Defending Embedded Systems with Software Symbiotes

Ang Cui and Salvatore J. Stolfo

Department of Computer Science
Columbia University
New York NY, 10027, USA
{ang,sal}@cs.columbia.edu

Abstract. A large number of embedded devices on the internet, such as routers and VOIP phones, are typically ripe for exploitation. Little to no defensive technology, such as AV scanners or IDS's, are available to protect these devices. We propose a host-based defense mechanism, which we call Symbiotic Embedded Machines (SEM), that is specifically designed to inject intrusion detection functionality into the firmware of the device. A SEM or simply the Symbiote, may be injected into deployed legacy embedded systems with no disruption to the operation of the device. A Symbiote is a code structure embedded in situ into the firmware of an embedded system. The Symbiote can tightly co-exist with arbitrary host executables in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. The Symbiote is stealthily embedded in a randomized fashion within an arbitrary body of firmware to protect itself from removal. We demonstrate the operation of a generic whitelist-based rootkit detector Symbiote injected in situ into Cisco IOS with negligible performance penalty and without impacting the routers functionality. We present the performance overhead of a Symbiote on physical Cisco router hardware. A MIPS implementation of the Symbiote was ported to ARM and injected into a Linux 2.4 kernel, allowing the Symbiote to operate within Android and other mobile computing devices. The use of Symbiotes represents a practical and effective protection mechanism for a wide range of devices, especially widely deployed, unprotected, legacy embedded devices.

Key Words: Symbiotic Embedded Machines, Embedded Device Defense, Cisco IOS Rootkit Detection

1 Introduction

A recent study demonstrates that there are a vast number of unsecured embedded systems on the internet, primarily routers, that are trivially vulnerable to exploitation with little to no effort. Several new exploits against Cisco IOS demonstrate the vulnerability of a vast number of high end legacy routers to easy exploitation. We propose a novel technique to detect and defend against advanced malware threats against the internet routing infrastructure, as well as a vast number of other types of embedded systems.

We present a host-based defense mechanism which we call "Symbiotic Embedded Machines" (SEM). SEM, or simply the Symbiote, is an experimental system that injects intrusion detection functionality within the firmware of a (legacy) embedded system and that senses the unauthorized modification of the device firmware. Symbiote injection may be randomized so that each instance is distinct from all other injected systems in order to thwart attempts by an adversary to disable the injected Symbiote. In general, we aim to create a symbiotic software construct which provides the following four fundamental security properties once it is active within the firmware of an embedded system or a host program:

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 firmware or host program. In order for the host to function as before, its injected SEMs must execute, and vice versa.
3. The Symbiote's code cannot be modied or disabled by unauthorized parties through either online or offline attacks.
4. No two instantiations of the same Symbiote is 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.

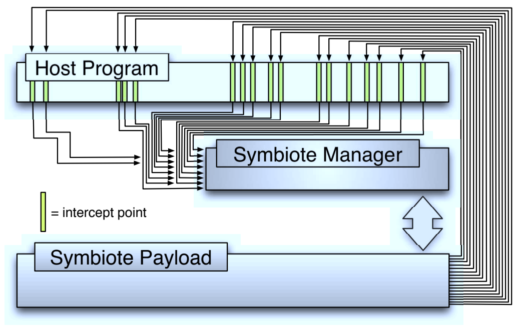

Fig. 1. Logical overview of SEM injected into embedded device firmware. SEM maintains control of CPU by using large scale randomized control-flow interception. SEM payload executes alongside original OS. Figure 6 shows a concrete example of how the SEM payload can be injected into gaps within IOS firmware.

We aim to demonstrate the highest levels of protection we believe we can achieve with this technology in a range of embedded system device types. An immediate application of the system presented in this paper is the fortification of existing vulnerable network routing devices. As Section 3 illustrates, the embedded security threat is particularly difficult to solve, especially if the goal is to improve the security of the existing software infrastructure. Network embedded devices like routers and firewalls are vulnerable to the same attacks as general purpose computers, but generally do not have the facility to execute third-party host-based defenses like anti-virus. Using the Symbiote, we have successfully injected a host-based root-kit detection mechanism into a closed-source proprietary operating system, Cisco IOS. We believe that the techniques discussed in this paper can be used to fortify existing vulnerable devices within the critical infrastructure, like smart power meters, machine to machine control systems, as well as everyday embedded devices like VoIP phones, home routers and mobile computers.

Figure 1 shows how a Symbiote is typically injected into a host program. A large number of control-flow intercepts are distributed randomly throughout the body of the host program, allowing the Symbiote Manager to periodically regain control of the CPU. Once the Symbiote Manager is invoked, it then executes a small portion of the its defensive payload before saving its execution context and returning control back to the host program. This allows the Symbiote and host program to execute in tandem, in a time-multiplexed manner without affecting the functionality of the original host program. The Symbiote injection process provides a probabilistic lower bound on the frequency in which the Symbiote will be invoked at runtime as a adjustable parameter. The Symbiote, which resides within the same execution environment as the host program has the ability to passively monitor or proactively alter the host program's behavior at runtime. Since the Symbiote is deeply intwined with its protected host program, attempts to corrupt or alter the Symbiote binary will either be detected by the Symbiote or cause the host program to crash. (See Section 5)

As we see in Section 4, Symbiotes can defend any arbitrary executable, even other Symbiotes. Unlike traditional anti-virus and host-based defense mechanisms which install into and depend heavily on facilities provided by the vulnerable systems they are meant to protect, the Symbiote treats its host program as an external and untrusted entity. Symbiotes do not depend on functionality provided by its host, giving it several critical advantages.

