OS Independent Run-time System Integrity Services

Abstract

A majority of critical server vulnerabilities in 2003 and 2004 were memory based [1]. Today's advanced viruses and worms attack software running in memory to circumvent operating system protections. Such attacks often disable intrusion detection systems in order to execute malicious payload. Malware can also make use of secrets stored in system memory to circumvent cryptographic security measures. An isolated execution environment contained within the system can provide substantial resiliency against such attacks in an OS independent fashion. We describe a firmware based approach that protects critical software agents from three classes of attacks, elimination, tamper and circumvention through the use of run-time integrity checking and unspoofable presence verification. We describe how this architecture can be leveraged to provide protection for secrets used by software. Finally, we describe our prototype implementation and provide the results of our experimental analysis, along with performance analysis to test the limits of this approach.

1 Introduction

As the methods for compromising the security of computer systems become ever more complex and advanced, so too have the methods for detection and remediation of such attacks. Today's commercially available intrusion prevention and detection systems (IDS) rely heavily on host resident software components such as anti-virus agents to monitor and analyze host state. In response to this model, attackers seek to disable these host resident components in the early stages of their attack, thereby allowing the attack to proceed undetected. Recent variants of the Bagle worm [22] and the Lion worm [15] are prime examples of this. Other attack approaches seek to gain escalated privileges and subsequently steal keys used by the host for cryptographic operations in order to compromise such operations. Numerous countermeasures against these attacks have evolved [4, 6, 7, 11, 13, 14]. Some of the more secure methods involve the use of an isolated execution environment for the purpose of monitoring part or all of the computing system [6, 13]. Such methods face limitations in that they do not deal with attacks that terminate the execution of monitored host memory resident IDSes, or they are vulnerable to device driver memory mismanagement or device DMA attacks [23].

We propose a system that verifies the integrity and presence[1] of host resident software agents. This architecture addresses vulnerabilities not dealt with in existing models and is capable of detecting attacks that *circumvent, tamper with or disable* critical software agents running on the computing system. We also demonstrate how our integrity services can serve to provide a generic mechanism that maintains the privacy of secrets used to perform cryptographic operations on the host. This capability prevents access to the secrets by attackers who have gained unrestricted access to all system resources (e.g. via a rootkit). Our prototype implements this architecture using the System Management Mode (SMM) on IA-32 platforms [9] as an isolated execution environment. By leveraging SMM capabilities, we demonstrate that this architecture is effective at detecting attacks against protected programs for both Linux* and Windows*. Additionally, we describe how this architecture is resilient compared to existing methods against attacks targeting the detection system itself. Finally, we demonstrate that dynamic program data can effectively be protected, a capability not provided by existing IDSes.

In section 2 we perform a threat analysis of existing and related approaches. Section 3 describes the details of our system architecture. Section 4 presents the experimental results of our prototype system, and section 5 describes how our system addresses many of the threats that remain unaddressed using current methods. The future direction of the work is outlined in section 6 and our conclusions are presented in section 7.

2 Background and Motivation

With the advent and deployment of XD based memory page level execution protection [10] for stacks and heaps, systems are capable of defending against various types of buffer overflow attacks [24, 25], attackers are faced with a higher barrier to compromise systems. Once this barrier is breached, attacks will continue to fol-

[1] By presence, we mean the verification of the execution of the program that has been checked for code and data integrity

low today's familiar pattern: first disabling security agents followed by the execution of malicious payloads. From the attacker's perspective, it is important to disable security agents because they are designed to detect such attack payloads. For example, firewalls have ingress rules to disallow some connections from remote systems. However an attacker trying to convert a system into a bot needs to be able to initiate such inbound connections. The attack solution is to disable the firewall or modify its rules. Other security agents and OS services are targeted in a similar manner.

According to 2005 CSI/FBI Computer Crime and Security Survey [2], 97% of enterprise networks in the US use perimeter firewalls, 72% of enterprises also employ intrusion detection systems. Despite of this, the same survey indicated that 55% of enterprise networks were still attacked by worms or viruses. Unfortunately, today's systems do not have a reliable way of protecting critical host agents. In this context, there is a clear need for a method to reliably detect attacks against host resident security agents. Once an attack is detected, systems can provide policy driven remediation[2]. The remainder of this section goes into greater detail about the nature of specific threats facing today's system, and how existing methods have tried to address these threats.

2.1 Related work

Today's IDSes use a variety of techniques to detect attacks. Popular methods include scanning host memory for attack signatures (pattern matching or signature matching), performing statistical analysis of host program behavior or network traffic emanating from the host, and looking for changes or additions made by attackers to critical system components.

The method used by a given IDS can typically be categorized as either a Host-based IDS, a Network based IDS (NIDS), or a hybrid IDS. A host based IDS attempts to detect intrusions while residing on the host. An example is an OS component that marks pages used for stack of software agents with the XD bit and triggers an alert when instructions from marked pages are executed. Host based IDSes also use heuristic rules for detecting intrusions. Hofmeyr et al. [8] use traces of system calls to determine whether a host is compromised. Williamson [27] proposed a host-based virus throttling system that relies on statistical analysis of traffic on the host. Even though such systems are very effective in detecting system intrusion using their complete visibility into the compromised host activity, they are themselves vulnerable to tamper by malware that can acquire unrestricted system access (e.g. via a rootkit) unless some external protections are afforded to them. The Witty worm is an example of such an attack [19]. Weaver et al. [26] propose a heuristic system implemented in network hardware. The issue with such heuristics based system is the occurrence of false positives.

NIDS on the other hand monitors multiple hosts on the network, and detects intrusion by observing anomalies in traffic activity [28]. The fact that such systems are independent of the host makes them more robust against attack from malware. However, such systems have to rely on externally observed behavior to detect intrusion, and hence may not be as effective as host-based systems which have access to context. Thus, in these approaches there is an inherent tradeoff between the visibility into the monitored host and robustness of the IDS. We attempt to address this issue in our approach.

