Code Validation for Modern OS Kernels

Thomas Kittel
Technische Universität München
Munich, Germany
kittel@sec.in.tum.de

Sebastian Vogl
Technische Universität München
Munich, Germany
vogls@sec.in.tum.de

Tamas K. Lengyel
Technische Universität München
Munich, Germany
tklengyel@sec.in.tum.de

Jonas Pfoh
FireEye, Inc.
Wilsdruffer Str. 27
Dresden, Germany
jonas.pfoh@fireeye.com

Claudia Eckert
Technische Universität München
Munich, Germany
eckert@sec.in.tum.de

Abstract

The proliferation of kernel mode malware and rootkits over the last decade is one of the most critical challenges the security industry is facing. While mechanisms such as UEFI secure boot in conjunction with signed driver loading effectively verify the integrity of the kernel at load time, runtime verification is still an open problem. Various security systems have been proposed solutions to protect the integrity of the kernel by performing hash-verification of code-pages. This approach requires one to keep track of a potentially large set of hashes. Other approaches that attempt to protect code-pages usually do so by heavily restricting the OS from performing otherwise benign optimizations at run-time.

In this paper we present an approach for syntactically verifying the integrity of kernel code with the use of semantic (binding) information. By leveraging virtual machine introspection, we examine all kernel code pages at runtime to verify their contents and to reconstruct the active system state. By emulating the OS's patching mechanisms, our system successfully differentiates between malicious and benign code changes. We demonstrate the ability to detect malicious kernel code with a set of rootkit samples. Our method does not restrict modern OS kernels from using otherwise benign patching routines. To further highlight the importance of practical kernel code validation, we also present a critical security issue in the Linux kernel that we discovered in our research which thus far remained unnoticed.

1. INTRODUCTION

The integrity of the OS kernel is crucial for the security of the entire system. If the kernel gets compromised, the attacker can further disable existing protection mechanisms and take full control over the system and all applications running on it. This makes the kernel a very lucrative target for malware authors.

In order to infect the kernel, malicious software generally loads itself into an area of memory reserved for the kernel. This can be accomplished by exploiting a vulnerability or loading a malicious module or driver. In either case, the result is the inclusion of malicious code in kernel space. Effectively the code base of the running kernel has changed. Such a change can generally be observed by an external entity and thus detecting such changes to the code base lends itself to malware detection.

A large body of research has focused on enforcing kernel code integrity by creating a white-list of hashes of the kernel's code-pages [6, 3]. While approaches that do not depend on hash-comparison, such as MoRE [11], have better performance and can ensure that code-pages remain static during the execution of the OS, they prevent the kernel from applying otherwise benign optimizations at runtime. Considering that the Linux kernel will soon introduce new features such as JIT code and dynamic security patching, the short-comings of these approaches will become even more limiting in the future.

In this paper we take an in-depth look at the limitations of current runtime integrity verification techniques. We then highlight patching mechanisms in the Linux kernel that could be abused by an attacker to modify kernel code pages in a manner that are particularly difficult to detect. Solving this challenge requires a deeper understanding of the kernel's various load and runtime self-patching mechanisms. In fact, there are several mechanisms for which the integrity and consistency of a change is verifiable but the resulting state still could be malicious. We also present a novel method to perform runtime kernel-code integrity checking that addresses the short-comings of existing systems. In experiments we show that our prototype implementation is both effective against kernel-level malware and efficient, which makes it well-suited for real-world applications.

In summary we make the following contributions:

- We show that current code validation techniques are not suitable to validate the code integrity of modern kernels.
- We examine various load time and runtime code patching techniques employed by modern OS kernels and discuss the challenges that they create for code validation.
- We demonstrate the importance of correctly validating modern kernel code. We do this with a practical example that enables an unprivileged *user* to load arbitrary executable code into the Linux kernel.
- We introduce a framework that can successfully validate the integrity and identity of dynamic kernel code and enforces additional security constraints.

2. RELATED WORK

As there is a plethora of work that is focused on kernel integrity validation in the following we only highlight the major directions in runtime validation.

One of the first systems that made use of a hash-based approach was Copilot [6]. Copilot was designed to calculate hashes of all Linux kernel and module code regions to detect malicious modifications. To achieve this, Copilot makes use of trusted hardware that is capable of calculating and comparing the hashes at runtime. The "good" hashes, which are required as a basis for the detection of modifications, are obtained by calculating the hashes in a system state that is considered to be non-compromised. Seshadri et. al [9] implement a similar hash-based approach to detect kernel rootkits with their system, Pioneer. In contrast to Copilot, however, Pioneer does not require special hardware.