The Symbiote:

Is agnostic to its operating environment. Since the Symbiote injects itself **into** its host program, it does not need to conform to any executable format. The Symbiote will execute as long as its host program is a valid executable, regardless of operating system type or version.

Can reside within any arbitrary executable, regardless of its functionality or position within the system stack. The unique injection mechanism allows the same Symbiote to operate within userland applications, device drivers, the kernel or even other Symbiotes. Furthermore, many instances of the same Symbiote can simultaneously operate on **multiple** levels of the system stack, enabling a new approach to systematically deploying defenses in depth.

Can be easily and safely be injected into proprietary black box operating systems. Since Symbiotes are agnostic to the inner workings of its host program and execution environment, deploying Symbiotes on proprietary systems is as easy as deploying them within well known ones.

Is self-sufficient, and does not depend on facilities provided by its host program. The Symbiote threats its host program as an untrusted and foreign entity. It does not leverage any external code to protect its host, and is therefore not vulnerable to attacks on other parts of the system.

Is self-protecting and stealthy, and thus is difficult to detect and deactivate by an adversary.

Is efficiently executed, utilizing the raw computational resource of the hardware platform, bypassing layers of overhead produced by OSs, or VMs that host an OS. One would prefer to use a SEM implementation of a security payload, rather than a reference monitor, for example, because of this performance advantage.

We demonstrate the advantages of Symbiotes by tackling a difficult, yet ubiquitous problem for which no effective host-based defenses currently exist. Our current implementation of a Symbiote, that we call Doppelgänger, is easily and safely injected into proprietary operating systems to protect resource-constrained embedded devices from a wide array of memory manipulation attacks. The unique properties of the Symbiote allows us to systematically fortify many different Cisco routers with the same root-kit defense payloads in an automated fashion. The Symbiotic approach is not specific to any particular device or operating system, and can be used to effectively mitigate the embedded device security problem.

This paper is organized as follows. Section 2 discusses existing defenses against code modification attacks, with an emphasis on the current state of host-based embedded system defense. Section 3 discusses the vulnerability of embedded devices, defines the threat model and surveys related work. Section 4 describes the Symbiotic Embedded Machine architecture as well as the white-list based rootkit detection payload in detail. Section 5 discusses an lower bound on the computational complexity of a successful attack against software Symbiotes in an online attack, as well as common attacks which can be levied against SEM. Section 6 shows experimental results and discusses the theoretical and experimental performance overhead of Doppelgänger, our implementation of SEM, for IOS versions 12.2 and 12.3 on a Cisco 7121 router. We conclude in section 7 suggesting that proactive protection of network embedded devices using SEMs with exploitation detection payloads is a viable strategy to mitigate large-scale compromise of our global communication networks and critical infrastructures. Appendix A contains performance evaluation data of the rootkit detection SEM payload running on IOS 12.3 on a physical Cisco 7121 router under load.

2 Related Work

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, Norton, Kapersky and Microsoft [1] 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 [16–18]. Sophisticated detection and prevention strategies have been proposed by the research community. Virtualization-based strategies using hypervisors, VMM's and memory shadowing [15] have been applied to kernel-level rootkit detection. Others have proposed detection strategies using binary analysis [9], function hook monitoring [22] and hardware-assisted solutions to kernel integrity validation [19].

Guards, originally proposed by Chang and Atallah [3], is a promising technology which uses mechanisms of action similar to Symbiotes. Originally proposed as an anti-tampering mechanism for x86 software, the guard mechanism have been used in both security research [5] as well as commercial products[1]. 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. To further improve the resilience of the protection scheme, a large number of Guards can be deployed in intricate networks as a graph of mutually defensive security units.

While promising, the Guard approach does have several draw backs and limitations which Symbiotes overcome. For example, since the Guard has no mechanism to pause and resume its computation, the entire guard routine must complete execution each time it is invoked. This limits the amount of computation each Guard can realistically perform without affecting functionality, specially when Guards are used in time sensitive software and real-time embedded devices. In contrast, the Symbiote Manager (See 4) allows its payload to be arbitrarily complex. Instead of executing the entire payload each time a randomly intercepted function invokes the Symbiote, the Symbiote Manager executes a small portion of the payload before pausing it, saving its execution context and returning control back to the intercepted function. This way, Symbiote payloads can implement arbitrarily complex defensive mechanisms, even in time sensitive software.

Lastly, the techniques used by Symbiotes, such as function interception, randomized payload injection, have been undoubtably used by malware authors in the past. Indeed, a Symbiote-like rootkit [4] has recently been disclosed for Cisco IOS. The Symbiote structure incorporates such traditionally "offensive" techniques for defensive purposes in order to hide and harden itself against attacks which aim to disrupt the Symbiote.

[1] www.arxan.com

3 Threat Model

We assume the attacker is technically sophisticated and has access to both zero-day vulnerabilities as well as compatible exploits allowing reliable execution of arbitrary code. We further assume that the attacker executes the attacks in an online fashion. In other words, the attacker must carry out the attack remotely against a running device without interfering with its function or causing it to crash or reboot. Attacks involving configuration changes or replacement of the entire firmware image (which requires a reboot) are excluded from our model because they can be detected by conventional methods. We also assume that the attacker has access to the original host program image, before any Symbiotes are injected into it.

Online attacks against the protected host program can be separated into two categories; those that attempts to disable or evade the Symbiotes protecting the host program, and attacks that do not. We first address existing attacks which target the host program and show how Symbiotes can prevent such attacks. Section 5 discusses multi-stage attacks which attempts to disable Symbiotes prior to executing their malicious payloads.