Hybrid IDSes address this issue by running in a hardware isolated execution environment within the monitored host that is not accessible from the primary operating system. Garfinkel et al. propose such a method by using virtualization [6]. They run the monitored operating system in a guest virtual machine, and instrument the Virtual Machine Monitor (VMM) to allow rule-based monitoring of the guest OS. One of the issues with this approach is that the IDS running in the VMM is not as secure as it first appears. Notably the IDS software itself is vulnerable to attack from a DMA device as described below in section 2.2. Hence it is not a truly isolated execution environment as required by a secure hybrid IDS. Petroni et al. [13] take a somewhat different approach to a hybrid IDS. They use a programmable DMA device to scan host memory and verify the integrity of software on the system. This IDS in its documented form has some limitations - while it detects modification of protected agent images, it does not protect against the other threat vectors described in section 2.2. An additional limitation of this system is its ability to verify the integrity of user-space components is questionable, since the page tables for user-space components are not located at a single, fixed and well-known address.

[2] We explicitly do not address remediation of an infected platform in this paper.

2.2 Threat analysis

In this section, we describe issues that are not solved in existing methods that our architecture addresses. The threats described here are possible once an attacker has breached the kernel boundary to gain highest privileges on the system. Given this assumption, attackers have a diverse array of methods at their disposal to evade detection. Examples include: reading and modifying any contents of main memory, directly touching device hardware, and modifying scheduler behavior among others.

Image Modification: Attacker modifies the in-memory executable code store of an agent to cause undesired behavior (for example, disable packet filtering in a firewall module).

Disable Agent: Attacker disables the agent and prevents its execution. The attacker can kill the process, steal interrupt vectors associated with a program, modify the OS scheduler and so on. This attack is especially effective against IDSes that rely only on memory read based integrity verification without execution verification.

Out-of-context Jump: Attacker jumps into the agent code at an inappropriate location to achieve its desired results. For example, if an agent periodically signs and sends system health information to a remote administrator, the attacker could compromise it by creating fake system health information, and making the agent sign this information by directly jumping into the signing and send functions.

Transient Image Modification: Attacker modifies the monitored agent after an integrity check[3], and restores it before the next integrity check. Thus, in between integrity checks, the agent could execute malicious code.

Dynamic Data Modification: Attacker modifies the data that should ideally be modified only by the agent. Such data, for example, could include firewall/filtering rules, or any program state.

Interrupt-based Attack: Attacker modifies the state of the agent when the agent is interrupted asynchronously.

System Management RAM (SMRAM) Cache-based Attack: Attacker modifies the cached copy of the SMM handler to execute malicious code in SMM. This attack vector is important in our design, since we leverage the privileged characteristics of SMM for better protection.

Multi-processor Attack: Attacker runs code in parallel to the agent on MP systems to cause undesired effects.

Buffer-overflow Attack: Attacker exploits a buffer-overflow vulnerability to gain privileged access.

Agent Circumvention: Attacker circumvents the agent. For example, if the monitored agent consists of a network device driver with an inbuilt firewall, the attacker could use its own implementation of the driver (without the firewall), and completely bypass the firewall or introduce additional code layers that go around the firewall.

Direct Memory Access (DMA) device attack: In a virtualized system, an attacker can instruct an I/O device to perform a DMA into memory associated with another VM (or the VMM), thereby writing an attack payload into the purportedly 'isolated' execution environment.

Stealing Cryptographic Secrets: Attacker steals the cryptographic secrets used by the authorized agent (e.g., 802.11i keys) and uses these secrets to impersonate the authorized agent. Usually agents store cryptographic secrets in the memory and use ring-0 protections provided by the operating system to guard such secrets. However, such secrets are not safe from attackers that have complete control over the system (e.g., using root-kit).

3 Methodology

Given such threats as those described above, the objectives of our solution are as follows:

1. Reliably detect run-time modification of a monitored software agent.
2. Reliably detect when a monitored software agent stops executing
3. Provide a secure mechanism for run-time protection of secrets for legitimate programs from other programs

In addition to achieving these high level objectives, the system must address serious threats detailed in 2.2 not dealt with by today's systems, such as DMA based device attacks. The system must achieve this in an OS agnostic manner with minimal performance overhead. The remainder of this section describes the System Integrity Services (SIS) architecture keeping these objectives in perspective.

The operations of the SIS revolve around the key concepts of *agent locality*, *agent integrity*, and *agent execution state*. *Agent locality* is the location in memory that a program resides at (both its executable code and

[3] An integrity check verifies that the programs code and data sections are not tampered with. Our integrity checking method is described in section 3.2.3.

its static and dynamic data). Locality information can be used to ascertain the identity of a software program requesting integrity services. The SIS restricts access to confidential information so that only operations executing from the agent's known location in memory as specified during registration can make use of such information. Operations attempting to access confidential information from other locations in memory are rejected. This concept of locality is fully described in section 3.2.4. *Agent integrity* represents whether or not an agent has been modified from its original state and is described in section 3.2.3. *Agent execution state* (section 3.3) represents whether a program is running and being scheduled to run by the operating system over time. When the SIS verifies these three attributes together, then the system can infer that the agent has not been compromised by an attacker (excluding unprotected dynamic data) and as such it can be allowed to make use of any agent specific secrets protected by the SIS. Section 3.1 describes the components which make up the SIS architecture. Section 3.1 describes the integrity services operation. Section 3.3 describes how these services are used to verify agent integrity, and section 3.3 describes how they are used to verify the execution state of the agent.

3.1 System architecture

The SIS architecture consists of functional blocks shown in Figure 1.

Host Software Agent – The agent on the host to be protected. This agent could be a device driver, a kernel security agent (such as a firewall), a security service (such as VPN), an OS kernel invariant or any other program.

Integrity Measurement Manager (IMM) – The IMM is the component which performs integrity and execution detection checks on the protected host agent(s) in order to verify that it has not been tampered with and is still executing. The IMM makes use of DMA to host physical memory to get access to the data it needs to validate.

