Self-modifying Kernel Code Verification

Bo-shi WANG\(^1,2\) and Bi-bo TU\(^1,a,*\)

\(^1\)Institute of Information Engineering, CAS, China
\(^2\)University of Chinese Academy of Sciences, China

\(^a\)tubibo@iie.ac.cn

*Corresponding author

Keywords: Security, Kernel Security.

Abstract. As we know, injecting malicious codes is a simple and effective way to add attack logic with the same privileges as the injection targets. Because of OS kernel vulnerabilities, the kernels face code injection threats. Considering the importance of kernels, researchers proposed some kernel code protection solutions. But almost all of them depend on an assumption that the kernels don’t modify the codes of themselves. However, the self-modifying codes do exist widely in the kernels. It impedes the application of these solution to the real world. Moreover, the rest of solutions don’t provide specific technologies to deal with the problem. So we propose the self-modifying kernel code verification technology to distinguish malicious kernel code modification from valid kernel code self-modifications. Our technology promotes the suitability of existing kernel code protection solutions so that we enhance the kernel security in the real environment indirectly.

Introduction

Nowadays, computing platforms are used in more and more fields so generic OS kernels like Linux have to add more and more features. It increases the size and complexity of kernels. At the same time, the number of security vulnerabilities of OS kernels grows along with it. For the performance reason, most OS kernels are designed as monolithic kernels. Thus, regional vulnerabilities influence the whole kernel so that OS kernels face great security threats.

For specific attack goals, attackers need to complete a series of attack actions. However, it’s not easy to construct these actions. Codes are the determinant factor of program behaviors, so injecting a piece of codes into attack targets is an effective and sample solution.

There are some present works protecting kernel codes from modifications. The most famous solution is SecVisor [1], which implements a lightweight hypervisor to prevent malicious modifications for the guest kernel. SecVisor handles the code addition and reduction, which are brought by the LKM (Loadable Kernel Modules) feature. But SecVisor can’t deal with kernel code self-modifications. So SecVisor and many follow-up solutions [2,3] assume that kernels that they protect don’t contain self-modifying codes. The others [4,5] solutions which don’t base on that assumption don’t provide a specific or effective technology to solve the problem.

However, there are various kernel features, which modify kernel codes including LKM, alternatives, and static keys, Ftrace, Kprobes, KGDB and BPF (Berkeley Packet Filter). These features are widely used in the kernel so that OS administrators would not like to adopt these present kernel code integrity protection solutions, which aren’t...
compatible with self-modifying kernel code. It causes that these security solutions cannot be used in the real world and OSes is still under serious security threats like no solutions.

So we propose the self-modifying kernel code verification technology to check additions, reductions and modifications for kernel codes to distinguish malicious kernel code modifications from valid kernel code modifications.

In this paper, we make several contributions:

- We summarize all kernel features which add, delete or modify kernel code.
- We find differences between valid and malicious kernel code modification behaviors of numerous features.
- We propose how to implement the verification methods.

**Threat Model and Assumptions**

We assume that there is a hypervisor, which prevents the kernel from any kernel code modifications and provides support to LSM like SecVisor. Any kernel code modifications cause that CPU traps into the hypervisor and we can handle these modifications.

We consider the attacker who controls everything in the system but the CPU, the memory controller, and system memory chips. The attacker might have zero-day vulnerabilities in the kernel and application software. The attacker may attempt to use these vulnerabilities to locally or remotely exploit the system.

**Design**

Different features have different characteristics so we describe them and corresponding verification methods separately. Because various features cause a modification, only if a modification is regarded as an invalid modification by all of features, we assert the modification is malicious. In the other word, a modification passes any validations of features, we permit the modification.

**Alternatives**

More and more hardware instructions are being added into the processors. Moreover, the new instructions have better performance. Nevertheless, for compatibilities with old processors, the kernel, which runs in various hardware platforms, does not use the new instructions. To reconcile the performance and the compatibility, the kernel automatically applies different instructions at runtime according to the processors feature supports. For the reason, the alternatives are proposed. The alternatives allow that specific instructions replace another specific instruction in a specific address at runtime. It shows that alternatives are useful. Moreover, alternatives do not depend on any kernel configurations so we cannot disable it. Thus, it is necessary for us to support alternatives.

