fashion is a topic of ongoing research.

7. EXPLOITATION DETECTOR

IOS rootkit and malware code is generally not publicly available. However, a survey of published persistent rootkit techniques reveals a commonality in their *modus operandi*. Specifically, rootkits such as [16, 13, 7] all modify some region of static IOS memory in order to inject their rootkit payload into the victim router. Thus, we implemented a white-list strategy to detect IOS malcode and rootkits described previously.

Known rootkits operate by hooking into and altering key functions within IOS. To do this, specific binary patches must be made to executable code. Therefore, a continuous integrity check on all **static areas** of Cisco IOS will detect all function hooking and patching attempts made by rootkits and malware. The rootkit detection payload described below is not specific to IOS, and can be used on other embedded operating systems as well. As Section 9 shows, our Symbiote payload accurately detects unauthorized modification of any monitored region of memory within milliseconds, and will accurately detect [16, 13, 7] immediately after successful exploitation of the victim device.

In the case of Cisco IOS, several large contiguous segments of the router's memory address space can be monitored using the checksumming mechanism described above. Figure 4 illustrates the memory layout of a typical IOS firmware image on a Cisco router. The darkened regions represent areas of the router's firmware which can be safely monitored by our checksumming mechanism. For example, regions containing executable code (text and firmware), and static data (rodata, ctors, sdata sections) should clearly not be modified at runtime. In practice, the typical IOS firmware contains large contiguous sections of memory which should semantically remain static during the normal operation of the router.

8. DESIGN AND OPERATION

Our sensor system has three components; a Symbiote-protected router, a monitoring station, and a capture and analysis system which automatically collects and analyzes forensics data once an alarm is triggered. The Symbiote within the IOS firmware simultaneously performs checksums on all protected regions of the router's memory while periodically communicating with an external monitor via a covert channel. In the event of an unauthorized memory modification within the router, the Symbiote will raise an alarm to the monitor, which then triggers the capture and analysis component of our system.

The proposed exploitation detection sensor can be deployed in one of at least three ways; natively, emulated within a general purpose computer, or as a shadow replica for a production device. The implementation of the monitoring station and capture and analysis engine changes depending on how the Symbiote-injected router firmware is executed; natively on embedded hardware or emulated on a general purpose computer.

When deployed natively, the monitor and capture components are integrated into the Symbiote payload and injected directly into Cisco hardware, producing a standalone sensor. When the detection payload raises an alarm, the Symbiote immediately triggers the core dump functionality from within IOS. This causes the bulk of the router's execution state to be captured and transferred via FTP or TFTP.

When deployed as an emulated sensor, using Dynamips for example, the monitoring and capture components of the sensor are implemented within the emulator. This reduces the footprint of the Symbiote and allows us to perform more sophisticated capture and analysis on the server running the emulation. For example, Dynamips was modified to continuously monitor a region of the router's memory for an encoded marker, which is set by the Symbiote payload only when an alarm is raised.

For testing purposes, we chose to modify a portion of the text that is printed when the "show version" command is invoked. In practice, many better covert channels can be used to communicate between the Symbiote and the router emulator.

In order to transform large populations of embedded devices into massive embedded exploitation sensor-grids, the native deployment is the most efficient and practical. For the purposes of testing and validation of our approach, the emulated deployment scenario is most appropriate. The shadow

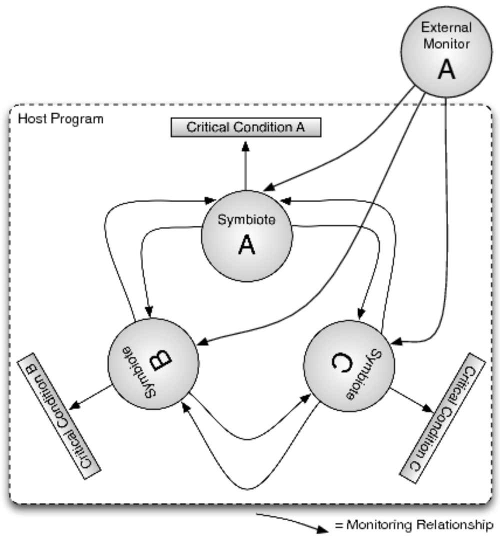

Figure 3: Full Mesh Self-Monitoring Symbiote Network

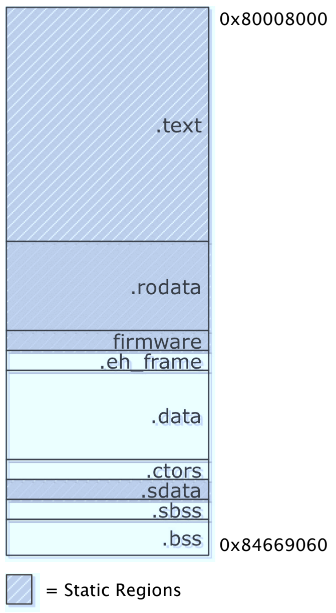

Figure 4: Memory layout of a typical Cisco IOS router

deployment is best for capturing and analyzing IOS exploits in mission critical production environments.

