HACS: A Hypervisor-Based Access Control Strategy to Protect Security-Critical Kernel Data

Jian-ping WANG\textsuperscript{1,\textsuperscript{a}*}, Pei ZHAO\textsuperscript{1} and Heng-tai MA\textsuperscript{1}

\textsuperscript{1}Institute of Software, Chinese Academy of Sciences, 4\# South Fourth Street, Zhong Guan Cun, Beijing 100190, P.R. China

\textsuperscript{a}jianpingzju@foxmail.com

*Corresponding author

Keywords: VMM, Rootkit, Security, Kernel Data.

Abstract. Rootkits are prevalent in today’s Internet. Using virtual machine monitor (VMM) is an attractive way to deal with rootkits. However, most of the previous studies do not focus on protecting kernel data using VMMs, especially for the data that may be dynamically changed. Direct kernel object manipulation (DKOM) attacks can stealthily detach kernel data objects belonging to the malicious program from kernel’s normal list, or overwrite import fields in the kernel. It’s hard for OSes or VMMs to distinguish between normal accesses and malicious ones to kernel data. Although some works provides access control policy to kernel data, it can’t detect loadable kernel module (LKM) rootkits that place malicious code to their installation procedure. This paper presents a framework, namely HACS, to protect kernel data using module white list based access control. HACS runs in VMM and intercepts the write requests to protected regions. A mediation strategy is proposed in this paper that judges whether the modification requests are legitimate or not. Malicious modifications requests are forbidden and normal ones are approved. Our experiments with real-world DKOM rootkits demonstrate that HACS provides robust rootkits detection ability, with lower performance overhead than other kernel data protection methods.

Introduction

Rootkits using DKOM attacks can hide malicious processes or modules by removing them from kernel’s normal process list or module list. In other attacks, they may change the address of interrupt descriptor table (IDT) or system call table to the carefully constructed functions to implement malicious proposes, or they may escalate process’s privileges by modifying the process’s user credentials with those of root users. However, most of previous studies can’t solve the problem perfectly. Virtual machine introspection (VMI) (VMWatcher [2]) can detect hidden rootkits, but it relies on a benign core kernel, and will fail to identify the presence of malicious software that has already detached import data objects from guest OS’ kernel or overwritten the import fields in the kernel. There are some studies that detect the DKOM attacks by protecting related kernel data, but they can’t deal with all kinds of attack scenarios. For examples, Sentry proposed in the paper [1] protected kernel data via memory access control, but it can’t detect rootkits that place malicious code to rootkit’s installation procedure.

In this paper, we propose HACS, a framework to protect kernel data by judging which code regions or modules are modifying the protected regions. HACS makes the contribution that it protects kernel data with a white list based access control strategy. With this strategy, HACS has the ability to detect rootkits with DKOM and Hijack system calls attacks. Especially, it can detect the rootkits that modify security-critical
kernel data during the rootkits are being installed. In Linux, rootkits are mostly installed as Linux LKMs. Some rootkits place malicious codes in module’s initialization function, these codes will be loaded to .init.text section of kernel code to execute. Normal kernel modules will also modify kernel data at their installation time. HACS’s strategy locates the module that launched the modification request, then denies or approves the request according to the module white list. The module white list contains credible kernel modules derived from experience.

Related Works

Several studies have proposed different solutions to protect kernel data’s integrity. Gibraltar [3] protected kernel data by hypothesizing invariants on kernel data structures based on tracing the execution of core kernel. The paper [4] developed a system that detects semantic integrity violations of kernel objects. It is possible for structures like a process, because process accounting list can be compared with process scheduling list to find the missing process-struct. But most kernel data struct don’t contain multiple views, such as kernel module lists. Sentry [1] divides the kernel struct into the protected and unprotected region. All attempted data alterations are mediated using memory access policy, only those invoked by legitimate kernel functionality are allowed. But Sentry has the limitation that it assumes code region from .init.text section are legitimate to modify protected data. But there are LKM rootkits that place malicious codes to .init.text section to execute. Instead of trusting all .init.text section, HACS maintains an experimental module white list, only modules in the white list are legal to modify the protected region. When LKM rootkits are modifying protected kernel data, HACS can locate the module that invokes the modification request, then denies or approves the modification request according to the module white list.

Some researchers have focus on protecting read-only kernel data. PatchGuard [5] protects read-only structures containing code pointers, such as the system service descriptor table (SSDT) and IDT. UCONKI [6] is a framework that controls access to read-only kernel data structures. These studies have the limitation that they can only protect the kernel data regions that can not be changed. However, lots of rootkits, such as these with DKOM technology can modify the kernel data that’s allocated dynamically from the heap, these areas can be modified legitimately by kernel code, it is not a good solution by simply making the protected area read-only. However, HACS can not only protect static kernel data but also dynamically-allocated kernel data by taking strategies to judge whether the alteration requests are legitimate.