The alternatives requires that kernel developers provide an old instruction to be replaced, one or two new instructions and processor features which new instructions depend on for each alternatives entry. In fact, some common codes help each alternatives entry to record some extra information for instruction replacements. Table 1 shows all fields of the alternatives entry. According to these fields, the kernel chooses the appropriate instructions. Moreover, we know which modifications are valid.
Table 1. Fields of the alternatives entry.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_offset</td>
<td>s32</td>
<td>offset of the old instruction from the current field</td>
</tr>
<tr>
<td>repl_offset</td>
<td>s32</td>
<td>offset of the new instruction from the current field</td>
</tr>
<tr>
<td>cpuid</td>
<td>u16</td>
<td>feature bit set for the new instruction</td>
</tr>
<tr>
<td>instlen</td>
<td>u8</td>
<td>old instruction length</td>
</tr>
<tr>
<td>replacementlen</td>
<td>u8</td>
<td>new instruction length</td>
</tr>
<tr>
<td>padlen</td>
<td>u8</td>
<td>length of build-time padding NOP instruction</td>
</tr>
</tbody>
</table>

We calculate all addresses of old instructions according to *inst_offset* field and store them in a hash table in advance. Once we find that the address of a modification is related to an alternative entry, in the other word, the address is in the hash table, we check whether the new instruction is the same as the replacement instruction indicated by the entry. If not, we think that the modification is not valid for this kernel feature.

**Static Keys**

There are some performance-sensitive fast-paths in the kernel. Some of them use rarely-used features. In generally, the kernel needs to check whether the feature is enabled. The checks often cause branch prediction failures in processors. It brings performance overhead and influences the fast-paths too much. Static keys switches JUMP instructions and NOP instructions depending on a key controlling whether a feature is enabled to implement branch semantics without performance overhead. Static Keys is useful so we decide to support it.

Static keys requires that the condition sentences based on the feature use a specific function to decide. The specific function only contains a NOP instruction and two return sentences. The first invariably returns false and the second invariably returns true. So the function returns false by default. Once the key is enabled, the NOP instruction is replaced by a JUMP instruction and the jump target is the second return sentence so that the function return true. Considering that instruction switches are not atomic, the kernel replaces the first byte of instructions by an INT3 instruction in the first step and then replaces the rest of the instruction. Finally, the kernel replaces the INT3 instruction with the first byte of the target instruction. So we regard the NOP instruction, the JUMP instruction with a valid target and each state of replacement process as valid states. All switches depend on jump labels. Table 2 shows all fields of a jump label where the jump information is stored.

Table 2. Fields of the jump label entry.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>jump_lable_t</td>
<td>address of the instruction</td>
</tr>
<tr>
<td>target</td>
<td>jump_lable_t</td>
<td>address of the jump target (the sentence which return true)</td>
</tr>
<tr>
<td>key</td>
<td>jump_lable_t</td>
<td>address of the related key</td>
</tr>
</tbody>
</table>

Like alternatives, we store all instruction addresses related to static keys in a hash table. The process of modifications is complicated and there are many intermediate states. But we summarize some principles to simplify the verification logic. Once we
detect any modifications and the address of the modification is in the hash table, we check whether the result of the instruction execution satisfies following conditions:

I. The new code bytes is a complete NOP instruction.
II. The new code bytes is a complete JUMP instruction and the jump target matches the target field of the jump label associated with the JUMP instruction address.
III. The new code is leaded by an INT3 instruction and the following bytes are made up of a subsequence of the NOP instruction in the condition I and a subsequence of the JUMP instruction in the condition II with the same addresses.

If one of three conditions is satisfied, we think the modification is valid.

