Chapter 5
Symbiotes and defensive Mutualism: Moving

Ang Cui and Salvatore J. Stolfo

Abstract If we wish to break the continual cycle of patching and replacing our core monoculture systems to defend against attacker evasion tactics, we must redesign the way systems are deployed so that the attacker can no longer glean the information about one system that allows attacking any other like system. Hence, a new poly-culture architecture that provides complete uniqueness for each distinct device would thwart many remote attacks (except perhaps for insider attacks). We believe a new security paradigm based on perpetual mutation and diversity, driven by symbiotic defensive mutualism can fundamentally change the 'cat and mouse' dynamic which has impeded the development of truly effective security mechanism to date. We propose this new 'clean slate design' principle and conjecture that this defensive strategy can also be applied to legacy systems widely deployed today. Fundamentally, the technique diversifies the defensive system of the protected host system thwarting attacks against defenses commonly executed by modern malware.

5.1 Introduction

We propose a host-based defense mechanism that we call Symbiotic Embedded Machines (SEM). SEM, or simply the Symbiote, is a code structure inspired by a natural phenomenon known as Symbiotic Defensive Mutualism. This phenomenon generally refers to any short- or long-term association between populations of different species where the survival or 'evolutionary fitness' of one or more population partners is enhanced by the association. Mutual benefits are often the result of some emergent behavior between two or more vastly different biological systems. This synergistic dynamic is observed across the spectrum of living things, from microbes

Ang Cui
Columbia University, e-mail: ang@cs.columbia.edu

Salvatore J. Stolfo
Columbia University e-mail: sal@cs.columbia.edu

like viruses and bacteria to fungi and to flora and fauna. When considered within the digital realm, Symbiotic Embedded Machines can be thought of as digital ‘life forms’ which tightly co-exist with arbitrary executables in a mutually defensive arrangement, extracting computational resources (CPU cycles) from it’s host while simultaneously protecting the host from attack and exploitation. Furthermore, the diverse nature of symbiotes provide inherent protection against direct attack by adversaries that directly target host defenses. Hence, defenses are defended by the principle of defensive mutualism.

We envision a general-purpose computing architecture consisting of two mutual defensive systems whereby a self-contained, distinct and unique Symbiote machine is embedded in each instance of a host program. The Symbiote can reside within any arbitrary body of software, regardless of its place within the system stack. The Symbiote can be injected into an arbitrary host in many different ways, while the code of the Symbiote can be ‘randomized’ by advanced polymorphic code engines. Thus, a distinct defensive Symbiote can be used to protect device drivers, the kernel, as well as userland applications. The combination of Symbiote with host program creates a unique executable different from any other instance, and thus breaks the mono-culture by creating a plethora of ‘moved targets’.

Once the Symbiote injection process is complete, it will execute along-side it’s host program. Since the Symbiote is a self-contained entity, it is not installed onto the host program in the traditional sense. Current anti-virus and host-based defenses must be installed onto or into an 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 it’s entire host program as an external and untrusted entity, and therefore eliminates this unsound trust relationship. Much like how certain ants reside within the Bullhorn Acacia tree and acts as a natural defense mechanism against harmful insects, Symbiotic Embedded Machines reside within its host executable, protecting it against exploitation and unauthorized modification. Just as the ants are unfamiliar with the inner workings of the Acacia tree, and as the Acacia tree is unaware of the existence of the ants, SEM’s reside within the target binary in a similar arrangement. At runtime, the host program requires the Symbiote to successfully execute in order to operate. The Symbiote monitors the behavior of its host to ensure it operates correctly, and if not, stops the host from doing harm. Removal, or attempted removal, of the Symbiote renders the host inoperable.

5.2 Related Work

Symbiotic Embedded Machines can be thought of as a generic way of injecting host-based defenses into arbitrary host programs. Traditional host-based defenses are typically installed into well-known operating systems to fortify the entire OS from various types of exploitation. For example, 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 [11, 12, 13]. Sophisticated detection and prevention strategies have been proposed by the research community. Virtualization-based strategies using hypervisors, VMM's and memory shadowing [10] have been applied to kernel-level rootkit detection. Others have proposed detection strategies using binary analysis [5], function hook monitoring [15] and hardware-assisted solutions to kernel integrity validation [14].

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.

The Symbiotic Embedded Machine provide a means of enforcing the integrity of system code and control flow within embedded devices. SEM's platform agnostic code injection methodology can be used to extend the use of run-time program monitors [4] for embedded devices. The vast majority of these devices are built on standard CPU architectures (MIPS, PPC, ARM etc). Therefore, compilation of executable code for these devices using languages like C is trivial. The SEM structure exploits this homogeneity and represents a general method of installing compiled code into firmware of existing network embedded devices, regardless of the underlying operating system, by finding "unused" portions of the firmware that allows stealthy embedded code.