SBCFI [5] also computes hashes to validate kernel and module code regions, but moves the validation component out of the guest system with the help of a hypervisor to increase the isolation of the validation component. To provide a base for the comparison they make use of a trusted store that contains all trusted executables. Binaries within the trusted store are relocated before the hash is compared based on their current location within the guest system. The problem with such an approach is that the set of possible hashes can grow arbitrary large during runtime.

Instead of validating kernel code another branch of research focuses on hindering writes to kernel code entirely. SecVisor [8] forbids writing to code pages by leveraging a hypervisor to trap write memory events. NICKLE [7] on the other hand redirects the instruction fetches to kernel code to a secure shadow version kept within the virtual machine monitor (VMM). The most recent approach, MoRE [11], splits the TLB so that data accesses, such as write attempts, point to a different memory location then the actual code pages. While the specifics on how they enforce static OS code pages differs, these approaches all come with the same penalty: legitimate kernel patching mechanisms, which are often there to improve performance, are disabled. For this reason Ianus [4] advocates for only a partial enforcement against kernel patching where kernel modules are restricted from modifying code that does not belong to the module itself. Also Srivastava et al. [10] introduced a system to restrict untrusted modules to modify the code pages of the core kernel. However, without understanding the specific changes that happen to the main kernel code, such restrictions can be easily circumvented.

The only previous work that considers configuration-specific patching is Patagonix [3]. At its core, Patagonix is also a hash-based validation system in which the hypervisor stores a hash of all valid code pages. Patagonix makes the assumption that patching generally only occurs during load and early boot time. Load time patches are handled with the help of a list of all possible memory locations that can be updated, and a list of all possible values for each memory location. This information is extracted from each binary before the validation process. Furthermore, the code validation relies only on *binding information*, that is, they do not rely on any "semantics implied by source and symbol information". While we agree that it is important to handle non-binding information carefully, as malicious code is not bound to this information, we argue that one *must* consider non-binding information to be able to properly validate the dynamic runtime changes conducted by modern OS kernels. For example, many of the runtime patching mechanisms used by the Linux kernel depend on the current software state of the running system. This information is non-binding by its nature. As Patagonix does not make use of non-binding information, it is unable to validate the changes conducted by these mechanisms.

The common problem with all these approaches is that they consider kernel code to be static once it has been loaded into memory. Modern kernels, however, make use of many optimizations that require runtime patching, which renders these approaches obsolete. To validate kernel code it is thus essential to understand the individual runtime patching mechanisms that the kernel uses. Once these mechanisms are understood, we can implement code integrity validation mechanisms that reliably detect malicious modifications at *runtime*. In this paper, we make the first step in this direction and investigate the runtime patching mechanisms of modern Linux kernels.

3. DYNAMIC KERNEL PATCHING

In this section, we present constructs and mechanisms within modern OS kernels that make simple hashing techniques ineffectual. Our investigation is primarily based on the Linux kernel (Version 3.8, 64-bit), however some constructs also apply to the Windows kernel as well. We begin by taking an in-depth look at the load time patching mechanisms that the kernel uses and then discuss commonly used runtime patching mechanism in more detail.

Position Independent Code: The first mechanism that makes plain hashing difficult is position independent code. While the virtual start address for the kernel itself is fixed at compile time, the load address for each module is dynamic. Thus, the compiler can not predict the addresses of external symbols at compile time. Instead the necessary addresses are patched during the module's loading process. This is known as relocation. The relocation information is contained within each module binary. For each segment within the module a relocation table lists all references to both internal and external symbols.

Configuration-specific Patching: In addition to patching the modules for external symbols as described above, the Linux (module) loader may also replace code with architecture and configuration-specific opcodes or code blocks that are only present in certain configurations. This is done to improve performance, leverage special features such as *symmetric multiprocessing (SMP)* or (para-)virtualization on architectures that support them, or to provide extensibility. With these mechanisms the kernel can even replace entire functions. Such patching not only takes place in kernel modules, but in the kernel code as well.

Alternative Instructions (Load Time): For certain functionality, the opcodes used vary depending on the CPU's feature set. Generally, this approach is used when later CPU models support more efficient instructions. A list of alternative instructions is provided with each kernel binary, ordered by the most preferred as last in the list. The Linux module loader replaces each instruction if the requested feature is supported by the CPU. Notice that this is the only configuration-specific patching mechanism that is considered by Patagonix [3].

Hypercalls (Load Time): Another situation in which code is commonly patched within the kernel is related to vir-

tualization, in the form of hypercalls. Hypercalls are analogous to system calls in a virtualized environment which enable the guest system to interact with the hypervisor. Examples of such hypercalls include enabling and disabling interrupts, writing to specific processor registers, and MMU related functions.