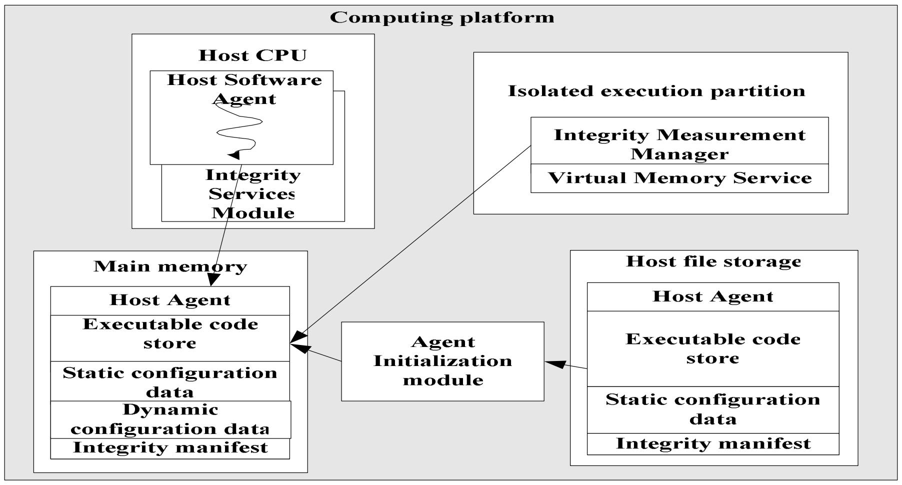

Figure 1: System Integrity Services Architecture

When the IMM detects an integrity discrepancy in an agent, it can trigger policy based remedial action.

Integrity Services Module (ISM) – This component runs in SMM [9] which is a privileged execution mode on Intel® x86 processors. It has memory sequestered from the OS which allows code running in SMM to run outside the purview of the OS. A processor enters SMM when a System Management Interrupt (SMI) is triggered. Once in SMM the processor executes code and data from SMRAM - a chipset protected region of main memory that is inaccessible to software previously executing on the processor or from programmable DMA devices. When the system enters SMM, all interrupts are disabled and system state including register contents are stored in the system saved-state map region within SMRAM. This state is then restored when the system resumes from SMM. These features afford code isolation and security capabilities critical for SIS.

Integrity manifest – This is the set of information which describes the identity of the agent. It includes information needed by the IMM to perform integrity verification. The integrity manifest is signed by the manufacturer of the program with an asymmetric key and this signature is verified by the IMM at agent registration.

Agent Initialization Module – This component is compiled into the protected host agent and it provides services to facilitate the agent's measurement. It performs memory manipulation such as pinning segments in

physical memory, provides registration with the IMM, and records relocation fix-up values so that the IMM can correctly reverse them when it performs integrity verification.
Virtual Memory Service – This component provides the ability to translate host virtual addresses to physical addresses in an OS agnostic manner from within the isolated execution partition. It interprets the IA-32 protected mode memory management features to provide this functionality.

Basic course of operation: The operation of the system is broken down into two phases – registration and steady-state. The registration sequence is shown in Figure 2.

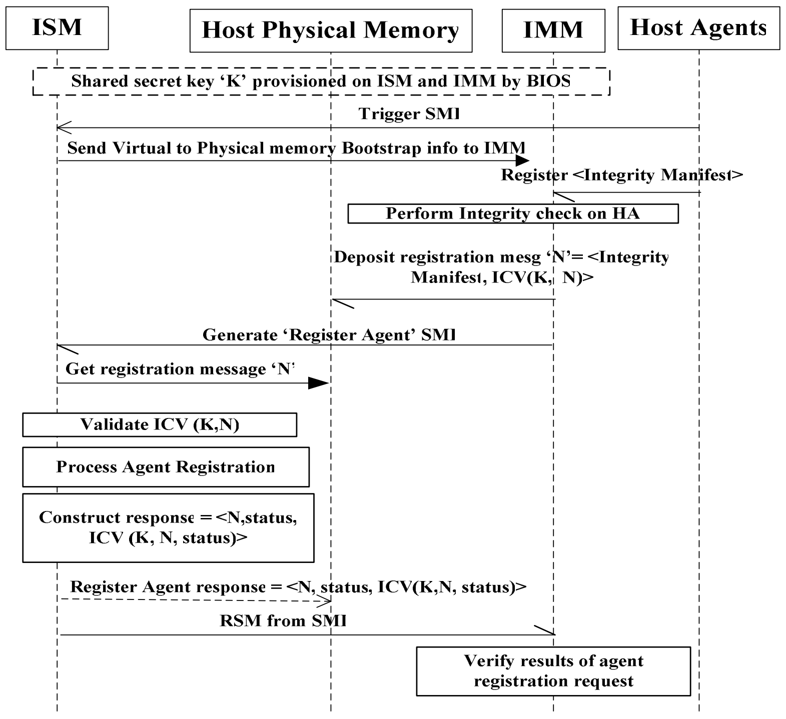
ISM
Host Physical Memory
IMM
Host Agents
Shared secret key 'K' provisioned on ISM and IMM by BIOS
Trigger SMI
Send Virtual to Physical memory Bootstrap info to IMM
Register <Integrity Manifest>
Perform Integrity check on HA
Deposit registration mesg 'N'= <Integrity Manifest, ICV(K, N)>
Generate 'Register Agent' SMI
Get registration message 'N'
Validate ICV (K,N)
Process Agent Registration
Construct response = <N,status, ICV (K, N, status)>
Register Agent response = <N, status, ICV(K,N, status)>
RSM from SMI
Verify results of agent registration request

Figure 2: Protected agent registration sequence

All host agents must register successfully with the ISM before the agent can use the integrity services. Every host agent is uniquely identified by an identifier defined during registration. The host agent begins the process by sending a registration message to the IMM. This provides the IMM with the virtual address ranges that comprise the host agent's measured segments. The IMM then verifies the integrity of the measured segments. Upon successful integrity verification of the host agent, the IMM registers the host agent with the ISM. The ISM records the host agent in its internal registered agent list that is maintained in SMRAM. The course of operation for integrity verification and presence verification are described in sections 3.2 and 3.3 respectively.