With respect to Cisco routers, we focus on rootkit techniques which make persistent changes to the IOS operating system. The SEM mechanism introduced in this paper is used to detect injected code that changes portions of the device that are otherwise **static** during the life time of the device. The Symbiote payload presented in this paper is designed only to detect unauthorized code modification. However, the SEM approach can also be used to detect exploitation in dynamic areas of the target embedded device like the stack and heap. Symbiote control-flow interception methods and payloads which defend against return-to-libc, return oriented and heap related attacks are currently under research.

The Symbiote implementation presented in this paper focusses on fortifying legacy network embedded devices. The next section discusses the embedded security problem and shows how Symbiotes can be used to defend network routers against code modification attacks.

3.1 Solving the Embedded Problem with Symbiotes

Network embedded devices are ubiquitous within the modern home, office and global communication infrastructures. Enterprise networking equipment are specialized embedded devices which power the world's communication backbones. Consumer network devices like wireless access points, web cams, networked printers and smart phones litter our homes, streets, offices and pockets and provide functionality on which we have come to depend. While network embedded devices like Cisco routers and firewalls constitute a large portion of our commercial, residential, enterprise and military communication infrastructures, little research has been devoted to understanding and mitigating the vulnerabilities of these black box devices. Similarly, since network embedded devices often are closed systems which use proprietary hardware and software, security mechanisms like anti-virus and host-based anomaly detectors found on general purpose computers do not exist for embedded devices. Consequently, there exists a large population of unprotected vulnerable embedded devices in the world. A recent study estimates that a hypothetical zero-day smart meter worm could propagate to 15,000 nodes in approximately 24 hours [12]. Large scale exploitation of routers have already been observed in the wild [2]. Furthermore, the detection of compromised embedded devices poses significant challenges due to the proprietary and limited nature of such devices. Therefore, a proactive, preventative defense strategy is not only the most desirable approach, but is also likely the only practical one.

The proof of concept defensive Symbiote payload we inject detects attempts and prevents all rootkits from working. Engineering such a generic defensive mechanism into black box devices is not easy. The challenge is at least twofold. First, embedded devices often use undocumented proprietary operating systems. These devices almost never provide an interface for installing new software on top of the existing firmware. Second, embedded device hardware and software is very diverse. If one were able to develop a working defense for a popular device, that defense will most likely not work across even minor software revisions for the same device, and will certainly not work for different devices from different hardware vendors

We demonstrate how Symbiotes overcome both obstacles by targeting two versions of Cisco IOS running on MIPS. The Cisco router IOS rootkit detection Symbiote, we call Doppelgänger, requires no modification of IOS, and is automatically loaded into firmware images of two major versions, 12.2 and 12.3. The SEM injection process requires a handful of parameters specific to the target firmware, including a list of randomly chosen control-flow intercept points and locations of usable memory. All such parameters are computed automatically by a simple single pass analysis of the target binary. Doppelgänger utilizes well-known code injection methods in a novel way by randomly diverting a very large set of control-flow intercept points. Doppelgänger uses these hooks to support the execution of arbitrary payloads which are both invisible to the original OS and highly resilient against unauthorized deactivation and removal. The Symbiote's control-flow intercepts are randomly distributed through out regions of the host program which are executed with high probability under normal operating conditions. This "live" code detection approach allows us to provide a probabilistic lower bound on the frequency in which the Symbiote will regain control of the CPU while the host program is in execution. (See Section 5).

We inject payloads with functionality that permits code to operate **alongside** the original device OS; not within it as a process, nor under it as a hypervisor would do. Such payloads allow us to monitor and control the original device's OS internals without being restricted by it. The accomplishment of this symbiotic feat also provides stealth as a by-product.

Several real-world considerations make the use of SEM for security purposes effective and practical. First, SEM is a deployment vehicle which largely abstracts away hardware and software diversity. This allows sophisticated security mechanisms to be written once and deployed across many different embedded devices. Second, the application of white-list based protection mechanisms is ideal for embedded devices which tend to have monolithic firmwares. Mechanisms, like code integrity verification, can be implemented efficiently and can detect any change to the code of the device (i.e. function hooking). For example, the rootkit detection payload presented in this paper is only **336 bytes** (See Section 6). Furthermore, while many “end of life” embedded devices are still in use today, vendors have little incentive to invest resources in maintaining and updating firmware for such devices. Thus, using SEM to retrofit these legacy devices with up-to-date end point defense mechanisms is an attractive and viable alternative.

4 Symbiotic Embedded Machines

The Symbiote is a self-contained entity and is not installed onto the host program in the traditional sense. It is injected into its host program’s code in a randomized fashion. Current legacy anti-virus and host-based defenses must be installed onto or into a legacy operating system, which places a heavy dependence on the features and integrity of the operating system. In general, this arrangement requires a strong trust relationship with the very software (often of unknown integrity) it tries to protect. In contrast, the Symbiote treats its entire host program as an external and untrusted entity, and therefore eliminates the unsound trust on traditional legacy systems.

4.1 The Symbiote-Host Relationship

The Defensive Mutualistic relationship between the Symbiote and host program can be broadly described as follows:

1. Both the Symbiote and the protected software host are functionally autonomous. Specifically, the Symbiote is not a standard piece of software that depends on and operate within the software system it is protecting. Instead, the Symbiote can be thought of as a fortied and self-contained execution environment that is infused into the host software.
2. The Symbiote resides within the host software, extracting computational resources (CPU cycles) to execute its own SEM payloads. In return, the SEM payloads will constantly monitor the execution and integrity of the host software, fortifying the entire system against exploitation. The Symbiote payload may execute repair operations on the host, or carry out any arbitrarily defined policy enforcement.
3. SEMs are injected into the host software rather then installed in the traditional sense. Once injected, the code of the SEM is pseudorandomly dispersed across the body of the host. Special mechanisms provided by the SEM injection process will assure that the SEM is executed along-side its host software.