As with relocation, hypercalls can not be inserted during compile time. Thus, the compiler provides hints in the compiled binary that a specific function (e. g., writing to the CR3 register, which holds the page tables for the currently executing process) is requested at a specific location in the kernel code. During load time the kernel is then able to decide how the requested function is provided and inserts the appropriate functionality. This can be done by inserting the corresponding opcodes directly, calling a function provided by the hypervisor, or even by jumping to a predefined location.

To facilitate this mechanism, the kernel maintains a table of alternatives introduced by a hypervisor-specific driver that can be used as a replacement for a given instruction. This feature is even employed in a non-virtualized or full-virtualized environment. In this case the kernel provides the native implementations for the requested functionality.

Symmetric Multiprocessing (Runtime): SMP describes an architecture with multiple CPUs that share memory, which is very common in modern PCs. There are portions of the kernel code that become critical sections (i. e., they share data that should only be accessed in a mutually exclusive fashion) when multiple CPU cores are available. In this case, the critical section must be protected with locks. However, the kernel only activates these locks if it is operating in an environment in which more than one CPU is present. This makes sense as the locking and unlocking operations are computationally expensive tasks and become unnecessary if there is only one CPU.

Furthermore, the Linux kernel supports enabling and disabling CPUs at runtime. This results in such patching also taking place at runtime. Note that adding and removing CPUs in a virtualized environment is frequently used for scalability. In addition to SMP locks it is conceptually also possible to patch arbitrary other instructions, in a fashion similar to alternative instructions, during runtime, once the number of active CPUs changes.

Jump Labels (Runtime): The Jump Labels mechanism is used within the kernel to optimize "highly unlikely" code branches to the point that their normal overhead is close to zero. Instead of checking whether a branch should be taken or not, the kernel either replaces the conditional jump with a no operation (NOP) instruction, and thereby omits the unlikely code, or with an unconditional jump to the unlikely code. Thus, the enabling/disabling of the function is an expensive task, but the runtime cost is completely avoided. Although it was mainly intended for debugging and tracing features, this mechanism is now frequently used both in the Linux scheduler and in the networking subsystem. For example, the latter uses it to activate and deactivate certain netfilter hooks.

Function Tracing (Runtime): Another mechanism that requires runtime code patching is the Ftrace function tracer. This tracer is used to debug the kernel or measure performance. It is commonly called at the beginning of each function within the kernel or its modules. For performance reasons, each tracer call is replaced by a NOP slide when the feature is currently disabled. Although the feature is similar to the Jump Labels mechanism, its implementation is different and it depends on other kernel data structures. Consequently, both mechanisms must be considered separately.

Summary. All of the runtime patching mechanisms mentioned above can be used at any time within the operation of the kernel. Consequently, the kernel's code pages are not static, meaning that simple hashing of code pages is not enough to validate its integrity. In addition, as the patching of the kernel's code depends on the current state of the running system, we must consider semantic information taken from within the guest OS to validate code integrity. To the best of our knowledge, we are the first to apply semantic knowledge for run-time verification of the kernel.

4. SYSTEM DESIGN

In the following we describe a new architecture for kernel code integrity validation that handles dynamic changes within kernel code. The overview of our architecture is shown in Figure 1. Our architecture makes use of virtual machine introspection (VMI) to provide both isolation and tamper resistance for our system. The three main components of systems are discussed more in depth in the following: (1) the *Preselector (PS)*, (2) the *Runtime Verifier (RV)*, and (3) the *Lazy Loader (LL)*.

4.1 Preselector (PS)

The task of the *PS* is to obtain the executable pages of the kernel, to divide code pages from data pages, and to associate code pages with a specific module or the kernel. To accomplish the first, the PS walks over the target system's page tables and extracts all supervisor pages that are present and executable. Since the virtual memory mechanism and the position and structure of page tables are specified by the hardware, this information is *binding*. That is, it reflects the true state of the system at the time of the validation.

The PS then assigns all pages, that were collected in the previous step, to their corresponding module or the kernel. Therefore, it extracts the list of currently loaded modules from the monitored system together with their code and data regions' virtual addresses. Based on the extracted data, it assigns the physical page frames to the modules by converting the obtained virtual addresses into physical addresses. The kernel binary is similarly processed. In this step the PS also separates code pages from executable data pages, by checking to which section the physical page belongs to.

In the next step, the PS performs an initial integrity check. It identifies pages that could not be mapped to either the kernel or a module and have the supervisor flag set. If such a page exists it is considered malicious. Thus, if a rootkit introduces code into the kernel and removes itself from the list of loaded modules, a common technique, it will be detected.