3.2 Data Integrity Services

The SIS provides general purpose data integrity services that are used as the foundation for implementing complex operations. These general purpose services include generation of cryptographic integrity check values (ICV) and access to secrets for other cryptographic operations. In addition, the SIS can perform tests that verify the caller's integrity and locality. These tests are used to determine if the caller is authorized to initiate a given operation. This allows the SIS to use the keys on behalf of the agent without divulging keys to the agent.

3.2.1 Host Agent Initialization

Every monitored agent includes an SIS specific initialization code section, which is executed when the agent is loaded. This initialization code is responsible for sending out a registration message to the IMM. The agent registration message includes the locations of the critical sections in host virtual memory, and the location of the agent’s integrity manifest. The agent integrity manifest is described in the subsequent section.

3.2.2 Integrity Manifest

The integrity manifest provides a signed summary description of the critical components of the host software agent when the agent is loaded into host memory. The appearance of an agent in memory depends on two factors: the original contents of the sections that comprise the agent in the disk image and the relocations performed on these sections by the OS loader. Thus, the integrity manifest describes these two entities for each of the monitored sections that comprise the agent.

The integrity manifest is made up of three tables: the section table, the relocation table, and the symbol table. Each entry in the section table includes a unique numeric identifier for the section, a bit-field describing properties of the section[4], and a SHA1 hash of the un-relocated image of the section. Each entry in the symbol table includes a unique numeric symbol identifier, the section to which the symbol belongs, and an offset of the symbol in that section. Each entry in the relocation table includes a numeric identifier of the section to which the relocation applies, an offset of the location in the section that needs to be relocated, a numeric identifier of the symbol on which the relocation is based, and a numeric relocation type indicating the relocation action corresponding to that entry. These relocation types mirror those defined in the Executable and Linkable Format (ELF) specification [20]. The Windows* Portable Executable (PE) [12] format supports only one relocation type that can be mapped to one of the ELF relocation types with slight modifications to the relocation semantics. This allows the manifest to remain OS agnostic.

The integrity manifest includes several other necessary fields. A header at the beginning of the manifest provides offsets and sizes of the various tables described above. Additionally, the header contains the size of the manifest and a signature for proving the integrity of the manifest itself as it is transferred to the IMM.

3.2.3 Integrity Verification

The integrity of a software agent must be verified over the course of its in-memory lifetime. This is necessary in order to detect run-time modifications. The first integrity check must be performed at registration time when the agent starts. After this, the agent must be checked periodically, or on a use-driven basis as it executes. Software agents are comprised of a number of critical memory regions that must be verified including executable code as well as static and dynamic data. The SIS can be used to protect critical memory sections from unauthorized access by computing an Integrity Check Value (ICV) or a Message Authentication Code using a secret key stored in SMRAM and an appropriate hash algorithm such as SHA-1 or SHA-256 [5] over the specified data buffer. The signature is computed using: *ICV = hash ((key) || hash (data buffer))*

The first integrity verification of an agent occurs when that agent is loaded. When the IMM receives a registration request from the host agent, the IMM reads the integrity manifest from host memory using the Virtual Memory Reconstitution Service (VMRS), described in section 3.2.7. If the signature on the integrity manifest is verified as correct, the IMM then reads the sections of the monitored agent as specified in the manifest. Using the relocation table provided in the manifest, the IMM reverses the relocation fix-ups in each section. It then computes per section ICV and compares it with the ICV from the appropriate section entry in the section table of the integrity manifest. If all of these match, then the IMM asserts that host software agent has not been modified from the time it was signed by the manufacturer.

Subsequent integrity verifications are performed when the software agent requests data integrity services via an SMI. At this time, the IMM is triggered to verify the agent’s integrity prior to allowing the agent to complete an integrity services operation. An optimization here is for the IMM to verify the integrity of only a small portion of the agent (a page of executable code in memory). This optimized check, while not as robust as a full verification, still ensures that the immediately relevant portion of the program has not been modified by

[4] This field could be used to denote whether the section stays unmodified, or whether it changes with agent execution among other things.

an attacker. An alternate, more robust, yet reasonably well-performing method would be to use a data and control flow model such as [7] to determine the relevant set of the program on which to perform an integrity check.

3.2.4 Locality Verification

Locality Verification by the ISM is necessary to prevent spoofing of requests to the IMM. It is used to ensure that an agent requesting services is the same agent that registered for those services. The ISM can identify the source of an SMI and subsequently the program that triggered the SMI. Once the source of an SMI is identified, the ISM can verify if the program requesting ISM services is a registered program as previously recorded in the registered agent list. The ISM can identify and verify the source of an SMI as follows-

- When the system enters SMM, the CR3, CS and EIP register values corresponding to the program instruction that triggered the synchronous SMI is recovered from the System Saved-state Map in SMRAM.
- The logical address of the instruction triggering the SMI is translated into a corresponding virtual address.
- Once the virtual address of the source is determined, the corresponding program is identified. To accomplish this, the virtual address of the SMI is compared with the virtual address ranges of registered programs.

3.2.5 Key Management

The SMRAM is used for storing the keys used by the SMM Security Services module for its various cryptographic operations. A platform master key is provisioned on the IMM running in the isolated execution environment and the ISM during the pre-boot phase prior to OS startup. The platform master key is then used for deriving session keys for various cryptographic operations.