KGDB

KGDB is a kernel debugger. When developers debug a kernel or kernel modules, they can use another computer with a GDB-like debugger frontend to connect the debugged computer by serial ports or network remotely. The debugger frontend communicates with KGDB in the debugged kernel by the remote GDB protocol, and KGDB does actual works. The processors provide hardware debug mechanisms, but because of hardware resource restrictions, the number of breakpoints is limited. Therefore, the software mechanism is necessary. The software mechanism requires that the instructions at breakpoints be replaced with INT3 instructions for breaking kernel execution. KGDB modifies the kernel codes only if the KGDB set or clear break points.

Because the breakpoints can be set anywhere, we cannot check the modifications by the addresses. We regard all modifications with INT3 instructions as valid actions. When we detect that a byte is replaced with the INT3 instruction, we record the address and value of that byte. If an INT3 instruction is replaced, we search the address of the replaced INT3 instruction in the recorded addresses. If we find, we check whether the new value is the same as the recorded value. If values match, we consider the modification is valid. Otherwise, we consider the modification is malicious.

Ftrace

Ftrace is an internal tracer designed to help out developers and designers of systems to find what is going on inside the kernel. It can be used for debugging or analyzing latencies and performance issues that take place outside of user-space [6]. What’s more, Ftrace is used to live patch the kernel. Thus, Ftrace is an important feature so we decide to support it.

Compilers add an extra CALL instruction at the entry point of functions, which can be traced. At boot time, these instructions are replaced with NOP instructions. When a trace point is enabled, the NOP instruction at the entry point of the related function is replaced with a CALL instruction. When a trace point is disabled, the CALL instruction is replaced with a NOP instruction. Like Static Keys, the processes of instruction modifications are intervened by INT3 instructions. What’s more, Ftrace has another trait that the called functions are some specific functions or trampoline functions. Specific templates generate the trampoline function. The unique difference between trampoline functions and their template is the jump target. And the jump target of the trampoline function is one of specific functions.

Because there are symbol tables in the kernel, we do a binary search on the sorted symbol tables. If we find that the address of modified codes is one of the entry point of function, we do checks based on three conditions like what we do dealing with Static Keys. The differences are that Ftrace uses CALL instructions instead of JUMP
instructions and that the called targets are not specified by some data structures but specified by a prepared function list and a discriminant function deciding whether a target is a trampoline. If the called target is a trampoline, we also check whether the jump target is in the jump target function list. Only all checks are passed, we think the modification is produced by Ftrace.

**Kprobes**

Kprobes enables developers and administrators to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. They can trap at almost any kernel code address, specifying a handler routine to be invoked when the breakpoint is hit [7]. Kprobes is useful so we support Kprobes.

In generally, Kprobes only uses INT3 instructions like KGDB. So we do checks like what we do in KGDB. What’s more, Kprobes has an optimizer, which replaces INT3 instructions with JUMP instructions. The jump targets are also some trampolines, so we do checks like what we do in Ftrace.

**BPF and LKM**

Because BPF [8] brings in user space codes, we cannot judge a modification is malicious or made by BPF. What’s more, BPF is a packet filter, so its usage scenario is restricted. Thus, we do not support BPF. SecVisor supports for LKM [9] so we do not need to do anything more.

**Evaluation**

We test our SKCV (Self-modifying Kernel Code Verification) method on a computer with Intel Core i5-4590T 2.00GHz CPU, 12GB memory and Ubuntu 16.04 OS. We run SPEC2000 with 10 times. Fig. 1 shows the result of the performance test. By analyzing the result, we find that the hypervisor with our technology is as fast as the native hypervisor. Because there are few code modifications in the kernel, the frequency of verifications is low and the performance overhead is little in result.

![Figure 1. Performance comparison between the native hypervisor and the hypervisor with self-modifying kernel code verification technology.](image)

**Conclusion**

Our self-modifying kernel code verification technology is based on current kernel code protection solutions and provides verifications for kernel code modifications from
kernel features. Our technology improves the usability of kernel code protection solutions and enhances the OS security indirectly. Because of few kernel code self-modifications, our technology does not introduce obvious performance overhead.

References