Finally, the PS processes executable kernel data pages. Although data pages should be marked as non-executable, this is not the case in practice. In the case of Linux, all allocated data segments and especially pages that are allocated with `kmalloc` were marked writable and executable. To solve this issue, the PS will mark all data pages as non-executable, a simple and effective solution. Since data pages should actually never be executed, this approach works well in our experiments.

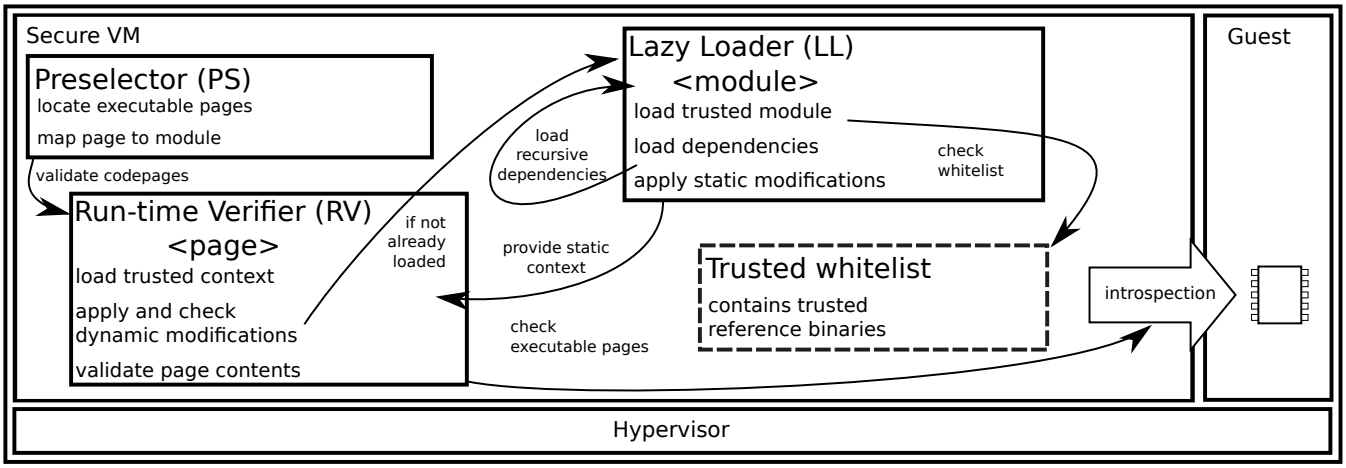

Figure 1: Architecture of the proposed code page validation framework.

4.2 Runtime Verifier (RV)

The *RV* is the heart of our system because it processes the code pages that it obtained from the PS. When it processes a page, it first checks whether the module the page belongs to has already been processed by the *LL*. If this is not the case it will invoke the LL, which is responsible for loading a *trusted* version of the respective module into memory.

Once a module has been processed by the LL, the RV will apply all predictable dynamic changes to the trusted reference binary and then compare each extracted kernel code page with the corresponding trusted reference. This comparison is conducted in a byte-by-byte manner. Once an inconsistency between the currently used code and the trusted version is detected, the RV validates whether this change is due to legitimate dynamic patching.

To check whether a change was conducted due to a dynamic patch, the RV makes use of a list of dynamic patch information that it obtains together with the module from the LL. This list is unique for each module (and the kernel) and contains information about each dynamic patch symbol within the binary. For each dynamic patch symbol, the list states the location of the symbol and the reason of the patch. Based on this information the RV can check whether a change that was conducted belongs to a dynamic patch symbol and if the currently applied change is valid for the given patch symbol. To make this scheme work, the RV additionally extracts all information from the monitored system that influences dynamic changes. This includes both information about the current architecture as well as information about the current system state. For example, a jump label can be enabled and disabled at runtime. Thus the RV must extract its current status from the monitored system before it can validate the code pages reliably. In this step the RV also checks the internal state of the running guest that is related to dynamic patching. This is both the kernel's information about the patching symbol as well as the current state. Only if the changes can be reconstructed and validated, the identity of the code page is validated. Notice that both information sources - the dynamic symbol list as well as the runtime information - are crucial, since an attacker is otherwise able to launch mimicry attacks, which we further discuss in Section 5.

4.3 Lazy Loader (LL)

In order to validate the integrity of a kernel code page in memory, the RV requires a trusted copy of that page from a trusted reference. To achieve this, the LL loads each module's (and the kernel's) binary from a trusted location, performs all load time modifications on it, and generates a list of dynamic patch symbols.

When the LL is invoked, it attempts to find the requested module using its name. Thus it requires access to a secure location containing all trusted kernel binaries. The contents of this location essentially functions as a whitelist and it is therefore crucial that its location is protected against attackers. We achieve this isolation through virtualization and store the trusted reference binaries outside of the monitored guest. In cases where the required binary is not contained within the secure location, the module is considered to be malicious. Otherwise the LL will load the binary into memory and will apply all predictable modifications, like relocation and loading of external symbols to it.