3.2.6 ISM operation

Every host agent that wishes to use the ISM must trigger an SMI of the appropriate type. This is a two-step process. In the first step, the host agent notifies the ISM that it intends to use integrity services via an SMI call. In the second step, the host agent fills in the input data buffers over which the ISM needs to perform the cryptographic operations and sends that through a second SMI invocation. These two steps cannot be merged because the data buffers are in host accessible memory and can be potentially corrupted by malware after a host agent has filled it up and before an SMI is triggered. One straightforward way for the host agent to protect against this threat is to disable all interrupts after filling in the data buffers and to then trigger an SMI. Disabling interrupts is not desirable for stability reasons. To address this, we introduce a token-based protocol that handles race conditions due to context switches that may be exploited by malware without needing to disable interrupts. Our protocol also allows for reentrant and multi-threaded host agents to update the data buffers. This approach is not as secure as disabling interrupts around the critical operation that creates the data. It however improves stability and limits the threat to a difficult to achieve race-condition attack. The two stages are as follows:

Stage1: Access Start Operation Sequence: The host agent triggers an *Access Start* SMI. In response to this notification, the ISM does the following steps: a. It verifies the locality of the agent that generated the SMI, b. It generates a token for host agent as follows: Token = hash (token-key || token-nonce || host agent ID) where token-key is a derived key and token-nonce is incremented only when the per host agent outstanding token-counter is zero. The ISM then increments the token counter to indicate an outstanding token for the specific host agent. Finally, the ISM returns the token to the host agent. This transfer of tokens can be done via processor registers directly so that such information is never lingering in memory.

Stage2: Integrity Operation Sequence: The host agent updates buffers that hold the data it needs to be operated on. The host agent then triggers a *do-operation* SMI notification. As part of this request, the host agent provides the ISM with the hash of the updated data buffers as well as the token that was provided to it in the first stage. Again as before, the tokens can be returned from processor registers directly reducing the possibility of in-memory attacks. In response to the do-operation SMI Notification, the ISM does the following: a. It verifies the locality of the agent that generated the SMI, b. It picks up the request and token from the data buffer associated with the host agent, c. It verifies if the token returned by the host agent was the same as what was generated in Stage1. If tokens mismatch, the ISM skips the next step of computing a signature using a data signing key. The ISM then must decrement the number of outstanding token counters and then checks if the outstanding token counter is 0. If the token counter resets to zero, it increments the token-nonce by 1 otherwise it returns the result of the requested operation and resumes from the SMI. Lastly, the host agent picks up the signature returned from the ISM.

3.2.7 Virtual Memory Reconstitution

The Virtual Memory Reconstitution Service (VMRS) maps host virtual addresses to physical addresses while running on the isolated execution partition. It relies on information such as the CR3 register value provided by a bootstrap agent running in the ISM. The VMRS has DMA access to host memory. It reads and parses the host page tables present in memory to translate virtual addresses to physical addresses. The host page tables consist of the Page Directory Pointer table (PDPT), the Page Directory table, and the Page Entry Table. These are maintained by the host OS in conjunction with the processor. They are used by the processor to translate virtual memory accesses to physical memory accesses. In order to read protected host agent segments that are several pages (4K size) in length, multiple translations are done for each segment when performing integrity verification operations.

The Host Bootstrap Agent (HBA) provides the VMRS with information about the host processor registers needed for reading the host page tables. These registers include the GDTR (Global Descriptor Table Register), and CR3 (Control Register 3) which provides the base address of the PDT or PDPT. This information along with the virtual or logical addresses, allow the VMRS to translate the addresses.

3.2.8 Updates to protected dynamic application data

Some pieces of dynamic data used by a program are particularly critical, high value targets. Examples would be packet filter rules associated with a firewall, or software version information associated with an anti-virus application. If the firewall policies were modified, it would be possible to gain unrestricted access to the network, leading to secondary attack vectors. These examples clearly demonstrate the importance of protecting these critical pieces of data, and detecting when an attacker modifies them.

The challenge in protecting such data is that it is also modified by authorized users after the agent is running. In SIS, the IMM plays the role of verifying changes to protected dynamic data. The sequence used for updating dynamic data is shown in Fig. 3. The host agent first generates the changed value and then it uses the SIS data integrity services to generate an integrity check value for the new data. This integrity check value is generated on behalf of the agent only if the locality is verified correctly. Due to the performance overhead associated with changes to protected dynamic data, it is not reasonable to protect all of an agent's dynamic data. Rather, only critical application data should be protected.

3.3 Agent execution state detection

One critical service the ISM provides is that of securely updating and verifying host agent execution state. In order to prove that the host agent is executing, it needs to periodically generate an indicator to the IMM that it is alive and running. Unfortunately, sending these "heartbeat" messages from the host is an inherently insecure endeavor, due to the fact that host available resources are vulnerable to attackers which have gained privileged access levels on the host. Any keys used for signing the heartbeat messages must at some point be placed in memory in order for the host agent to perform operations with them. Once the keys are host accessible, they can be used by other software programs with the appropriate privileges. Here, we describe our approach that allows for the IMM to detect the execution state of host agents conveyed through heartbeat messages in a reliable, confidential and un-spoofable manner.

Each time the host agent needs to send a heartbeat message, it requests the ISM to generate an ICV over the heartbeat message using the token-based approach discussed in section 3.2.6. The signature is generated by the ISM as follows: *HB Signature = Hash (Data-Sign-Key || host agent ID || nonce || hash (heartbeat message)).* The inclusion of the host agent ID in signature generation ensures that per-host agent unique signatures are generated. The signature is returned to the host agent, which then appends it to the heartbeat message that is sent to the IMM. This signature is then verified by the IMM. The signature ensures the integrity of each heartbeat message. The fact that the correct signature can only be appended to messages whose locality and integrity has been verified by ISM ensures the authenticity of the heartbeat message. If an attacker tries to forge the heartbeat message, it would be detected due to an incorrect signature.