Architecture of HACS

HACS is designed as a framework to protected kernel data using module white list based access control. HACS runs inside the hypervisor that intercepts write requests to protected regions. HACS consists of several modules: strategy, protected list, module white list and an exception handler. Strategy module is the core of HACS’s protections. There are write requests to protected data, the strategy module can determine whether the requests are legitimate or not. Protected list contains the addresses of kernel variables or objects which need to be protected. Module white list contains reliable modules that are allowed to alter the protected data in the protected list. When there exists a write request to the regions in protected list, and the request is judged as legitimate by strategy module, exception handler module will make the request can
continue on normally. If the request is judged as illegal, exception handler module will deny the request, emit a warning and turn the control over to guest OS.

HACS’s strategy module determines which code regions are allowed to alter security-critical kernel data, they are called credible code regions in HACS. Any modification request will be denied which does not fall into the credible code regions. The credible code regions are defined as followed:

1. All core kernel code that doesn’t belong to loadable modules or drivers has the rights to manage its own kernel data regions. Take Linux kernel for example, the core kernel code ranges from the symbol _text to _etext.
2. In Linux kernel, the code segment from symbol __init_begin to __init_end that contains the code for booting the kernel is trusted.
3. In Linux kernel, if the code segment is from symbol __initcall7_start to __initcall8_start that contains the code for initializing LKMs, and the code segment is from a trusted module that’s in the module white list, it’s allowed to modify the protected data. An algorithm called FunctionPath is proposed in HACS to find which module is launching the modification request to protected data.

The protected list consists of a list of \{start_address, end_address\} elements, where start_address and end_address give the ranges of protected region. The start_address and end_address should be the format of guest virtual address. The module white list is a list of kernel modules that are allowed to modify protected kernel data regions, which consists of a list of \{module_name\} elements. If there exists a LKM that has launched a modification request, FunctionPath algorithm proposed in HACS will identify the module’s name. The modification request will be approved or rejected by judging whether the module’s name is in the module white list. For example, Adore-ng rootkit is a LKM rootkit that removes itself from the module list by changing prev and next pointers of module-struct during Adore-ng’s initialization. FunctionPath algorithm will find it is the Adore-ng module altering the protected region when Adore-ng is being installed. Since Adore-ng is not in the module white list, it is prohibited. The module white list is constructed by users’ experience. For example, e1000 is Intel’s network card driver module, and can be put into the module white list.

```
switch (exit_reason) {
  2  case (ept_violation):
  3    if (!address_mapped()) {
  4      map_address();
  5    }
  6    if (write_legal()) {
  7      set_pte(pfn, writable|readable);
  8      start_single_step();
  9    } else {
 10      emit_warning();
 11    module_name = FunctionPath();
 12    remove_module(module_name);
 13  }
 14  case (single_step_exception):
 15    if (caused_by_HACS()) {
 16      set_pte(pfn, readable|unwritable);
 17      stop_single_step();
 18    } else
 19      forward_control_to_guest();
 20}
```

Figure 1. Procedure of the exception handler.

EPT mechanism protects memory at the granularity of pages, HACS calculates which pages locate within these address ranges in the protected list, thus getting a list of page frame numbers (PFNs). These protected pages are set read-only to prohibit illegal modifications, the exception handler is used to make the legitimate modifications to proceed normally. Figure 1 shows the procedure of the exception handler. Exit_reason is the exit code for the VM Exit event. Ept_violation happens under the following two conditions: firstly, there exists a memory access to GPA without mapping to a HPA in the EPT paging structs; secondly, there exist unauthorized accesses to memory pages
according to the pages’ permission bits in page table entry (PTE) fields. In the procedure of dealing with eptViolation, the mapping from GPA to HPA must exists firstly (Line 3 ~ 5), then checks whether the write request is legitimate with the function write_legal (Line 6), which is determined by HACS’s strategy. If the write request is legal, the processor’s single-step interruption mechanism is taken advantage to make the write request to continue on correctly. HACS makes the legal request to re-access the read-only pages by setting the desired protected pages writable (Line 7), then CPU is set to single-step mode (Line 8). If single-step exception happens, and it’s triggered by HACS, the permission of the accessed pages is reset to read-only, then CPU’s single-step mode is closed (Line 14 ~ 18). If the single-step is launched by other programs, like debugger, this exception is reflected to guest applications and the control is forwarded to guest OS (Line 19). If the write request is illegal, a warning is emitted to the guest OS (Line 10). FunctionPath function helps to find the module that makes the modification request (Line 11), then the discovered malicious module is uninstalled via remove_module function (Line 12).