8.1 Native Sensor Deployment

In the native deployment scenarios, the Symbiote-injected firmware is loaded directly onto the target embedded device, i.e. a Cisco router. The Symbiote payload executes natively on the embedded hardware, alongside the original firmware. Native deployment allows the Symbiote to operate in embedded systems for which emulation is not feasible. For example, a large portion of Cisco devices can not be emulated by existing software due to the use of undocumented, proprietary hardware. In practice, most modern high-performance networking equipment falls within this category. Therefore, native deployment is most practical for injecting Symbiotic defenses into the diverse range of embedded devices found on the Internet substrate.

8.2 Emulated Sensor Deployment

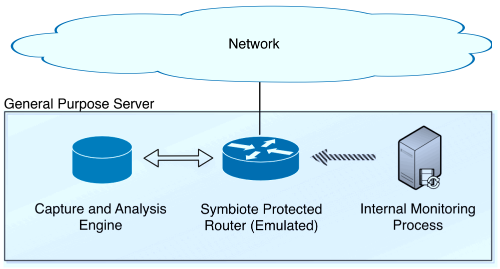
Network
General Purpose Server
Capture and Analysis Engine
Symbiote Protected Router (Emulated)
Internal Monitoring Process

Figure 5: Emulated Deployment of Symbiote-based Cisco IOS Detector

Figure 5 illustrates a typical *emulated* deployment of our sensor. Instead of running the Symbiote-injected firmware natively on embedded hardware, the firmware is emulated on a general purpose computer. In the case of Cisco IOS, Dynamips is used to emulate specific devices such as 7200 series routers used in our testing and evaluation environment. This differs from *simulated* honeypots in a significant way: the use of actual IOS firmware. The emulation of real Cisco IOS allows us to create highly interactive honeypots which exposes real IOS vulnerabilities to potential attackers.

The emulated deployment has several advantages which make it the ideal approach for developing and testing experimental prototype Symbiotes. First, debugging proof of concept Symbiotes in an emulated environment is slightly more convenient than doing so on native embedded hardware. Second, the general purpose computer which hosts the emulation usually has far greater computational capacity than the embedded hardware which it is emulating. Therefore, computation can be offloaded from the Symbiote payload onto the general purpose host computer. This can potentially allow the Symbiote payload to perform complex computations not feasible on actual embedded hardware. The Symbiote payload presented in this paper is simple and requires relatively little CPU power. However, this can be replaced with payloads more akin to behavior based anomaly detectors which can require significantly more resources. The development of Symbiote-based anomaly detection mechanisms is an area of active research (See Section 10).

Lastly, the emulated sensor deployment can usually simplify the capture and analysis component of the sensor. In the case of the sensor presented in this paper, we modified the Dynamips emulator to *atomically* capture the entire memory state of the IOS router once the Symbiote payload emits an alarm. The Dynamips emulator conveniently allows us to halt the router's CPU briefly while the memory capture takes place on the host computer. Once this operation completes, the memory snapshot, along with all network traffic received by the router is automatically processed and archived for analysis. Once the Symbiote payload emits an alarm, our modified Dynamips emulator continuously dumps the memory state of the router at a configurable frequency.

8.3 Shadow Sensor Deployment

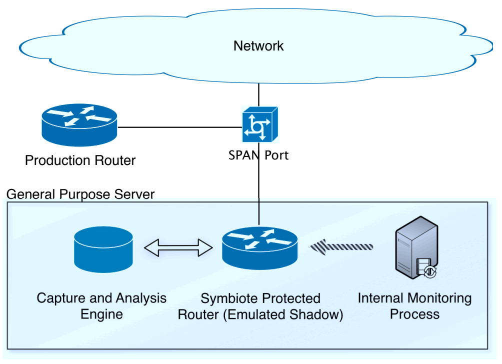
Network
Production Router
SPAN Port
General Purpose Server
Capture and Analysis Engine
Symbiote Protected Router (Emulated Shadow)
Internal Monitoring Process

Figure 6: Shadow Deployment of Symbiote-based Cisco IOS Detector

In order to detect exploitation against high-performance embedded devices within production environments, we must be able to deploy Symbiote-based sensors in a way which will not cause unintentional service outages on the monitored devices. In such cases, the use of a second, identical embedded device as a *shadow* sensor is most appropriate. Figure 6 illustrates a typical shadow sensor deployment.