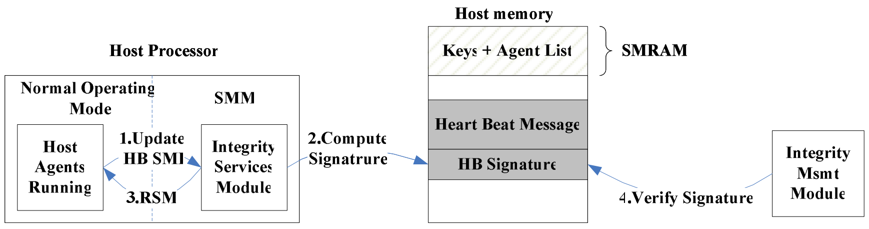

Figure 3: Detection of host agent execution state

3.4 Handoff of secrets

The ISM can also hand off derived keys to the host agent if it so desires, to use its own cryptographic algorithms. The token protocol described in section 3.2.6 is used to achieve this handoff using registers after agent locality is verified. The agent may disable interrupts during this operation. Also platform interrupt service routines (ISRs) can be modified so that on interrupt these critical registers are cleared so that these secrets are not divulged due to scheduler or external interrupts. Additionally, these modified ISRs can be integrity checked.

4 Effectiveness

Our prototype implements the SIS architecture described in section 3. Using this prototype we performed several measurements in order to determine the architecture's effectiveness relative to existing methods, as well as its performance impact on the system. This section presents these results.

It is important to note that recent worm attacks like, Slammer, Code red, Blaster and the Witty worm are all memory-resident worms that infect kernel components in the operating system. The affected kernel components can be checked for liveness and tamper using the SIS approach. Another key metric determining the effectiveness of the system is the amount of time it takes before detecting an infection. This is particularly important for these fast spreading day-0 memory resident worms like Slammer [21] and Blaster [17] and for worms like Witty that carry a damaging payload. We observed the following processing requirements for the SIS to verify the integrity and liveness of a typical software network driver implementing firewall and worm heuristics. This information was captured using Intel® VTune™ Performance Analyzer on our test platform over 100 runs. The number of instructions needed for complete integrity verification was 98766 on average. With a Clocks-per-instruction of 1.93, the number of clock cycles was 190618. The test platform processor was clocked at 1.5GHz implying a window of 0.13 milliseconds to verify integrity and locality (heartbeat verification). Additionally, the turnaround time for an empty SMI handler is 70 microseconds. This gives us a total time of 0.2 milliseconds to integrity verify the network driver code and static data.

Now, consider the case of a fast worm like SQL Slammer that propagates at a bandwidth limited rate of 26,000 unique connections per second [21]. Also consider a simple UDP address-scan worm propagation heuristic that is run on the platforms Network Interface Card (NIC) as shown by Weaver [26] which can track outgoing UDP packets to unique addresses. With such a heuristic it is possible to detect worm-like traffic with a heuristic setting of 8 unique connections within a time window of 1 millisecond. The issue with such a heuristics approach in general, is that it can generate false positives. Using the SIS approach, the integrity of the host firewall can be verified to ensure that the anomalous behavior observed in traffic is not a false positive. If a SIS integrity check for the firewall is triggered due to events from such NIC heuristics, for this worm speed, an additional 5 connections of the worm (over the initial 8 connections) will manage to leave the infected platform. With a sufficiently sparse address space it can be deduced that this increase will not increase the overall penetration of the worm in the network. The SIS integrity checks increase the effectiveness of such a system. Similarly it follows that various other platform heuristics can interact with the SIS system to affect platform policies based on the integrity of critical software services on the platform.

An important application of the SIS is the capability to allow critical host applications to perform cryptographic operations based on their code integrity and locality. A general example of such an application is a network service layer signing network packets as it deposits them into memory for the next service layer to pick them up. Such a handoff mechanism avoids packet insertion attacks which are a common method of circumventing operating system protections. For our experiment, the NIC device driver invokes the heartbeat which triggers an integrity check of the driver and can be used to sign network frames. Access points or gateways can then verify the integrity of frames before forwarding frames to the network. Since SMIs effectively pause the operating system we measured the effect of requesting an integrity check (and packet signature) at different intervals to measure the impact on network throughput. The side of the network device driver whose code store we measured was approximately 2.8MB in size when loaded into memory. The throughput tests were run using NTttcp with default buffer size setting of 64K and number of buffers set to 10000. The test was performed on two different machines as shown below:

Processor speed	3.0 GHz	
Baseline throughput	550	Mbps
w/ intermediate driver -no SMI	542	Mbps
w/ SMI on every nth packet	Mbps	% of baseline
1	162	29.89%
2	308	56.83%
10	422	77.86%
50	524	96.68%
100	528	97.42%

Table 1a: Network throughput for SIS verified driver

Processor speed	2.4GHz	
Baseline throughput	518	
w/ intermediate driver no SMI	480	
w/ SMI on every nth packet	Mbps	% of baseline
1	101	21.04%
2	181	37.71%
10	325	67.71%
50	428	89.17%
100	468	97.50%

Table 1b: Network throughput for SIS verified driver

The graph below illustrates that close to 70-80% of network throughput is obtained when the driver requests an integrity check every 10 packets. The intent is to close down the time window for worm spread like activity. An integrity check failure on the signature of the packet deposited by the IMM can signal a potential attack on the platform. Additionally, the driver can randomize which packets are chosen to be signed (and the driver itself verified) to disallow attackers to take advantage of the periodicity of the requested integrity check.