**Implementation**

HACS is implemented on BitVisor [7] with version 1.4, which is a thin hypervisor that enforces I/O device security. Thin hypervisor is used for the reason that it can guarantee security reliability and considerable performance overhead. This session will introduce the implementation of HACS. Firstly, it’s introduced what kernel data needs to be protected. Secondly, FunctionPath algorithm is detailed.

What kernel data to protect is determined by how the various rootkits attack. Some rootkits detach process struct or module struct from the process accounting list or module list to hide them by changing the prev and next pointers of the detached struct’s previous and next nodes. If the prev and next pointers of a node belongs to the process accounting list and module list, they will be added to HACS’s protected list. Some rootkits may overwrite the contents of interrupt descriptor table (IDT) or system call table to change the normal function calls’ addresses to malicious ones. The IDT and system call table are added to HACS’s protected list. Users can add more security-critical kernel data to the protected list.

```
1 string FunctionPath() {
2   address_t func_argu1, func_argu2, module_init_entry;
3   string module_name;
4   while(!callstack.end()&&func_argu2!=symbol(do_one_initcall)){
5     func_argu1 = *(ebp+8);
6     func_argu2 = *(ebp+12);
7     ebp = *ebp;
8   }
9   module_init_entry = func_argu1;
10  module_name = mod_name_entry_map.find(module_init_entry);
11  return module_name;
12}
```

Figure 2. Procedure of FunctionPath algorithm.

If it is a LKM that launches write request to protected regions, FunctionPath will find the module’s name. FunctionPath traverses the call stack from the function that invokes the modification to the kernel’s function that initializes the module. In order to perform FunctionPath, guest OS’s kernel needs to be compiled with a configuration that maintains a stack frame base pointer (ebp register) during the guest kernel runs.
Guest OS’s kernel will be paused in order to keep a consistency state of stack frame pages when performing FunctionPath. Figure 2 shows the procedure of FunctionPath algorithm. Firstly, we will detail how to initialize LKM in Linux kernel. do_one_initcall function is adopted which takes the address of the modules’s initialization function as its argument and calls the modules’s initialization function. The call stack is traversed from the function that launches the modification request to do_one_initcall function. To determine whether the do_one_initcall function is reached, the do_one_initcall function’s address is added to the argument list of do_one_initcall function as its second argument. When there is a call request to FunctionPath, the algorithm finds the location of current stack frame from ebp register. The address of current function’s first argument is gotten by adding 8 bytes to the value of ebp register (Line 5). The second argument’s address is gotten by adding 12 bytes to ebp register (Line 6). The previous stack frame’s address is stored at the location pointed by ebp register’s content (Line 7). This process (Line 4 ~ 8) is repeated until it reaches to the do_one_initcall function. The do_one_initcall function’s address can be found in file /boot/System.map as symbol(do_one_initcall). The address of the module’s initialization function is stored in the module_init_entry variable (Line 9). mod_name_entry_map is a map struct that records the mapping from the address of module’s initialization function to the module’s name. Both of the module’s name and address of module’s initialization function are stored in the module-struct. When kernel initializes the module, the module’s name and address of module’s initialization function are put into the mod_name_entry_map struct. The module’s name is fetched from mod_name_entry_map struct with module_init_entry variable (Line 10).

**Evaluation**

In this Section, we evaluate HACS with real-world rootkits to demonstrate its attack detection capability, then detail its performance overhead. The evaluation is measured on an Intel Core i7-4790 3.60 GHz system with 8 cores and 8GB RAM. The guest OS is Ubuntu 14.04 with a version 2.6.39 kernel. BitVisor 1.4 is used as the hypervisor.

We tested HACS against a collection of real-world rootkits. Table 1 shows the results. “Hide process” column shows whether the rootkits can hide processes and the technology they take. “Hide module/itself” column shows whether the rootkits can hide modules and the technology they take. It’s exciting to see from the results that HACS can detect DKOM rootkits that detection methods based on VMI (VMWatcher [2]) can’t detect. For LKM rootkits (Adore-ng) that place malicious code (remove itself from module list) to their installation procedure, HACS can detect the hidden rootkit module, while Sentry [1] can’t. Rootkits with DKOM attacks modify process/module node’s prev/next pointers to hide processes and modules. Since process/module node’s prev/next pointers are in the protected list, malicious modifications will be detected with HACS’s white list based access control strategy and forbidden with HACS’s exception handler. Rootkits with Hijack system calls (HSC) attacks modify system call table to change the normal function calls’ addresses to malicious ones. Since the system call table is in HACS’s protected list, modifications from rootkits to system call table will be detected and denied by HACS.
Table 1. Real-world rootkits evaluated with HACS.