SEM can also be thought of as a novel type of embedded device rootkit. Unlike prior works [9, 6, 8], which are adaptations of existing methods onto embedded operating systems, SEM contains a payload delivery mechanism designed specifically to operate within unfamiliar and heterogeneous proprietary operating systems. SEM can **automatically** inject the same types of rootkit payloads to execute across many different firmware versions and physical device types without requiring deep knowledge of each firmware instance.

5.2.1 Related Work: Software Guards

Guards, originally proposed by Chang and Atallah [2], 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 [3] as well as commercial products[1]. A Guard is a simple

[1] www.arxan.com

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 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.

Removing the limitation on the complexity of Symbiote payloads allows us to further address several draw backs of the Guard framework. Because each guard can only compute for a very short amount of time, they generally performed simple checksums on small patches of software. In order for guards to checksum over the entire protected binary, an intricate network of guards must be injected. Furthermore, guards must be individually instantiated and hooked into the control flow of its protected binary in a specific way in order for the entire guard network to be mutually defensive. This heavy dependence on the execution flow information of the protected program makes the guard injection process complex and error prone. For example, static analysis of the target binary can not always reveal its runtime control flow behavior, specially when computed control-flow transfers are used. In contrast, a single Symbiote payload can compute the checksum of the entire protected host program, and does not require detailed knowledge of control-flow transfers within the host program. Therefore, the Symbiote injection process is greatly simplified and less error prone.

5.3 The Symbiote / Host Relationship

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

1. Each entity in the symbiotic relationship must have their own innate defenses. In the case of our proposed system, adaptation, randomization and polymorphic mutation will be applied to both the protected software system as well as the injected SEM's.

2. Both the Symbiote and the protected software host will be genetically diverse and functionally autonomous. Specifically, the Symbiote will not be a standard piece of software that depends on and operates within the software system it is protecting. Instead, the Symbiote can be thought of as a fortified and self-contained execution environment that is infused into the host software.
3. The Symbiote will reside within the host software, extracting computational resources (CPU cycles) to execute it's 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.
4. SEM's 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 the host software.
5. 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-fortified 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.
6. Symbiotes are moving targets. No two instantiations of the same Symbiote is ever the same. Each time a Symbiote is created and prepared for injection into a host program, its code is randomized and mutated, resulting in a vastly genetically dissimilar variant of itself. When observed at the macro level, the collective Symbiote population is highly diverse.

5.3.1 Software Symbiotes and Possible Hardware Extensions

Figure 5.1 illustrates the process of fortifying an arbitrary executable with a Symbiote. In our prior work we have demonstrated the feasibility of the software-only Symbiote, a Symbiote which is completely implemented in software and can execute on existing commodity systems without any need for specialized hardware. While the software-only Symbiote is capable of delivering the three fundamental security properties described in this section, additional hardware can greatly improve the efficiency and monitoring/mitigative capabilities of the Symbiote, as well as provide even tighter security guarantees in certain situations. Section 2.1 discusses several of such hardware extensions.

Symbiote Creation: The Symbiote is prepared for injection into the host program. A set of policies and defensive payloads are combined with a generic stub Symbiote binary. This process produces a completely self-contained executable loaded with a Symbiote execution manager, Symbiote monitoring engine, as well as the chosen set of defensive payloads and policies.

Mutation and Randomization: Both the host program and Symbiote binaries are analyzed, randomized and mutated into an unique instantiation of their original pro-