4. The Symbiote and host program must operate correctly in tandem. The Symbiote monitors the behavior of the protected host program, and can alert on and react to exploitation and incorrect behavior. The Symbiote is also self-fortied with anti-tampering mechanisms. If an unauthorized party attempts to disable, interfere with or modify the Symbiote, the protected host program will become inoperable if the attempt is successful.
5. A Symbiote may be injected recursively into another Symbiote to provide the same protection to a Symbiote in cases requiring extreme fault tolerance and security.
6. No two instantiations of the same Symbiote are ever the same. Each time a Symbiote is created and prepared for injection into a host program, its code is randomized and mutated using polymorphic engine technology, resulting in a dissimilar variant of itself. When observed at the macro level, the collective Symbiote population is highly diverse.

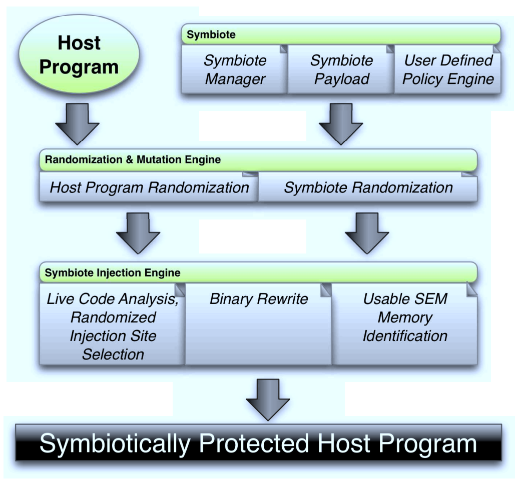
Host Program
Symbiote
Symbiote Manager
Symbiote Payload
User Defined Policy Engine
Randomization & Mutation Engine
Host Program Randomization
Symbiote Randomization
Symbiote Injection Engine
Live Code Analysis, Randomized Injection Site Selection
Binary Rewrite
Usable SEM Memory Identification
Symbiotically Protected Host Program

Fig. 2. Generic end-to-end process of fortifying an arbitrary host program with a Symbiote. Our proof of concept Symbiote, Doppelgänger, is completely implemented in software and can execute on existing commodity systems without any need for specialized hardware.

Each instantiation of a Symbiote is polymorphically mutated and randomized during the injection process. Therefore, studying and reverse engineering one instance of a particular Symbiote provides the attacker with little to no useful information about the specifics of any other instantiation of the same Symbiote. The Symbiotic Embedded Machine structure creates an **independent** execution context from the native operating system at runtime. SEM uses the newly created context to execute arbitrary payloads. These payloads can be written in any high level language (typically C). We may view SEM as a structure which moves the entire IOS environment into one logical context and creates another for the SEM payload. Once done, the SEM acts as an improvised Virtual Machine Manager and executes both logical contexts in a time multiplexed manner.

It is important to note that SEM does not use traditional virtualization techniques. Due to the fact that most network embedded devices do not have hardware hypervisor or virtualization support, the methods we use to achieve execution context separation use only standard CPU instructions. Techniques such as control-flow interception and inline hooking have also been used in software debuggers and reverse engineering frameworks. In this sense, SEM can be thought of as a sophisticated dynamic debugger rather than a virtualization mechanism.

4.2 Doppelgänger: A Symbiote Protecting Cisco IOS

Figure 1 shows the three logical components of Symbiotic Embedded Machines: Control-Flow Interceptors, Symbiotic Embedded Machine Manager (SEMM) and the SEM Payload. Together, all three components are injected *in situ* into the target embedded device firmware. The injection process can be carried out offline (*i.e.* creates new fortified firmware) or dynamically (*i.e.* during exploitation, as a part of a multi-stage shellcode). In practice, the injection process can be accomplished with minimal invasiveness. Since SEM is injected *in situ*, the size of the resulting firmware image is unchanged. For example, our current implementation of Doppelgänger, along with the rootkit detection payload requires only 1384 bytes to be injected into IOS. Figure 5 illustrates typical "gaps" within IOS firmware which can safely be used to embed the SEM payload.

For generality, SEM does not rely on firmware specific code features like system calls or variants of libc. The Control-Flow Interceptor component uses inline hooks to intercept a large number of functions within the target firmware. Upon invocation of an intercepted function, control of the CPU is redirected to the Symbiotic Embedded Machine Manager (SEMM), which executes a small portion of the SEM payload. For concreteness, the SEMM manages the execution of injected SEM payload as follows:

1. Store the execution context of the native OS (i.e. IOS).
2. Load the context of the SEM payload.
3. Compute how long the SEM payload can run, based on current native OS system utilization.
4. Execute the SEM payload for that amount of time.
5. Store the execution context of the suspended SEM payload.
6. Load the execution context of the native OS at the time the SEMM assumed control.
7. Restore CPU control to the invoked function.