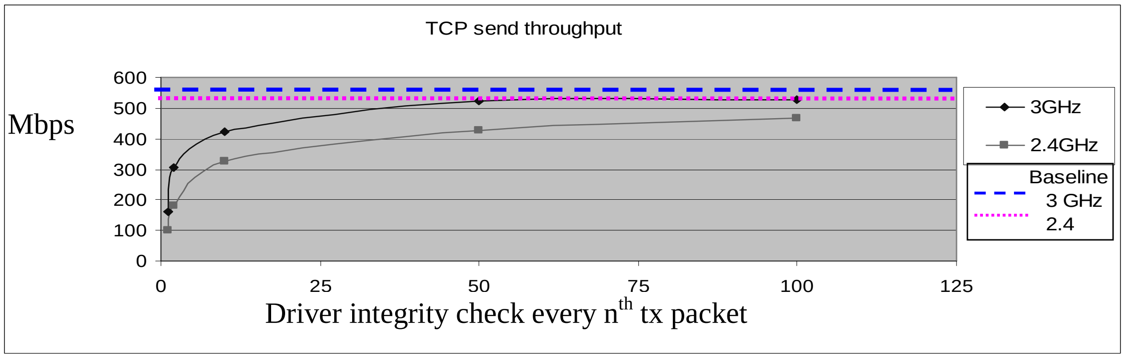

Figure 5: Net throughput graph for SIS verified driver

The SIS system can also be applied to detecting root kit installations by monitoring OS invariants and detecting changes which occur when a root kit is installed, as long as the root kit extensions requires modifying at least one byte of the monitored components. In this respect, the SIS based method is a different approach from other work on detecting root kits using file or kernel memory diff based methods as recommended in Strider [29] although they also propose the use of memory inspection from an isolated environment. A potential issue with the diff based approach is the large amount of time taken to complete the analysis, considering current worm propagation speeds. Our current performance measurements cover verification of targeted kernel components. Performance analysis of an entire kernel for root kits is left for future analysis. We think that the manifest based integrity verification mechanism used by SIS offers an easy to deploy yet capable solution for detecting image modification at runtime.

Finally, SIS can also be used effectively to perform encryption, decryption, digital rights management or any other cryptographic operations that require secrecy of some part of the data used in the operation.

5 Discussion

The SIS model is designed to deal with a wide range of threats not addressed by existing models as described in section 2.2. In this section we describe how the features of our design address the attack vectors described earlier and also describe some issues that remain unaddressed.

5.1 Threats addressed by SIS

Image modification: If agent code is modified in memory, SIS will detect this on subsequent integrity checks.
Disable agent: If the agent is disabled, the attacker cannot produce the secure heart beats and the IMM will detect that the valid heartbeats have stopped.
Out-of-context jumps: This attack is partially addressed through the use of a token based SMI entry method. If necessary interrupt disabling can also be used.
Transient image modification: SIS requires the agent to trigger an SMI to perform the critical operations. Also, the ISM performs one-page code-integrity verification around the point from where the SMI is triggered. Thus, it becomes difficult (though not impossible) for the attacker to exploit this attack vector to get access to critical operations. Additionally, if the agent uses a code-obfuscation techniques [30] it becomes practically impossible to modify other code pages without touching the code page around the SMI trigger.
Dynamic data modification: SIS addresses this threat vector by signing and verifying the source and validity of each dynamic data update.
Interrupt-based attack: Our system addresses this problem by disabling interrupts and verifying that interrupts were in fact disabled from within the SMI handler, otherwise the operations are not performed.
SMRAM cache-based attack: SIS thwarts this attack by ensuring that the SMI handler flushes the appropriate lines of the cache before exiting.
Multi-Processor system attack: SIS addresses this attack by performing critical operations (such as signing) inside SMM, thus stopping all the other threads. Further, the agent can always stop other cores if it is performing a critical operation that requires privacy from other cores. Finally, if the agent uses code-obfuscation, then it becomes virtually impossible for other threads to attack the agent using this race condition.
Buffer-overflow attack: SIS does not *directly* address this attack vector. We expect buffer overflows to be addressed by use of software approaches such as Stackguard [3]. Additionally, we expect hardware approaches such as the XD bit to address attacks that attempt to execute code from the stack. However, we think that if the XD functionality is enabled, then the most-likely attack vector that would be used by malware is to modify the agent's dynamic data. Since SIS can verify dynamic data state, this attack vector is likely to be detected. Additionally, if a buffer-overflow attack modifies any SIS monitored components, the attack will be detected.
Agent circumvention: In SIS, the agent must trigger an SMI to perform critical operations such as signing or obtaining keys. The agents could easily tie these operations to their critical functionality. For example, 802.11i layer or VPN service can use SIS where an external AP or Gateway verifies the integrity of the packets.
DMA device attack: The ISM is not vulnerable to DMA based device attacks because devices cannot access SMRAM due to hardware restrictions.
Stealing Cryptographic Secrets: In our design, it is extremely difficult to steal the cryptographic secrets, since secrets are stored in SMRAM that is not accessible to even the highest privilege software running in the context of the OS. As described the ISM uses the combination of entry-point verification and code verification to ensure that such secrets are not divulged to unauthorized programs.

5.2 Threats unaddressed by SIS

Application specific attacks - There are a number of attacks that are not amenable to discovery at the level at which our system operates. An examples attack is one that does not touch the measured agent code or data but modifies the data buffers the agent operates on before or after the agent gets a chance to access them. Examples of such attacks for a NIC are ring buffers being written into after they have been inspected by a measured firewall agent. Similar attacks also attack the standard OS service or API hooks between layers to introduce data or control frames to influence incorrect operation of the measured agent without modifying their code or static data. Potential SIS based countermeasures against such attacks are application specific - for example, applying integrity check values to packets as they are being transmitted from one software service to another on the same platform or across platforms.

Out of context jumps: Our approach does not sufficiently address attacks where rogue software issues an out-of-context jump into measured agent code. For example, malware may manage to jump into measured agent code which issues the SMI to deposit a signed heartbeat for the management controller. Potential countermeasures against this approach are to use a control flow check service running in ISM that performs a control flow check at the same time as it performs an integrity check. Other possible countermeasures are disabling interrupts or modifying ISRs to clean up critical data when control is transferred to another program.

6 Future work

Addressing open threats: We are considering the countermeasures proposed in the discussion such as changing the polling based model to add components of an event driven protection model versus a reactive-only model. Such an event driven model allows for integrity verification when specific memory gets accessed, and it allows checking critical pieces of code in parts to improve performance.

Extended protection: SIS can be extended to protect critical pieces of stock kernels such as the Interrupt Vector Table. It can also be applied to measure Ring-3 applications. Additional extensions can be made to SIS to provide additional protections for shared libraries and Dynamic Load Libraries (DLL's).