As the name suggests, incoming network traffic is mirrored from the production embedded device to a Symbiote-injected shadow device, which runs the same firmware as the production device. The Symbiote sensor injected into the shadow device continuously monitors the shadow device, quietly emitting alerts when malicious activity is detected.

The performance of the shadow sensor is critical, as it must be able to keep up with the production router. Thus, minimizing the control-plane latency and computational overhead of the Symbiote is critical to the effectiveness of the detection system. We discuss preliminary performance data in the next section. The development of Symbiote-based shadow sensors is an area of active research.

9. PERFORMANCE AND OVERHEAD

We measure the performance and overhead of our Symbiote-based exploit detector using two quantitative metrics: *computational overhead* and *detection latency*. The Symbiote-protected router is an emulated Cisco 7200 series router running IOS 12.3. Two neighbor routers are used to verify that the Symbiote-protected router's original functionality is unchanged. One neighbor router is an emulated 7200 series router running standard IOS 12.3. The other neighbor router is a physical Cisco 2921 router running IOS 12.5. Each router is configured to expose a cross-section of functionality typically seen on production routers. Specifically, a large number of local loopback interfaces are configured on each router. OSPF routing is enabled on all three routers, along with a suite of standard services like IP-SLA, SNMP, HTTP{S} and SSH.

A stress-test script automatically generates network traffic throughout the test environment, and periodically accesses services on all the test routers. All routers are continuously monitored to ensure that all services operate correctly throughout testing. The workload script also periodically forces route-table re-calculations by perturbing the various OSPF routers on the network. In effect, the stress-test script simulates a typical use profile for the IOS routers in the test environment. The same stress-test script is run against several variants of the Symbiote-injected IOS firmware in order to illustrate key performance features of our system.

The computational overhead and performance of our system is a configurable parameter. As the figures in this section shows, the scheduling algorithm used within the Symbiote Manager directly impacts the resource consumption of the Symbiote payload, and thus the overall utilization of the host device as well as the detection latency. Two scheduling algorithms are discussed in this section: *fixed burst-rate* and *inverse-adaptive*.

As the name suggests, the fixed burst-rate scheduling algorithm instructs the Symbiote payload to execute for a fixed burst-rate each time the Symbiote Manager is invoked through a randomly placed execution intercept. On the other hand, the inverse-adaptive scheduling algorithm calculates the payload burst-rate based on the elapsed time since the Symbiote Manager was last invoked; the longer the elapsed time, the longer the burst-rate.

Intuitively, we can expect the fixed burst-rate scheduling algorithm to execute the Symbiote payload *more frequently* as the host system becomes more utilized. This simple algorithm executes the Symbiote payload more frequently when the Cisco router is heavily utilized, and less frequently when the router is idle. In contrast, the inverse-adaptive scheduling algorithm increases Symbiote payload burst-rate when the system is under-utilized, and throttles back the Symbiote payload when the router is under high load.

We analyze the performance of 15 Symbiote-injected IOS images under the same stress-test; 7 variants using the fixed burst-rate Symbiote scheduler and 8 variants using the inverse-adaptive Symbiote scheduler. As the next three subsections show, the fixed burst-rate Symbiote scheduler aggressively executes the Symbiote payload, and achieves the least detection latency (approximately 400 ms). However, this aggressive scheduler tends to amplify CPU utilization of the protected router, causing very high control-plane latency when the router is under load. Although the higher fixed burst-rate values like 0x7FF and 0xFFF detected IOS modification very quickly, it also caused the router's control-plane to be less responsive.

In contrast, the inverse-adaptive Symbiote scheduler produced slightly longer detection latencies (approximately 450 ms), but was able to significantly reduce the control-plane latency of the host router, even under high load.

9.1 Computational Overhead

The same stress-test script is run against various versions of the Symbiote-injected IOS image in order to show how the Symbiote Manager's scheduling algorithm affects the CPU utilization of the router. Two major scheduling algorithms are measured: fixed burst-rate (Figure 7) and inverse-adaptive (Figure 8). **Burst-rate** values presented in the next two sections represent the number of iterations of the main Symbiote payload executed each time the Symbiote Manager is invoked.