4.3 Live Code Interception with Inline Hooks

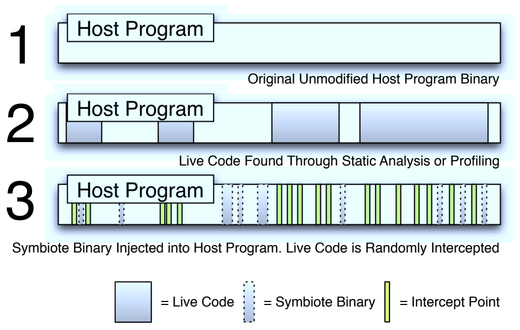
1
Host Program
Original Unmodified Host Program Binary
2
Host Program
Live Code Found Through Static Analysis or Profiling
3
Host Program
Symbiote Binary Injected into Host Program. Live Code is Randomly Intercepted
= Live Code
= Symbiote Binary
= Intercept Point

Fig. 3. Symbiote Injection Process.

Figure 3 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, random intercept points are chosen out of the live code regions found. Lastly, each 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.

Control-flow intercepts are distributed in a randomized fashion through out the host program's binaries in order to ensure that the Symbiote regains control of the CPU periodically. We would like to ensure that these randomly chosen intercept points are located within regions of code which will be frequently executed at runtime. This problem is difficult to solve with high accuracy in the general case. However, our purposes do not require the classification mechanism to be absolutely accurate. In reality, implementing a sufficient solution for real-world host programs is not too difficult. Section 4.4 discusses the methods used in our experiments for live code classification.

Once regions of code within the host program are chosen for control-flow interception, the Symbiote injection process imbeds interceptors as well as the Symbiote binary into the host program. The Symbiote implementation presented in this paper uses a Detour [21] style inline function hooking mechanism for control-flow interception. Note that while we injected our intercepts within the function preamble in the current Symbiote implementation, this is not a requirement. Control-flow intercepts can be embedded in arbitrary positions within the host program using existing binary instrumentation techniques.

Detour [21] style inline hooking is a well known technique for function interception. However, SEM uses function interception in a very different way. Instead of targeting specific functions for interception which requires precise *a* *priori* knowledge of the code layout of the target device, SEM randomly intercepts a large number of functions as a means to re-divert periodically and consistently a small portion of the device's CPU cycles to execute the SEM payload. This approach allows SEM to remain agnostic to operating system specifics while executing its payload **alongside** the original OS. The SEM payload has full access to the internals of the original OS but is not constrained by it. This allows the SEM payload to carry out powerful functionality which are not possible under the original OS. For example, the IOS rootkit detection payload presented in Section 4.5 bypasses the process watchdog timer constraint, which terminates any IOS process running for more than several seconds, because the detector operates outside the control of the OS.

Stealth is a powerful 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.

4.4 Automatically Locating Control-Flow Intercept Points

Control-flow intercept points are chosen randomly out of candidate *live* code regions within the host program. The way code regions are classified as *live*, as well as the number of intercepts chosen from each region directly affects the frequency in which the Symbiote will gain control of the CPU, which in turn directly affects the performance and overhead of the Symbiote.

Both dynamic and static methods of live code classification are considered for our experiments. First, the host program is executed under a profiler in order to observe live code, or code coverage, under normal operating conditions[2]. Using code coverage analysis to classify live code is advantageous because it can not produce false positives, i.e. dead code can not be classified as live code. However, this dynamic approach can not classify regions of code which are reachable only through rare or malformed program input. Therefore, we augment our code coverage based live code classifier with static analysis of the control-flow graph of the host program. Figure 4 shows the live code regions of a typical IOS router firmware image after our initial analysis. Control-flow intercept points will be chosen randomly out of these code regions (shown in white) to periodically divert CPU control to the injected Symbiote. Note that intercept points can, and should also be placed in the binary outside of the detected live code regions.

4.5 Rootkit Detection Payload

To detect IOS malcode and rootkits described in the previous section, we implement a white-list strategy. Known rootkits operate by hooking into and altering

[2] In the case of IOS, we profiled the router image using Dynamips under various workloads.

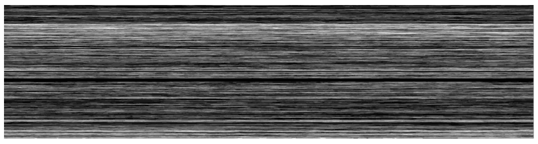

Fig. 4. Live Code Regions (White) Within IOS 12.4 Firmware (Black). Code Range: 0x80008000-0x82a20000

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. For the white-list strategy to be effective, the protected kernel code must either remain static during legitimate operation, or only be allowed to change in **predictable** ways. For example, while some embedded operating systems support legitimate mechanisms to dynamically update the kernel, the contents of those updates are and static and known *a priori*. Therefore, the checksums of approved updates can be calculated and distributed to SEM a head of time.

Formally, let

$$H_c = F_{hash}(S_c)$$

where B is a binary firmware (eg. IOS), and $\{S_c\}$ is a set of contiguous code segments within B we wish to monitor. If H_c outputs a cryptographically secure hash function over all monitored code segments, a change in H_c, then, indicates a change within at least one code segment in $\{S_c\}$.

$$H_c = \{x | x \in S_c, F_{hash}(x)\}$$

Furthermore, we can compute and monitor multiple hash values $\{H_{c_i}\}$ over any arbitrary subset of $\{S_c\}$. By doing so, it will give arbitrary resolution on the location of code modification at cost of increased memory and computational overhead.

5 Computational Lower Bound of Successful Software-Only Symbiote Bypass

This section discusses multi-stage attack strategies which attempt to disable the Symbiote prior to executing their malicious payload. We provide an intuitive lower bound of the computational cost of a successful attack against software-only Symbiotes. We also discuss ways of detecting and defending against such multi-stage attacks.