Finally, the LL must extract all runtime patchable locations for each module. It iterates over all patchable locations within the module and extracts all of the symbols that are relevant for runtime patching. In this process, it already applies all patches that are handled during load time. The list of the dynamic patch symbols as well as the binary representation is then returned to the RV.

5. IMPLEMENTATION

After giving a general overview over our proposed framework, we now discuss the implementation in more detail.

In the following we present our implementation of a framework to enable integrity validation of Linux kernel code pages on the x86 architecture. To handle dynamic patching, it is essential that our system is able to access the hardware and high-level software state of the guest system at runtime. While the former is easily possible from the hypervisor, accessing the high-level software state of the guest kernel from the VMM requires that we bridge the semantic gap [1]. To solve this problem, we navigate to the desired data structures by following pointers through the object graph starting at global variables. The location of the global variables is obtained from the trusted kernel reference binaries. The

layout of the corresponding structures is derived from the binaries *DWARF* debug information.

5.1 Identifying Executable Pages

The first component of our architecture is the PS. To be able to validate kernel code pages and to detect hidden code pages, our system initially requires a list of all executable kernel pages contained within the guest's memory. The PS obtains this list directly from the trustworthy underlying hardware, as it iterates through the page tables that are currently used by the system. The physical address of these page tables is contained within the CPU's `CR3` register.

Once the list of executable kernel pages is generated, the PS maps each page in the list to the kernel's or a module's code section. It extracts the addresses of important kernel structures such as the location of the `.text` or `.data` segment of every kernel module directly from the guest systems memory by reading the corresponding data structures. The addresses of the kernel's `.text` and `.data` section are extracted from its binary representation.

In practice there are two types of executable pages: dedicated code pages and executable data pages. In our test environment all of the kernel's data pages were actually mapped as executable, which is consistent with the fact that newly allocated pages also are mapped as executable per default. As previously mentioned we set all pages non executable.

5.2 Handling Load Time Patching

Similar to the kernel, we use a multi-staged process to reconstruct the contents of each executable page. First, all load time modifications are precomputed in the LL. In addition to the relocation and external symbol resolution, this phase also includes the patching of hypercalls and the processor dependent improvements. For a specific target system these steps are only reproduced once. The dynamic runtime patching mechanisms are considered in the second phase, which is conducted by the RV.

After the executable pages are assigned to their corresponding kernel component, the PS calls the RV to validate the contents of each page. The RV then invokes the LL for each module in order to obtain the validation context for that module. If the module's context is already initialized, the LL returns that context. Otherwise it initializes the trusted context as already described in the previous section. Note that this essentially replicates the kernel's loading process, as the LL also loads all of the dependencies of the requested modules.

After loading the trustworthy reference and all of its dependencies, our framework takes care of the relocation of internal symbols. Therefore, the binary representation of each module contains a list of locations and their corresponding symbols that need to be relocated. For each location and each internal symbol we calculate its virtual address in the memory of the inspected VM. We then replace each reference to a symbol with its absolute virtual address or a relative offset to this address depending on the type of relocation.

After the relocation we resolve external symbols. As with relocation, we replace all references to external symbols with the absolute address of the external symbol or a relative offset to its location. As to avoid relying on potentially compromised data-sources, our system doesn't rely on the monitored kernel's resources (e. g., its `System.map` or its internal list of exported symbols). Instead, we follow the kernel's dependency mechanism, by recursively loading all dependencies of the current module and initializing their full context for later usage. Thereby we also create our own list of (exported) symbols for each of the kernel binaries. When resolving an external symbol we consult our internal list of symbols.

Next, we process alternative instructions that are provided with each binary. As a reminder, this feature allows one to substitute specific instructions within the code with other instructions based on the current hardware. To decide whether the substitutions should be conducted, we obtain the necessary information from the virtual hardware. With a list of features at hand we now walk through the list of alternative instructions and substitute the referenced instructions, whenever the required feature is available.

Finally, the LL updates the instructions within the binary which depend on the host hypervisor. For this purpose, each binary contains a segment with a list of locations together with the type of patch that is to be applied. In contrast to the alternative instructions, the possible patch *values* for the locations are contained within a kernel data structure provided by the virtualization solution, not the kernel binary itself. Whether the patch that is applied is a simple instruction patch or a jump/call to another function is determined by the specific virtualization driver that is in use.