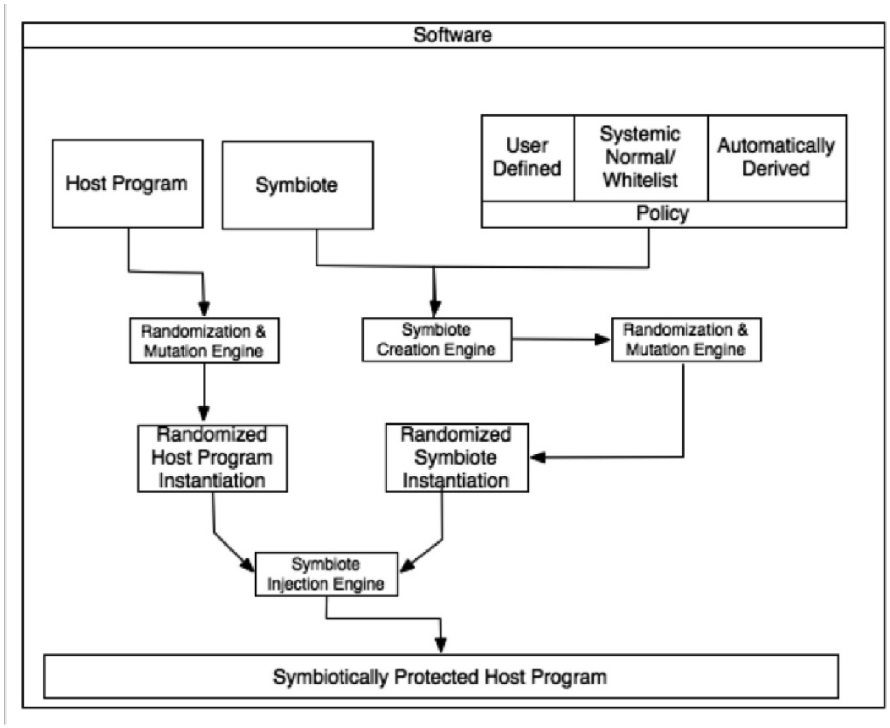

Software
Host Program
Symbiote
User Defined
Systemic Normal/ Whitelist
Automatically Derived
Policy
Randomization & Mutation Engine
Symbiote Creation Engine
Randomization & Mutation Engine
Randomized Host Program Instantiation
Randomized Symbiote Instantiation
Symbiote Injection Engine
Symbiotically Protected Host Program

Fig. 5.1 Symbiotic Embedded Machine

gram. These new binaries are functionally equivalent to the original code. However, techniques like ISR, ASR and polymorphic mutation are used to greatly increase the randomness and diversity of both the host program as well as its defense Symbiote.

Symbiote Injection: The Symbiote Injection Engine analyzes both executables and injects the Symbiote into the randomized host program, producing a single fortified program. One or more Symbiotic Monitoring Engines (SEM) can be injected into a piece of arbitrary executable code to augment the target code with sophisticated defensive capabilities. Unlike existing host-based defense and anti-virus mechanisms, SEM's do not operate on top of or as a part of the protected application or operating system. Instead, Symbiotes are essentially infused into the protected executable, providing the following four fundamental properties:

1. The Symbiote executes alongside the host software. In order for the host to function as before, it's injected SEM's must execute, and vice versa.
2. The Symbiote's code cannot be modified or disabled by unauthorized parties through either online or offline attacks.
3. 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

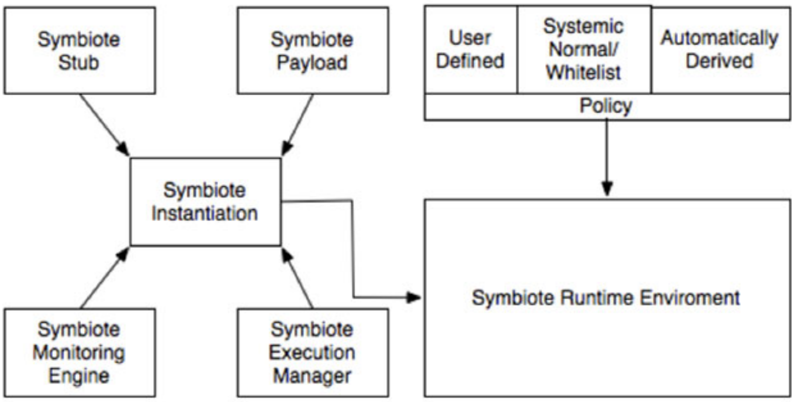

Symbiote Stub
Symbiote Payload
User Defined
Systemic Normal/ Whitelist
Automatically Derived
Policy
Symbiote Instantiation
Symbiote Runtime Enviroment
Symbiote Monitoring Engine
Symbiote Execution Manager

Fig. 5.2 Symbiotic Embedded Machine

at runtime. Hence, malware that attempts to hijack the host's execution environment cannot see the Symbiote, but the Symbiote can see the malware.

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. 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.

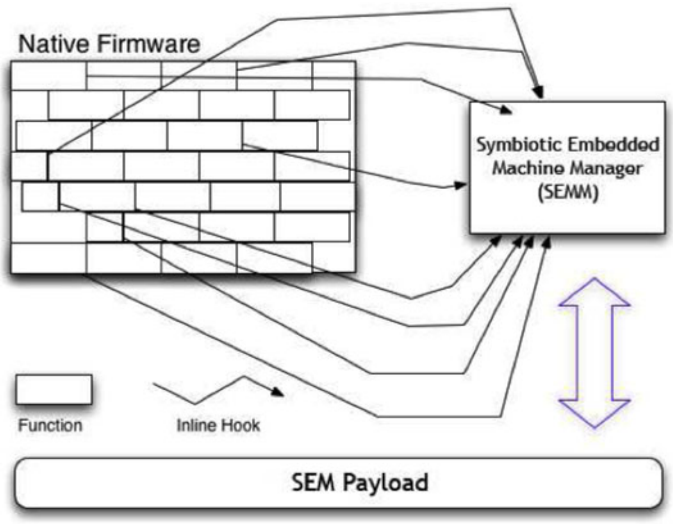
The Symbiote code structure, displayed in Figure 5.2, is modular and configurable through a standard interface. At instantiation time, a Symbiote is created by simply mixing and matching code that delivers the desired functionality from each of the following five principal components:

Symbiote Stub: The stub is the base platform of the Symbiote. It dictates how the Symbiote's code will be embedded into the host program, and how tandem execution with the host is accomplished.