Naturally, the software-only Symbiote is not invulnerable to attack, and can not guarantee absolute protection when deployed as the *only* security mechanism. Instead, software-only Symbiotes should be deployed in tandem with traditional network and host-based mechanisms in a defense in depth arrangement.

Generally, software-only Symbiotes can be successfully bypassed in two ways:

Attack 1: Remove control-flow intercepts. If the attacker can remove all control-flow intercepts within all live code regions before the Symbiote's detection latency, the attacker can prevent the Symbiote from ever regaining control of the CPU.

Attack 2: Deactivate the SEMM or Payload. If the attacker can locate and patch the Symbiote's manager or payload code, the Symbiote can be completely disabled.

Before we analyze the two attacks mentioned above, consider the set of binaries that constitutes a typical host program. Regions of binaries within the host program can be classified as live code, reachable code or dead code. Clearly, dead code is not reachable via any possible execution path. Conversely, reachable code can be executed under some set of inputs. More importantly, *live code*, a subset of reachable code, is *frequently* executed under typical inputs of the host program. In other words, live code represents the regions of the host program active under the *normal behavior model* of the specific host program in a specific environment.

The Symbiote control-flow intercepts are randomly distributed within the live code regions, while the Symbiote Manager and Payloads are distributed randomly through out the entire host program[3].

Both attacks reduce to a common general problem of identifying all P out of N bytes, P being the bytes belonging to the Symbiote component under attack, N being the bytes of the host program in which P can exist. In the case of attack 1, the attacker must identify and remove all control-flow intercepts, P injected into all live code regions, N (assuming that this is known). Since the Symbiote binary is polymorphically mutated at injection time, the attacker can not search for a well-known Symbiote signature through the binary. Instead, the attacker must compare an unmodified copy of the host program with the victim host program during an online attack. This is essentially equivalent to at least a linear operation over the size of all live code regions.

Similarly, since the Symbiote binary is distributed randomly throughout the host program, an attacker must identify all code regions belonging to the Symbiote. There are many ways to do this. However, since no well-known signature exists for the Symbiote code, the attacker must perform dynamic disassembly in order to follow control-flow intercepts to a piece of Symbiote code. Alternatively, the attacker can perform a linear comparison of the entire host program to identify all injected Symbiote code. In the former case, the attacker's problem is reduced to attack 1, because unless all control-flow intercepts are removed, the attacker can not be sure that all Symbiotes are removed. In the latter case, the attacker must use a linear amount of CPU and network I/O, which again reduces to the problem of identifying P bytes out of N.

[3] While the Symbiote is distributed randomly through out the binary of the host program, the injection process ensures that the Symbiote code can not be inadvertently executed by the host program. In other words, the control-flow intercepts are the only mechanism in which the Symbiote code will be invoked.

To put these attacks into perspective, the average size of the host programs analyzed in our experiments is approximately 35 MB, the size of live code regions considered for control-flow interception is approximately 10 MB. Each host program contains approximately 75,000 functions, all of which can be intercepted. (Note that control-flow interception need not take place only at the function preamble, but can exist anywhere within the function body.) If the attacker attempts to perform a linear comparison, at least portions of the unmodified host program will have to be transferred over the network during the online attack. The attacker can also attempt to dynamically disassemble the 10 MB of live code. Both attack strategies require a very large amount of network I/O or CPU which raises the bar quite high for the attacker to overcome without being noticed.

6 Symbiote Performance and Computational Overhead

We randomly choose a set of control-flow intercept points within *live* regions of the target host program. The method and parameters used to determine *live* regions, as well as the number of intercept points chosen gives us fine grain control of $p(\alpha_i, \delta, \tau_q)$, and gives us a probabilistic bound on the frequency in which the Symbiote will gain control of the CPU. Section 4.4 discusses the methods we used to extract "live" regions from the host program.

Consider the computational cost of an injected SEM during some time period τ_q.

Let $\{\alpha_1 ... \alpha_n\}$ be the set of all functions in binary firmware β.

Let $g(\alpha_i, \tau_q)$ be the cost of SEM per invocation at time period τ_q.

Let $h(\alpha_i)$ be the binary function representing whether function α_i is "intercepted" by the SEM.

Let $p(\alpha_i, \delta, \tau_q)$ be the number of times function α_i will be invoked during time period τ_q, given some probability distribution δ.

Note that the probability distribution δ is derived from the "live" code analysis performed during the Symbiote injection process. Suppose a control-flow intercept is inserted into a piece of "live" code which is known to execute with some probability, according to the normal execution model of the host program. We can claim that the Symbiote control-flow intercept will also be invoked with at least this probability. Thus, the "live" code analysis gives us a probabilistic lower bound on the frequency in which the Symbiote will regain control of the CPU over any time period τ_q.

Let the SEM cost function $g(\alpha_i, \tau_q)$ be:

$$g(\alpha_i, \tau_q) = O_{SEMM} + O_{payload}(\alpha_i, \tau_q) \tag{1}$$

Where O_{SEMM} is the (**constant**) cost of invoking the SEMM and $O_{payload}(\alpha_i, \tau_q)$ is the amount of the SEM payload to execute (**variable**), given function α_i and time period τ_q.

The **Lower bound on SEM cost C_q, over time period** τ_q can be expressed as:

$$C_q = \Sigma_i O_{SEMM} * p(\alpha_i, \delta, \tau_q) \tag{2}$$
$$= O_{SEMM} \Sigma_i p(\alpha_i, \delta, \tau_q) \tag{3}$$

Intuitively, the lower bound on the SEM cost is simply the overhead of invoking the SEMM multiplied by the expected number of times that the SEMM will be invoked over time period τ_q.