As we know exactly which hypervisor is being used, we are able to validate these patches through a whitelist. For this purpose our framework provides a plugin system to be easily extensible. This enables it to support different hypervisors (e. g., KVM, Xen) or additional patching mechanisms that may be introduced in the future as well. Since we, in this case, cannot trust the kernel's data structure, the whitelists for each hypervisor are generated from a trusted copy of this kernel data structure. In contrast to the original structure, the copy does not contain addresses, only the name of the symbol that is used. This is because the address of a function may vary due to relocation, while the symbol name is unique.

5.3 Handling Runtime Patching

After load time modifications are applied, the LL hands the initialized context to the RV. The task of the RV is to validate all dynamic modifications that cannot be predicted. Therefore, it analyses the current system state and also applies changes conducted by the runtime patching mechanisms. Afterwards it iterates byte-by-byte over all executable pages of the monitored system and compares them with the appropriate pages of the internal reference binaries. If a difference is detected, the RV will match the difference against one of the known runtime patching mechanisms.

The first mechanism checked is the SMP related patching. We obtain the number of currently active CPUs and adapt the locks within the trusted reference accordingly. Note that the number of CPUs is not necessarily static on a modern system. For example, in virtualization-based cloud environments it is common that vCPU cores are added and removed at runtime.

The next mechanism handled is the Ftrace function tracer. To validate an Ftrace function call, the RV first checks if the corresponding tracing functionality is enabled within the guest. Based on the state of the tracing mechanism, it then ensures that the Ftrace function call is either replaced with NOPs (tracing disabled) or that the call points to the `__fen-`

`try__` symbol (tracing enabled). The latter can be validated using the internal symbol list of our framework.

Jump Labels are another feature of modern Linux kernels that requires patching at runtime. To reiterate, with this feature a kernel developer is able to mask unlikely branches during normal execution. The mechanism provides a list of offsets inside the executable code together with a jump destination for each offset. At runtime the branch can be enabled or disabled by calling a specific function. The kernel therefore handles all jump targets within an internal data structure. Each entry in that structure contains a key, that indicates the current status of the jump label. To validate jump targets, our framework compares the value of this key with the current state of the jump target on the executable page. If the states match, the current target on the executable page is compared with the original target specified in the trusted reference binary. The change is only considered benign, if all information is consistent.

The final check that is performed by the RV is to ensure that pages that only partially contain code are valid. This situation arises when a code segment of a module or the kernel is smaller than an entire page of memory. In this case the code segment will only occupy a part of the page, while the remainder is unused. Since the page is executable, an attacker could try to inject code by modifying the unused code areas. In practice such an attack should be easily detectable as all unused kernel code regions are by default set to zero on Linux. Thus our framework protects against such attacks by ensuring that the unused space on the last page of a code segment does not contain any non-zero bytes after the end of the code. We will further discuss in Section 7 how some of this space is currently also dual mapped into userspace.

6. EVALUATION

After having introduced our framework for dynamic code integrity validation, we now present the evaluation of our framework. First, we evaluated the effectiveness of our approach in the case of kernel code integrity violations. For this purpose we evaluated the detection capabilities of our system using multiple rootkits. In a second set of experiments, we measured the performance impact introduced by our system. In the following we describe the experiments and their results in more detail. We conducted all tests on an AMD Phenom II X4 945 Processor with 16 GB of RAM. As monitored guest we chose Ubuntu 13.04 with 512 MB of RAM on the KVM hypervisor. The validation component of our framework as described in Section 5 was executed within the host OS.

6.1 Effectiveness

The primary goal of our proposed framework is to reliably detect kernel code integrity violations. To test our framework in this regard, we conducted multiple experiments with kernel rootkits. As one integral purpose of rootkits is to provide stealth, this enabled us to validate the effectiveness of our framework in a real-world scenario. The rootkits we tested with were four well-known kernel rootkits: adoreng, enyelkm.en.v1.1, intoxonia-ng2 and override. All make use of Direct Kernel Object Manipulation (DKOM) to hide themselves inside the victim kernel's memory.

With our experiments we verified that we were successfully able to detect all of these rootkits. In our tests we could distinguish three detection cases. In the case that the rootkit removed itself from the module list within monitored guest, our framework was unable to match its executable code pages to a kernel component. It marked the rootkits executable pages as malicious. When a rootkit did not hide itself from the list of loaded modules, our framework's LL didn't find a corresponding trusted representation of the module and thus marked the module as malicious. To further test our system we also renamed one of the rootkits to reflect a module contained in our whitelist. In this experiment the LL loaded the trusted version of the *decnet* module and the RV detected the mismatch.

Additionally, in all cases our framework detected all changes to the kernel code that were introduced by the rootkits. As our framework successfully identified all valid dynamic patches within the code pages, we had no false positives related to the kernel's dynamic patching mechanisms. Furthermore, the framework correctly informed us about the code pages that contained content that originated from userspace and automatically set all executable data pages to not executable.