Symbiote Payload: The payload is the actual defensive mechanism that is executed in tandem with the host program. Payloads are arbitrarily complex standalone executables. For example, code integrity checkers, proof carrying codes and anomaly detectors can all be implemented as a Symbiote Payload.

Symbiote Monitoring Engine: The Monitoring Engine acquires and organizes static and runtime information about the host program. It enables the Symbiote payload to fully inspect the host program, and provides an event-driven interface, allowing the payload to alert and react to runtime events within the host program.

Symbiote Execution Manager: The Execution Manager is the resource manager for the Symbiote. It controls the tandem execution behavior of the host program / Symbiote pairing. Specifically, the execution manager controls how and when



the Symbiote and the host program is executed on the CPU. Execution managers can implement different static or dynamic CPU allocation algorithms, leverage single/multi-core hardware architectures, as well as utilize specialized hardware.

Policy: The Policy is a collection of rules which the Symbiote will enforce.

The manner in which Symbiotes are injected into (legacy) host programs in a novel fashion using inline hooking. Inline hooking is a well known technique for function interception. However, the Symbiote injection process 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, the Symbiote injection randomly intercepts a large number of automatically detected function entry points. The inline hooks inserted provide as a means to redivert periodically and consistently a small portion of the device's CPU cycles to execute the SEM payload. This approach allows SEMs 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.

Figure 5.3.1 provides an overview of this injection process whereby Symbiote control code (the SEM Manager) and its executed SEM payload are dispersed throughout a binary using gaps of unused memory created by block allocation assignment.

5.3.2 Applications of Symbiotes and Further Research

The Symbiote is a self-contained code entity that does not depend on features within its host program to function. Instead, the Symbiote treats the host program as an external untrusted entity, and uses its own internal monitoring and analysis facilities to protect the host program. Since no assumptions are made about the functionality of the host program, a Symbiote can reside within any level of the software stack. Further, multiple Symbiotes can reside within the same software system as well as within the same piece of individual executable. This software defense strategy fundamentally rearranges the trust relationship and dependencies between the defense mechanism and the protected program.

The Symbiote treats all external code as untrusted software, thereby drastically reducing the amount of trust and dependence it places on the system in which it resides. The Symbiote and the Defensive mutualistic protection strategy can subsume the functionality of current security mechanisms under a new paradigm where the security software co-exist with, but completely distrusts the host program which it is protecting.

Proof Carrying Code: Proof-Carrying Code is a technique which can validate the integrity of untrusted code. Since the Symbiote is directly injected into the host program, a Symbiote payload implementing PCC can be trivially injected into arbitrary untrusted code.

Host-based IDS: The Symbiote Monitoring Engine collects and organizes the runtime information about the system in which it resides. By injecting an IDS payload into the host operating system or individual host programs, complex IDS and Anomaly Detection mechanisms can be directly injected into the host system with extremely fine granularity. Note that deploying a host-based IDS in this manner is extremely attractive because the monitoring system does not depend on the functionality provided by the operating system. Should the OS be compromised, the Symbiote's visibility into host system will remain unaffected.

Rootkit Detection: Rootkit detection using software-only Symbiotes have already been demonstrated to be feasible and effective on proprietary embedded systems like Cisco IOS and Android devices.

References

1. Microsoft Corporation, Kernel Patch Protection: Frequently Asked Questions. http://tinyurl.com/y7pss5y, 2006.
2. 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.
3. 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.

4. Ligati et al. Enforcing security policies with run-time program monitors. Princeton University, 2005.
5. Christopher Krügel, William K. Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *ACSAC*, pages 91–100. IEEE Computer Society, 2004.
6. Felix "FX" Linder. Cisco IOS Router Exploitation. In *In BlackHat USA*, 2009.
7. 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.
8. Michael Lynn. Cisco IOS Shellcode, 2005. In BlackHat USA.
9. Sebastian Muniz. Killing the myth of Cisco IOS rootkits: DIK, 2008. In EUSecWest.
10. Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Lippmann et al. [7], pages 1–20.
11. Dror-John Roecher and Michael Thumann. NAC Attack. In *In BlackHat USA*, 2007.
12. Skywing. Subverting PatchGuard Version 2, 2008. Uninformed,Volume 6.
13. 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.
14. 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.
15. Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In Lippmann et al. [7], pages 21–38.