The computational cost of SEM C_q, over time period τ_q is:

$$C_q = \Sigma_i g(\alpha_i, \tau_q) * h(\alpha_i) * p(\alpha_i, \delta, \tau_q) \tag{4}$$

The **Upper bound on SEM cost C_q over time period τ_q.** is a function of the number and distribution of functions intercepted in order to execute the SEMM and the cost of the payload execution the SEMM manages. Let $h(\alpha_i) = 1$ for all functions α), then

$$C_q = \Sigma_i g(\alpha_i, \tau_q) * p(\alpha_i, \delta, \tau_q) \tag{5}$$
$$= \Sigma_i (O_{SEMm} + O_{payload}(\alpha_i, \tau_q)) * p(\alpha_i, \delta, \tau_q) \tag{6}$$
$$= O_{SEMm} \Sigma_i p(\alpha_i, \delta, \tau_q) + \Sigma_i O_{payload}(\alpha_i, \tau_q) * p(\alpha_i, \delta, \tau_q) \tag{7}$$

Observations

- The distribution δ, and therefore, $p(\alpha_i, \delta, \tau_q)$ can not be changed (without changing the host's original functionality), and varies with respect to different devices and firmware.
- The function $h(\alpha_i)$ can be used to control SEM CPU utilization but is binary and **imprecise**.
- The function $g(\alpha_i, \tau_q)$ can be used to control SEM CPU utilization[4] **precisely**.

We can vary the **number** of control-flow interceptions ($h(\alpha_i)$) and the **amount** of SEM payload that is executed at each invocation ($g(\alpha_i, \tau_q)$) to control precisely the amount of CPU time used by the SEM. We can implement these two mechanisms in the **SEMM** to divert more CPU cycles to the SEM during periods of low CPU utilization and divert less during periods of high CPU utilization. Figure 6 shows actual CPU utilization when Doppelgänger and our rootkit detection payload are installed on a physical Cisco 7120 router with $g(\alpha_i, \tau_q)$ set to several fixed values. This parameter directly affects the portion of the CPU that is diverted to executing the SEM payload. Figure 7 and Table 1 shows an

[4] In practice, O_{SEMm} is much smaller than $O_{payload}()$, therefore, the second summation in equation 7 dominates over the first (Section 6.1).

inverse relationship between $g(\alpha_i, \tau_q)$ and the amount of time required to detect a modification of IOS, which we call the **detection latency**.

Clearly, the more CPU resources the Symbiote Manager diverts away from the host program, the shorter the detection latency will be. However, this can also impact the performance of the host program. Therefore, the Symbiote Manager must perform the important task of regulating, or *scheduling*, the Symbiote payload for execution in a way which optimizes both detection latency and overall host program performance. This can be reduced to scheduling algorithms which control the frequency of Symbiote payload invocation $h(\alpha_i)$, as well as the duration of the payload's execution at each invocation $g(\alpha_i, \tau_q)$.

Such scheduling algorithms are critical in regulating the resource consumption of the Symbiote payload, and must be adaptive to the current resource utilization of the host program. For example, an *inverse adaptive* algorithm can throttle back the Symbiote payload's execution rate when the host program is highly utilized, thus preventing the Symbiote from disrupting the functionality of the host program when resource utilization is nearing its limits. Similarly, *real-time* and *batch-like* scheduling algorithms can also be implemented in the Symbiote Manager. The development of such adaptive scheduling algorithms within the Symbiote Manager is an area of ongoing research.

6.1 Experimental Results: Doppelgänger, IOS 12.2 and 12.3, Cisco 7121

Methodology Doppelgänger, our proof of concept SEM implementation is injected into IOS 12.2(27c) and IOS 12.3(3i) on the a Cisco 7120 router. The rootkit detection payload is implemented in C, and calculates a single hash covering the .text memory range **0x60008000** to **0x61662000**. As a proof of concept, we implemented CRC-32 as the hashing function used by the rootkit detection payload.

Two sets of experiments are done to demonstrate both performance characteristics and accurate IOS code modification detection. First, to test CPU utilization, the Cisco 7120 router is put through a standard workload script with varying SEM payload execution burst rates. The workload script touches a cross section of standard router attack surface by performing tasks like enable / disabling routing, generating system status dumps, reconfiguring routing parameters and advertised routes, etc. The CPU utilization is measured by SNMP polling.

To demonstrate IOS code modification detection, we simulate the installation of a rootkit by modifying a SEM protected IOS firmware with added function hooks and code. We then boot the Cisco router with the altered image and measure the time required for the SEM payload to detect the modification. We configure the payload detector to **halt** the router once the modification is detected. This is also done with varying SEM payload execution burst rates to demonstrate the relationship between SEM payload execution rate and runtime detection latency. Performance evaluation data are included in the Appendix.

SEM Payload Burst Rate			
0xF	0x1F	0xFF	0x7FF
56s	43s	35s	0.3s

Table 1. Average Detection Latency at Different SEM Payload Burst Rates IOS 12.2 (Excluding Boot Time)

Experimental Results Figure 6 demonstrates CPU utilization of the 7120 router when the SEM payload execution burst rate, or $g(\alpha_i, \tau_q)$, is varied. Figure 7 shows the total elapsed time (from boot up to router halt) of detection with various SEM payload execution burst rates. Table 1 is the average detection latency **excluding** router boot time (approximately 11 seconds).

Experimental Findings

- The Cisco router continues to function with Doppelgänger running concurrently, even during periods of near maximum CPU utilization.
- SEM CPU utilization can be controlled by varying the payload execution burst rate within the SEMM.
- Detection Latency is inversely proportional to SEM CPU utilization (and SEM payload execution burst rate).
- IOS code modification detection rate is 100% with 0% false positive.