6.2 Performance

To verify the applicability of our approach we also measured the performance imposed by our framework. As the kernel code is usually only patched infrequently, we provide a worse case evaluation scenario. We decided to monitor and validate the guest system live in a continuous manner which generates as much stress to the system as possible. At the same time, we executed a memory and I/O intensive task within the monitored VM. We compiled apache2 repeatedly.

In the first test, we only consider the case in which our framework is uninitialized and the LL component of our framework needs to load all trusted reference binaries. On average it took $4.051s$ to validate all kernel code pages without prior initialization. As this loading step is only required once, the case in which the framework is already initialized is a more accurate measurement of run-time overhead. In this case, the validation of all kernel code pages took on average only $0.279s$. During that time all 141 executable kernel code pages of our test environment where identified and validated. The check of a single page thus only took $0.002s$ ($2ms$), which is a negligible overhead.

6.3 Security Considerations

After evaluating the effectiveness and the performance of our system we now talk about the security of our approach. To validate the integrity of the kernel code pages our framework considers untrusted semantic information provided by the guest. Since this information is *non-binding* and could be tampered with, careful consideration has to be given to this serious security issue. In our case, we only derive auxiliary information from the guest system. That is, even if an attacker changes the information that our framework extracts, it does not break our approach.

In the following we will discuss the binding and non-binding information our framework uses. Our system first extracts information about all executable supervisor pages within the monitored system. As this information is directly obtained from the virtual hardware it reflects the true state of the system and is therefore trustworthy.

Furthermore, we also extract a list of all currently loaded kernel modules. We use this list to relate the pages in the

system to specific modules. If an attacker manipulates this list (e. g., by removing a module from it) our system will not relate the pages of the hidden module to a known module and considers them as malicious. Also, if the addresses of the text or data segment of a module is changed we detect this inconsistency. Thus using information contained from this data structure does not affect the security of our system, as any malicious alteration will be detected.

Another piece of information we process is the list of exported kernel and module symbols. The kernel's list of loaded symbols is a data structure that is commonly used for this purpose. As we reconstruct the kernel's loading process and thus regenerate this information within our framework, we do not depend on the kernel's possibly compromised data structure.

In the next step our framework generates the contents for SMP locks and alternative instructions that depend on the current state of the monitored system. This step is once more based on the virtual hardware, and thus can also be considered trustworthy. (Para-)virtualization related patching on the other hand is more difficult to validate. This is because the modifications conducted by the kernel directly depend on a kernel data structure provided by the corresponding hypervisor's driver. Our framework also validates the integrity of this data structure. As discussed before, we leverage a whitelist that contains a specific symbol name for each paravirtualization target. This is important, as in the case of paravirtualized Xen, some code locations are patched with an unconditional jump instruction and it is otherwise not possible to validate these jump targets.

To extract information as to whether a hook for the Ftrace or Jump Label mechanism is enabled, we again inspect the current state of the guest kernel. The current state of the jump/call is contained both within the kernel code and inside a kernel internal management data structure. To validate the integrity of the current system state we check if the state of the code is consistent with the kernels control data structure. If the current state is not reflected in the trusted data structure, we consider the change malicious. We furthermore verify that the control data structure is consistent with the information that is contained within the trusted reference binary.

Having outlined the security considerations in our system, we see that handling code changes due to the paravirtualization feature requires a whitelist. We added this functionality to increase the security of the guest kernel, though it is possible to validate the integrity without this whitelist. It is however, very important to understand that this mechanism can be leveraged to subvert the security of the kernel without directly affecting its integrity as this mechanism is intended to allow for arbitrary patching and hooking. The fact that this mechanism can be abused is a matter of the design of the mechanism itself. Virtualization therefore represents a perfect example of the edge cases that are not considered by existing systems. We introduce the ability to leverage whitelists in this situation to reduce the attack surface.

7. USER CODE IN THE LINUX KERNEL

To underline the importance of research in the field of kernel code integrity, we want to describe a particularly alarming behavior of the Linux kernel that we discovered during our research. On a high level, the issue we encountered allows an unprivileged user to load arbitrary code into kernel space. While the code is not executed by default, this makes exploitation in many cases trivial as the attacker only needs to find a way to point the instruction pointer into her code. This represents a critical issue, as there are dozens of security mechanisms such as secure boot, signed driver loading, $W \oplus X$, and Supervisor Mode Execution Protection (SMEP) that solely exist to hinder an attacker from loading code into kernel space or executing userland code from kernel space.