Endpoint Access Control: The integrity assurances that SIS makes can be reported to a software *trust agent* on the host that communicates with access control servers in the network for end-point access control. Network based policy decision points can evaluate the posture of the end-point based on data coming from a tamper-resistant evaluation entity instead of a purely software approach, which is susceptible to attacks itself. Additionally, the separation of the platform measurement component from the software being measured allows the policy decision points to create a security association with the measurement module, which even the trust agent cannot snoop on. The measurement data reported to the decision points can therefore not be tampered with.

Other Platform security tools - Our approach complements other platform approaches such as software based intrusion detection systems and hardware firewalls. Platform firewalls can provide continuous monitoring of platform traffic to block the offending platform if an attack tries to take advantage of the periodicity of our integrity verification. Additionally, our method can signal integrity failure events to such platform intrusion detection engines which can increase the level of filtering/monitoring they perform. Such specific feedback to platform firewalls from the integrity measurement module can also prepare the platform for automated remediation by either (re)moving the platform to a remediation VLAN, allowing connections to only specific remediation URLs or opening specific ports for remediation from management consoles.

7 Conclusions

We have presented a method for reliable run-time detection of tampering, disabling and circumvention attacks against protected software agents in a system. By leveraging the capabilities of SMM in IA-32 processors it is possible to determine when attackers prevent a protected program from executing, and when a protected program is modified by an attack. The use of SMM also enables this method to be independent of the Operating System running on the host system. Our prototype of this method demonstrates that the architecture is practical in terms of performance and for detecting many real threats currently unaddressed in existing systems. Experimental results show that using minimal packet sampling the network performance for a network driver protected via this method was 70-80% of the observed baseline rate. This approach can thus be practically applied to performance sensitive communications security services including firewalls and VPN services.

References

[1] Determina Inc. *Memory Firewall* FAQ*. http://www.determina.com/tech/memfirewallfaq.asp

[2] CSI/FBI Computer Crime and Security Survey. http://www.usdoj.gov/criminal/cybercrime/FBI2005.pdf

[3] Crispin Cowan et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*. In Proceedings of the 7th USENIX Security Symposium, January 1998.
[4] L. Catuogno, I. Visconti. A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code. *Proceedings of the Third International Conference on Security in Communications Networks,* 2002
[5] Eastlake, D.,Jones, P., US Secure Hash Algorithm 1(SHA1) *RFC 3174,* Sept 2001
[6] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection based Method for Intrusion Detection. *Network and Distributed Systems Security Symposium Conference Proceedings*, 2003
[7] J. Giffin, D. Dagon, S. Jha, W. Lee, B. Miller. Environment-Sensitive Intrusion Detection. *Proceedings of The 8th International Symposium on Recent Advances in Intrusion Detection*, Sept 2005
[8] S. A. Hofmeyr, A. Somayaji, and S. Forrest. *Intrusion Detection using Sequences of System Calls* Journal of Computer Security Vol. 6, 1998
[9] Intel Corporation. *Intel Architecture Software Developer's Manual* Vol. 3, 2001
[10] Intel Corporation. *Execute Disable Bit Software Developer's Guide,* June 2004
[11] G. H. Kim, E. H. Spafford. The design and Implementation of Tripwire: a System Integrity Checker. *Proceedings of Conference on Computer and Communications Security,* November 1994
[12] Microsoft Corporation. *Portable Executable and Common Object File Format Specification*, February 1999.
[13] N. Petroni, T. Fraser, J. Molina, W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. *Proceedings of the 13th USENIX Security Symposium*, August 2004
[14] E. Shi, A. Perrig, L. Van Doorn, BIND: A Fine-grained Attestation Service for Secure Distributed Systems, *IEEE Symposium on Security and Privacy,* 2005
[15] Symantec security response. Lion.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/linux.lion.worm.html`
[16] Symantec security response. W32.Beagle.BA@mm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.beagle.ba@mm.html`
[17] Symantec security response. W32.Blaster.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html`
[18] Symantec security response. W32.SQLExp.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.sqlexp.worm.html`
[19] Symantec security response. W32.Witty.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html`
[20] Tool Interface Standard. Executable and Linking Format Specification v.1.2, May 1995.
[21] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, Nicholas Weaver. *Inside the Slammer worm.* IEEE Security & Privacy, Vol 1 No. 4, Jul-Aug 2003.
[22] Sophos report: Bagle-AU worm disables Windows XP* SP2 firewall. October 29, 2004 http://www.sophos.com/virusinfo/articles/bagleaufw.html
[23] Microsoft WHDC. *Testing for Errors in Accessing and Allocating Memory.* June 10, 2004 http://www.microsoft.com/whdc/driver/security/mem-alloc_tst.mspx
[24] Aleph One, *Smashing the Stack for Fun and Profit*, Phrack, Volume 7, Issue 49
[25] Bulba and Kil3r, *Bypassing StackGuard and StackShield*, Phrack, Volume 5, Issue 56
[26] Nicholas Weaver, Stuart Staniford, Vern Paxson *Very Fast Containment of Scanning Worms.* 13th USENIX Security Symposium Pp. 29–44
[27] Jamie Twycross and Matthew M. Williamson *Implementing and Testing a Virus Throttle.* 12th USENIX Security Symposium Pp. 285-294
[28] Martin Roesch *Snort - Lightweight Intrusion Detection for Networks* Proceedings of the 13th USENIX conference on System administration. Pp. 229 - 238
[29] Yi-Min Wang; Doug Beck; Binh Vo; Roussi Roussev; Chad Verbowski. *Detecting Stealth Software with Strider GhostBuster.* Microsoft Research MSR-TR-2005-25. February 2005
[30] Cloakware* http://www.cloakware.com ·

· Intel x86 processor and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Copyright © Intel Corporation 2005. *Other names and brands may be claimed as the property of others.