<table>
<thead>
<tr>
<th>Rootkit</th>
<th>Hide process</th>
<th>Hide module/itself</th>
<th>Target OS</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adore-ng</td>
<td>YES(HSC)</td>
<td>YES(DKOM/HSC)</td>
<td>Linux&gt;2.6</td>
<td>Detected</td>
</tr>
<tr>
<td>Iyyl’s(modified) [8]</td>
<td>YES(DKOM)</td>
<td>YES(DKOM)</td>
<td>Linux&gt;2.6</td>
<td>Detected</td>
</tr>
<tr>
<td>PhalanX</td>
<td>YES(DKOM/HSC)</td>
<td>Not support</td>
<td>Linux 2.6</td>
<td>Detected</td>
</tr>
<tr>
<td>SucKIT</td>
<td>YES(DKOM/HSC)</td>
<td>Not support</td>
<td>Linux 2.6</td>
<td>Detected</td>
</tr>
<tr>
<td>Enyelkm</td>
<td>YES(HSC)</td>
<td>YES(HSC)</td>
<td>Linux 2.6</td>
<td>Detected</td>
</tr>
</tbody>
</table>

The system actions related to the system calls, processes and modules are tested to demonstrate the performance overhead. The results are shown in Table 2, “Normal OS” column means the case of same OS without hypervisor. Firstly, we measure the system call time with Lmbench, a Linux benchmark tool. The results indicate that HACS only incurs 0.83% ~ 2.14% overhead. Secondly, we measure the process-related time with Lmbench. The results indicate that HACS incurs 42.57% ~ 44.41% overhead. Finally, we measure the time taken by module load (insmod) and unload (rmmod) operations. A sample module that traverses the list of loaded modules is used. It can be seen from the results that HACS incurs 66.26% ~ 71.42% performance overhead.

Table 2. Performance overhead evaluated with HACS.

<table>
<thead>
<tr>
<th>Actions</th>
<th>Normal OS[ms]</th>
<th>HACS[ms]</th>
<th>Overhead[ms]</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple syscall</td>
<td>0.2931</td>
<td>0.2989</td>
<td>0.0058</td>
<td>1.98%</td>
</tr>
<tr>
<td>Simple read</td>
<td>0.3358</td>
<td>0.3386</td>
<td>0.0028</td>
<td>0.83%</td>
</tr>
<tr>
<td>Simple write</td>
<td>0.3497</td>
<td>0.3572</td>
<td>0.0075</td>
<td>2.14%</td>
</tr>
<tr>
<td>Process fork+exit</td>
<td>63.1395</td>
<td>90.0167</td>
<td>26.8772</td>
<td>42.57%</td>
</tr>
<tr>
<td>Process fork+execve</td>
<td>67.2840</td>
<td>97.1373</td>
<td>29.8533</td>
<td>44.37%</td>
</tr>
<tr>
<td>Process fork+/bin/sh -c</td>
<td>475.5455</td>
<td>686.7500</td>
<td>211.2045</td>
<td>44.41%</td>
</tr>
<tr>
<td>insmod</td>
<td>0.07945</td>
<td>0.13619</td>
<td>0.05674</td>
<td>71.42%</td>
</tr>
<tr>
<td>rmmod</td>
<td>0.08922</td>
<td>0.14834</td>
<td>0.05912</td>
<td>66.26%</td>
</tr>
</tbody>
</table>

Summary

In this paper, we presented HACS, a hypervisor-based scheme for protecting kernel data. The results demonstrated that HACS can detect rootkits with DKOM and HSC attacks. For the rootkits that place malicious codes to their installation procedure, like Adore-ng that detaches itself from module list to hide itself, HACS uses the white list based access control strategy with FunctionPath algorithm to detect the malicious codes, while traditional kernel data protection methods can’t. HACS provided satisfied performance overhead with the assist of thin hypervisor. HACS assumes that the core kernel code and the hypervisor are trusted. If rootkits inject malicious codes into the core kernel code or the hypervisor, HACS will fail to detect the rootkits. In the future, we will do more research on the security of the core kernel code and the hypervisor.

Acknowledgement

This research was financially supported by National Natural Science Foundation of China (Grant No. 61402451).

References


[8] https://github.com/geekben/rootkit