7 Concluding Remarks

We presented a Symbiotic Embedded Machine (SEM), a new and novel software mechanism that provides a means of embedding defensive software into existing embedded devices. Using a specific SEM implementation we call Doppelgänger, we were able to automatically inject a rootkit detection payload into a Cisco 7120 router running multiple firmware images across two major IOS versions, 12.2 and 12.3. By injecting under 1400 bytes of code into the IOS firmware, Doppelgänger protects the router from all function hooking and interception attempts. Our white-list based rootkit detection payload does not require *a priori* knowledge of IOS internals, or signatures of known rootkits, and can protect the router against any code modification attempts. As the SEM structure operates alongside the native OS of the embedded device and not within it, it can inject generic defensive payloads into the target device regardless of it's original hardware or software. Due to the unique nature of network embedded devices, we posit that retrofitting these widely deployed vulnerable devices with defensive SEM's is the best hope of mitigating a significant emerging threat on our global communication infrastructure. SEM is a generic defensive mechanism suitable for general purpose host protection. Our ongoing research aims to demonstrate the advantages of the Defensive Mutualistic paradigm and Symbiotes over traditional AV solutions.

8 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.

References

1. Microsoft Corporation, Kernel Patch Protection: Frequently Asked Questions. http://tinyurl.com/y7pss5y, 2006.
2. Network Bluepill. Dronebl.org, 2008. http://www.dronebl.org/blog/8.
3. 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.
4. Ang Cui, Jatin Kataria, and Salvatore J. Stolfo. Killing the myth of cisco ios diversity: Towards reliable, large-scale exploitation of cisco ios. USENIX Workshop on Offensive Technologies, August 2011.
5. Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *OSDI*, pages 75–88. USENIX Association, 2006.
6. Ligati et al. Enforcing security policies with run-time program monitors. Princeton University, 2005.
7. Nick Harbour. Win at Reversing: API Tracing and Sandboxing Through Inline Hooking, 2009. In BlackHat USA.
8. Fouad Kiamilev and Ryan Hoover. Defcon 16, 2008. Demonstration of Hardware Trojans.
9. Christopher Krügel, William K. Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC*, pages 91–100. IEEE Computer Society, 2004.
10. Felix "FX" Linder. Cisco IOS Router Exploitation. In *In BlackHat USA*, 2009.
11. 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.
12. Stephen McLaughlin, Dmitry Podkuiko, Adam Delozier, Sergei Miadzvezhanka, , and Patrick McDaniel. Embedded firmware diversity for smart electric meters. In *HotSec 10*, 2010.
13. Michael Lynn. Cisco IOS Shellcode, 2005. In BlackHat USA.
14. Sebastian Muniz. Killing the myth of Cisco IOS rootkits: DIK, 2008. In EU-SecWest.
15. Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Lippmann et al. [11], pages 1–20.
16. Dror-John Roecher and Michael Thumann. NAC Attack. In *In BlackHat USA*, 2007.
17. Skywing. Subverting PatchGuard Version 2, 2008. Uninformed,Volume 6.

18. 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.
19. 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.
20. Martin Rinard Vijay Ganesh, Tim Leek. Taint-based directed whitebox fuzzing. IEEE 31st International Conference on Software Engineering, 2009.
21. Redmond Wa, Galen Hunt, Galen Hunt, Doug Brubacher, and Doug Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.
22. Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In Lippmann et al. [11], pages 21–38.

[Performance Measurements of Root Detection SEM Payload on Physical Cisco 7271 Router Running IOS 12.3]

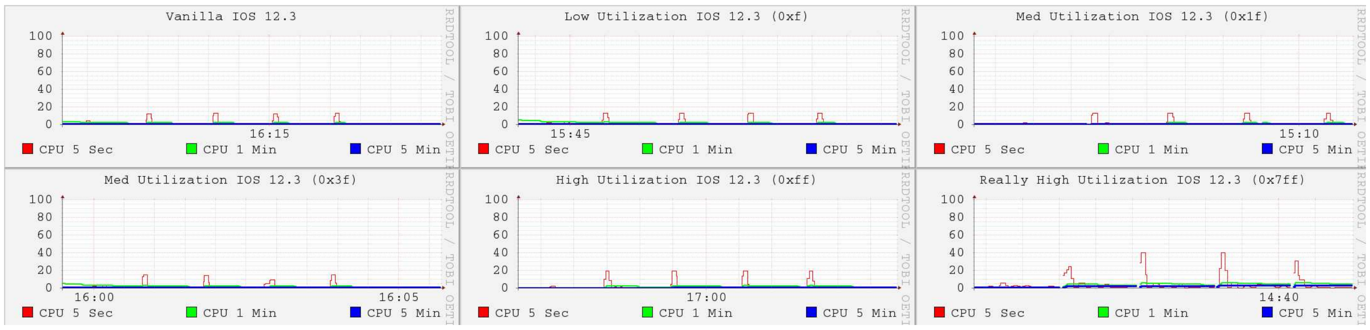

Fig. 5. CPU Utilization on Cisco 7121 Router Using Different SEM Payload Execution Bursts Rates ($g(\alpha_i, \tau_q)$) for IOS 12.3. Note the Direct Relationship Between $g(\alpha_i, \tau_q)$, SEM Payload Execution Time and Total CPU Utilization. Terms Low, Med, High, and Really High Utilization Corresponds to Varying SEM Payload Burst Rates, $g(\alpha_i, \tau_q)$.