Figure 7 shows the CPU utilization of 7 variants of the *fixed burst-rate* Symbiote scheduler, which unconditionally executes the Symbiote payload for a constant number of CPU cycles each time the Symbiote is invoked via its many control-flow intercepts. The units used, burst-rate, is the number of iterations of the checksum Symbiote payload that is executed each time the Symbiote Manager is invoked.

This Symbiote scheduler disregards the current CPU utilization of the host device. At higher burst-rate values like 0x7FF and 0xFFF, the router's CPU utilization tends to remain above 95% under heavy load, causing large spikes in control-plane latency. (See Figure 11)

Figure 8 shows the CPU utilization of 8 variants of the *inverse-adaptive* Symbiote scheduler, compared with the baseline CPU utilization of the unmodified IOS image under the same stress-test. The inverse-adaptive scheduler is configured with *maximum* burst-rates from 0x1FFFF to 0xFFFFFF. Unlike the fixed burst-rate Symbiote scheduler, the inverse-adaptive scheduler throttles how much the CPU is diverted to the Symbiote based on current host device utilization. As a result, Symbiotes with inverse-adaptive schedulers can achieve comparable detection latencies while significantly reducing its impact on the host router's control-plane latency. (Compare Figure 11 and Figure 12).

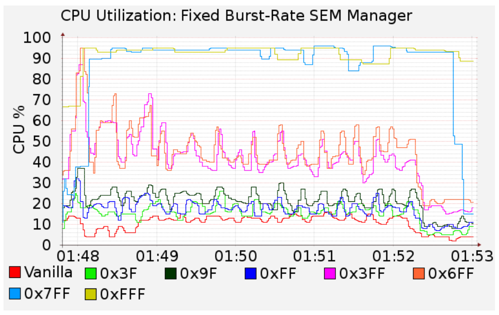

Figure 7: CPU Utilization: Fixed Burst-Rate SEM Manager

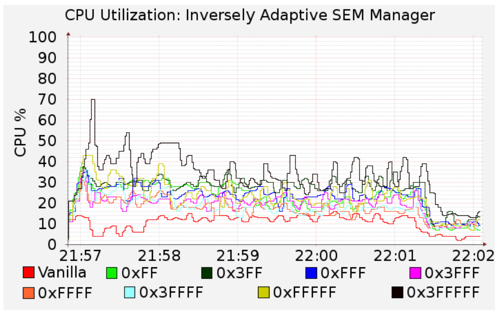

Figure 8: CPU Utilization: Inverse-Adaptive SEM Manager

9.2 Detection Performance

In order to measure the detection latency of our exploitation detection Symbiote, a simple vulnerability which allows arbitrary memory modification is artificially introduced into the Symbiote-injected IOS image. Using an automated script, this vulnerability is triggered, modifying a random byte within monitored memory regions. A timer is simultaneously started in order to measure the time it takes the Symbiote payload to detect the event.

Figure 9 shows a roughly linear relationship between the Symbiote's fixed burst-rate value and the Symbiote's detection latency. As expected, the Symbiote detection latency decreases as the Symbiote payload's execution burst-rate increases. However, as Figure 11 shows, the fixed burst-rate Symbiote scheduler causes significant increases in the router's control-plane latency.

Figure 10 shows the detection latency of Symbiotes using the inverse-adaptive scheduler. As the figure shows, these Symbiotes can achieve comparable detection latency values as the fixed burst-rate versions, but as Figure 12 shows, the Symbiote's impact on the router's control-plane is significantly reduced.