The root of the problem is related to performance optimizations, by which the x86 version of the Linux kernel uses 2MB pages to store its code segments. In case the last page of the kernel code is not completely occupied, the system has a hole at the end of the kernel code segment. This hole is effectively allocated to the kernel but remains unused. We found that instead of letting this memory go to waste, the kernel reuses this unused memory region and assigns it in the form of 4KB pages to userland processes, despite the fact that it is already mapped into the kernel code segment with both supervisor and executable privileges.

To show the concern of this architectural decision, we implemented an unprivileged userspace applicaton capable of inserting code into the kernel. In particular, we use the `pagemap` feature within the `/proc` file system to read the current mapping of virtual to physical pages. To ensure that our process has a page mapped into the kernel's text pages, we allocate multiple page-aligned memory areas, checking `/proc/self/maps` and `/proc/self/pagemap` each time to determine whether the most recent allocation is also mapped in the kernel's code segment. In practice, this was surprisingly effective. We found that it only takes a couple of tries until a page is mapped into the kernel.

Once we control a physical page that is also mapped inside the kernel text segment, we need to obtain its virtual address in order to be able to invoke it. As it turns out, calculating this address is as simple as: `<.text> + (pagenr * 0x1000)`.

This is the case as the first 8 MB of physical memory are mapped to that address in the so called *identity mapping* on the x86_64 architecture. If we assume a vulnerability inside the kernel which lets us redirect the control flow to a specific address, we are able to use that vulnerability to jump to the code we loaded into the kernel. Note that the vulnerability can be relatively simple as we only need to be able to control the instruction pointer. Traditionally, a vulnerability would require the ability to upload a payload, control the instruction pointer, and in some cases control the stack pointer in order to exploit it.

Kemerlis et al. [2] recently found a similar problem in the Linux kernel. They describe that the Linux kernel employs an identity mapping of the entire physical memory for performance reasons and that this area is not required to be executable by the kernel. This is a different mapping than the kernel identity mapping we described in this section. They then go on to then describe how the physical identity mapping can be exploited and propose a solution for mitigation. Their mitigation method unmaps a page from the physical identity mapping once it is allocated to userspace. After the page is no longer allocated in userspace, the page contents are wiped before it will be mapped to the identity mapping again. In contract, the vulnerability we discovered shows that a section of the kernel identity mapping is not occupied by kernel code and is instead allocated to unprivileged userspace applications. While the attack surface described by Kemerlis does overlap with our vulnerability,

the mitigation method does not apply. Thus, we propose that the physical memory, that is part of the kernel identity mapping, not be allocated to userspace applications.

8. CONCLUSION

Validation of kernel code integrity is an important aspect of runtime integrity checking. Previous approaches assumed the kernel code to be static at runtime or only depend on non-binding information while checking the integrity of kernel code. In this work, we have shown that modern kernels also employ dynamic runtime code modification. Due to complicated loading, debugging, and optimization processes, kernel code must be considered highly dynamic. To address this, we introduced and implemented a dynamic code validation framework using both binding and non-binding state information from the monitored guest. We reconstruct trusted copies of kernel code pages and differentiate between valid and invalid changes inside these pages. We also validate the integrity of the kernel's internal state related to the dynamic patching.

To show the viability of our system, we presented several experiments to test both the effectiveness and performance overhead of our system. We were able to show that our framework is able to detect all four of the rootkits we tested. Additionally, we were able to show that the performance overhead is as low as $2ms$ for every change that is made to the kernel code.

Finally, we discussed several security concerns we had as a result of our investigation. We identified a yet undiscovered a double mapping of executable supervisor code pages that are also mapped to userspace. This enables an unprivileged attacker to place arbitrary executable code within the kernel without violating protection mechanisms such as signed driver loading, $W \oplus X$ or SMEP. We discussed these issues and also successfully implemented a POC to exploit the issue.

9. REFERENCES

[1] P. M. Chen and B. D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001.

[2] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium*. USENIX Association, Aug. 2014.

[3] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th Usenix Security Symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[4] D. Oliveira, J. Navarro, N. Wetzel, and M. Bucci. Ianus: Secure and holistic coexistence with kernel extensions - a immune system-inspired approach. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1672–1679, New York, NY, USA, 2014. ACM.

[5] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM.

[6] N. L. J. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194. USENIX Association, 2004.

[7] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of 21th ACM SIGOPS symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[9] A. Seshadri, M. Luk, E. Shi, A. Perrig, and L. van Doorn andPradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM symposium on Operating Systems Principles*, pages 1–16, New York, NY, USA, 2005. ACM.

[10] A. Srivastava and J. T. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *NDSS*, 2011.

[11] J. Torrey. More: measurement of running executables. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 117–120. ACM, 2014.