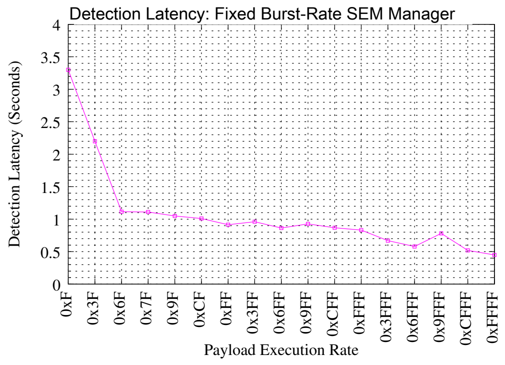

Figure 9: Detection Latency: Fixed Burst-Rate SEM Manager

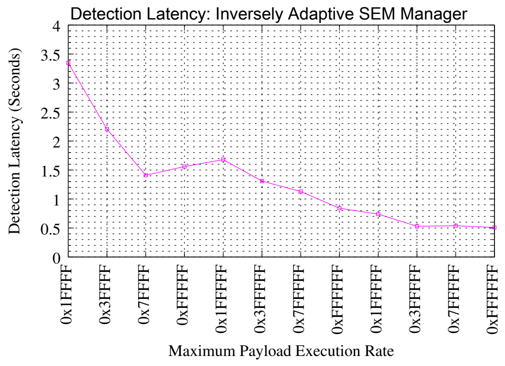

Figure 10: Detection Latency: Inverse-Adaptive SEM Manager

9.3 Control-Plane Latency

Control-plane latency is an indicator of how responsive the router is. High control-plane latency can cause a router to drop routing adjacencies and break various time-sensitive network protocols. Note, however, that this measurement will not significantly affect the latency of traffic passing through the router, as most modern routers have hardware-accelerated forwarding engines which are decoupled from the control-plane.

Control-plane latency is measured by sending ICMP-echo messages from the test PC to the router's local loopback interface. The round-trip-time is collected and shown in Figure 11 for Symbiotes using fixed burst-rate scheduler variants, and in Figure 12 for Symbiotes using inverse-adaptive scheduler variants. Clearly, the inverse-adaptive Symbiote scheduler significantly reduces the Symbiote's impact on the host router's control-plane latency while achieving comparable detection latency values as fixed burst-rate Symbiotes.

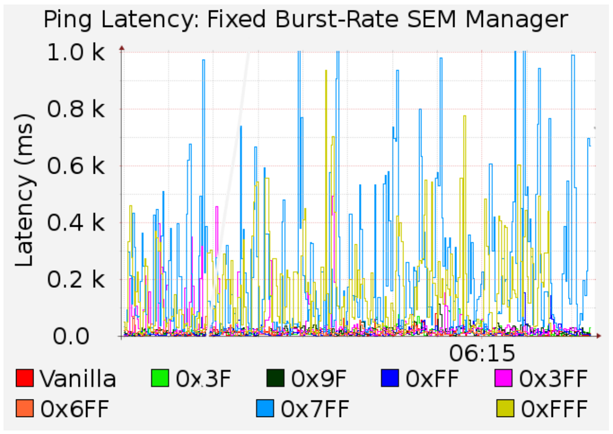

Figure 11: Ping Latency: Fixed Burst-Rate SEM Manager

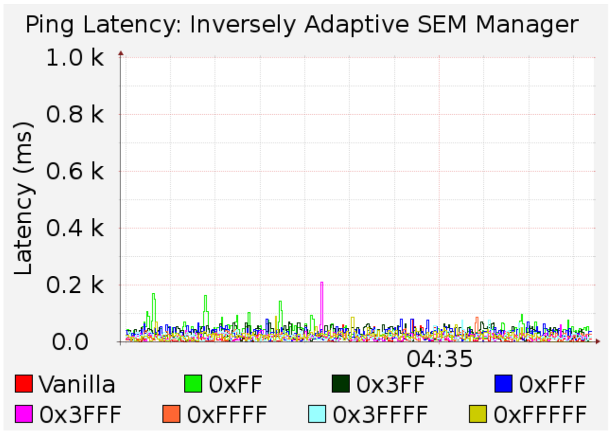

Figure 12: Ping Latency: Inverse-Adaptive SEM Manager

9.4 Discussion

Preliminary performance results shown in this section suggests that high performance exploitation detection is possible in Cisco IOS. Furthermore, an optimized Symbiote scheduling algorithm can greatly improve performance of the overall sensor system by reducing both detection latency and the Symbiote's impact on the router's control-plane latency. Optimization of the detection latency and the induced control-plane latency is an area of active research.

10. FUTURE WORK

The Symbiote-based sensor presented in this paper is a first step towards demonstrating the feasibility and novel capability of Symbiotic defense systems. The Symbiote structure allows complex payloads to be injected into legacy embedded devices, allowing the payload to safely execute alongside the original firmware without altering the embedded device's functionality. The checksumming payload we injected into Cisco IOS can be replaced with a wide range of defensive payloads. Below are several new Symbiote payloads currently under development.

10.1 Embedded Self-Healing

The checksumming Symbiote payload discussed in this paper can be extended to **reverse** unauthorized modification of memory after it is detected. A self-healing Symbiote payload can be used to identify and restore regions of memory which have been maliciously modified.

10.2 Embedded Anomaly Detector

Symbiote payloads can implement existing anomaly detection algorithms. For example, behavior modeling strategies which monitor resource utilization, control and data flow patterns can be injected into embedded devices via Symbiote payloads.

10.3 Large-Scale Embedded Sensor Grid

The exploitation detection sensor described in this paper can be injected into large numbers of embedded devices like Cisco routers in order to monitor and analyze 0-day exploitation of embedded devices. We believe the use of Symbiote-based exploitation sensors in the wild is a feasible and effective way of monitoring and analyzing exploits levied against the internet substrate. A large-scale Symbiote-based sensor grid can potentially give us real-time visibility into embedded device exploitation on a global scale.

Furthermore, Symbiotes can be used to transform embedded devices into other kinds of sensor grids as well. Symbiotes can allow us to use hardware components of embedded devices in novel ways not intended by its original design. For example, many power-consuming, EM emitting components can be transformed into covert communication channels. Existing sensors on embedded devices, combined with such covert channels can transform a wide gamut of innocuous embedded devices into a web of remotely controlled mobile sensors.

11. CONCLUSION

The Symbiote mechanism can be used to augment legacy embedded devices with novel functionality in an OS agnostic manner. The applications of this capability are numerous, and will help make the introduction of modern host-based defenses on existing embedded devices a feasible reality. The checksumming Symbiote payload described in this paper is a starting point in demonstrating the unique advantages of Symbiotic defense systems. We have demonstrated that the

Symbiote can automatically augment Cisco IOS with effective anti-rootkitting capabilities. This accomplishment has laid the foundation for the construction of a large sensor-grid of legacy embedded devices in order to accurately detect and analyze the exploitation of the devices which make up the fabrics of our global communication infrastructures.

12. ACKNOWLEDGEMENTS

This material is based on research sponsored by Air Force Research labs under agreement number FA8750-09-1-0075. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This material is also based on research sponsored by DARPA contract: CRASH program, SPARCHS, FA8750-10-2-0253.

13. REFERENCES

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

[2] Microsoft Corporation, Kernel Patch Protection: Frequently Asked Questions. http://tinyurl.com/y7pss5y, 2006.

[3] The End of Your Internet: Malware for Home Routers, 2008. http://tinyurl.com/3d9yv9l.

[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] Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In Tomas Sander, editor, *Digital Rights Management Workshop*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2001.

[7] Ang Cui, Jatin Kataria, and Salvatore J. Stolfo. Killing the myth of cisco ios diversity: Towards reliable, large-scale exploitation of cisco ios, 2011. 5th USENIX Workshop on Offensive Technologies.

[8] 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.

[9] Ang Cui and Savaltore J. Stolfo. Defending legacy embedded devices with software symbiotes. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *RAID*, volume 6961 of *Lecture Notes in Computer Science*. Springer, 2011.

[10] Abdallah Ghourabi, Tarek Abbes, and Adel Bouhoula. Honeypot router for routing protocols protection. In Anas Abou El Kalam, Yves Deswarte, and Mahmoud Mostafa, editors, *CRiSIS*, pages 127–130. IEEE, 2009.

[11] Christopher Krügel, William K. Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC*, pages 91–100. IEEE Computer Society, 2004.

[12] Felix ”FX” Linder. Cisco Vulnerabilities. In *In BlackHat USA*, 2003.

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

[14] Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors. *Recent Advances in Intrusion Detection, 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*, volume 5230 of *Lecture Notes in Computer Science*. Springer, 2008.

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

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

[17] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Lippmann et al. [14], pages 1–20.

[18] Dror-John Roecher and Michael Thumann. NAC Attack. In *In BlackHat USA*, 2007.

[19] Skywing. Subverting PatchGuard Version 2, 2008. Uninformed,Volume 6.

[20] Yingbo Song, Pratap V. Prahbu, and Salvatore J. Stolfo. Smashing the stack with hydra: The many heads of advanced shellcode polymorphism. In *Defcon 17*, 2009.

[21] Salvatore J. Stolfo, Issac Greenbaum, and Simha Sethumadhavan. Self-monitoring monitors. Technical Report cucs-026-09, Columbia University Computer Science Department, April 2009.

[22] Vikas R. Vasisht and Hsien-Hsin S. Lee. Shark: Architectural support for autonomic protection against stealth by rootkit exploits. In *MICRO*, pages 106–116. IEEE Computer Society, 2008.

[23] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In Lippmann et al. [14], pages 21–38.