IEICE TRANS. INF. & SYST., VOL.E97–D, NO.6 JUNE 2014

LETTER

A Virtualization-Based Approach for Application Whitelisting*

Donghai TIAN[†,††a)], **Jingfeng XUE**[†], **Changzhen HU**[†], ***Nonmembers*, *and*** **Xuanya LI**[†††], ***Student Member***

SUMMARY A whitelisting approach is a promising solution to prevent unwanted processes (e.g., malware) getting executed. However, previous solutions suffer from limitations in that: 1) Most methods place the whitelist information in the kernel space, which could be tempered by attackers; 2) Most methods cannot prevent the execution of kernel processes. In this paper, we present VAW, a novel application whitelisting system by using the virtualization technology. Our system is able to block the execution of unauthorized user and kernel processes. Compared with the previous solutions, our approach can achieve stronger security guarantees. The experiments show that VAW can deny the execution of unwanted processes effectively with a little performance overhead.

key words: *whitelisting, virtualization technology*

1. Introduction

As more and more software vulnerabilities get discovered, software security problems become very important. By exploiting these vulnerabilities, attackers could hijack the programs' execution and then intrude into the end hosts. Furthermore, malicious processes could be planted into the compromised hosts. Relying on the planted processes, attackers can remotely control the end hosts for launching several attacks (e.g., data collection and distributed denial of service).

To prevent the malicious processes from getting executed, system administrators may deploy the Antivirus tools into the end hosts. Before running a executable file, the Antivirus will check whether the file contains malicious code. If it is, the Antivirus will deny the execution. In practice, the Antivirus tools really raise the bar for executing known malicious code. However, since the tools usually depend on the extensive signature database, they are limited to detecting new malicious code [1]

To address the problem of the Antivirus, many researchers propose whitelisting approaches [2]–[4]. The basic idea of these approaches is to maintain a whitelist of programs, which include executable files and libraries. Only the programs included in the whitelist can get executed. In this way, unwanted programs would have no chance to run. However, existing whitelist approaches store the important whitelist data in the kernel space, which could be manipulated by attackers to bypass the execution enforcement. Moreover, the existing methods mainly focus on user-level protection, and they cannot block the execution of an unwanted kernel process.

In this paper, we present a virtualization-based application whitelisting system, called VAW, which can block the execution of unauthorized user and kernel processes. VAW is implemented on the Xen hypervisor, and it only requires minimal changes to the existing programs and OS kernel. Our approach makes the following contributions:

- We propose a novel application whitelisting method based on the virtualization technology.
- We leverage the Secure-In-VM (SIM) framework [6] to protect the whitelist data located in the kernel space.
- We design and implement a prototype of VAW based on Xen. The evaluations show that our system can prevent an unwanted user or kernel process from being executed.

2. Overview of Our Approach

The goal of VAW is to build a system that can prevent an unauthorized process from being executed. Different from previous whitelisting systems, our approach exploits the process scheduler in the OS kernel to enforce the process execution. Specifically, whenever a process needs to be scheduled, the scheduler will judge whether the process is included in the pre-defined whitelist. If it is, the scheduler will carry out process switching. Otherwise, the scheduler will choose another process that is located in the whitelist to run.

To register a process to the whitelist, we employ the trusted communication channel between applications and the underlying hypervisor. Considering the efficiency of process scheduling, we maintain the whitelist data in the kernel space such that the scheduler can access the data directly. In order to protect the whitelist from being tempered by attackers, we leverage the SIM framework to isolate the scheduler and whitelist from the OS kernel. In this way,

Manuscript received December 24, 2013.

[†]The authors are with the School of Software, Beijing Institute of Technology, China.

[††]The author is also with State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China.

[†††]The author is with National Engineering Laboratory for Information Security Technologies, Institute of Information Engineering, Chinese Academy of Sciences, China.

*This work is supported partially by the National High-Tech Research Development Program of China under Grant No. 2009AA01Z433 and the Open Foundation of State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences) under Grant No. 2013-4-1.

a) E-mail: donghaitad@gmail.com

DOI: 10.1587/transinf.E97.D.1648

the other kernel components cannot access the whitelist directly. Furthermore, the interactions between the process scheduler and other kernel components are well controlled so that even kernel attacks cannot bypass our protection to hijack the scheduler's execution.

3. System Design and Implementation

We have developed VAW, a prototype based on Xen to demonstrate our approach. As Fig. 1 shows, the major components are located in the guest kernel and hypervisor. The Authenticator is responsible for authenticating whether the target process or kernel module can fork a new process. The Info stores the authenticated information. The role of the scheduler component is to select a feasible process from the whitelist to run.

3.1 Process Registration

Regarding process registration to the whitelist, we leverage the trusted communication channel, by which the user process can transfer information to the hypervisor securely. In general, the work flow of our process registration is illustrated in Fig. 2. First, the target process invokes a hypercall with a secret key to notify the hypervisor that it can fork a process. Then, the hypervisor will check whether the secret key is valid. If it is, the hypervisor will mark the target process with a flag to indicate that it can create a new process. After that, the execution is transferred back to the user space. Next, the target process invokes a system call (i.e., `fork()`) to create a new process, which causes the execution trapped into the OS kernel. Once the process descriptor is fully created, the OS kernel will issue a hypercall for process registration. If the parent process is not marked before, the new child process cannot get registered.

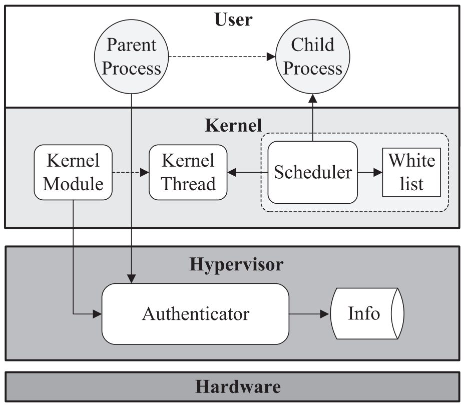

Fig. 1 The VAW architecture.

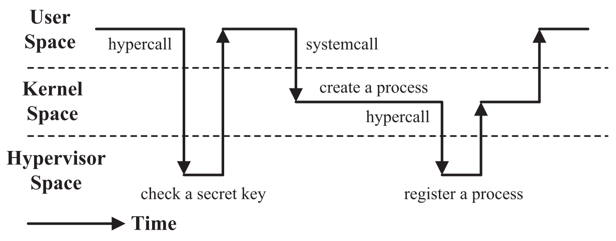

Fig. 2 Process registration procedure.

To select a process identifier as the registered data for process registration, we cannot rely on the internal field (e.g., pid) of a process descriptor in that kernel-level attackers could temper with the internal field easily. Instead, we utilize the memory area of a process descriptor as the registered data for process registration.

Furthermore, to detect loading unauthorized executable into the registered child address space, we apply the previous approach [5] to intercept the `mmap` and `execve` system calls for authorization. If the loaded executable is unauthorized, the registered process will be removed from the whitelist.

On the other hand, considering some system processes (that we call kernel threads) only run in kernel mode, we utilize a similar approach for kernel thread registration. Before creating a kernel thread, the target kernel component should first invoke a hypercall with a secret key. Then, the hypervisor will verify the key. Next, the kernel component will issue the kernel function `kernel_thread()` to create a new kernel thread. After the kernel thread descriptor is created, the OS kernel will re-invoke a hypercall with the to register the kernel thread.

To protect the kernel thread registration from being hijacked by kernel exploits, we introduce the watchpoints to monitor the kernel's execution behavior. Specifically, we use the vmcall instructions to instrument the OS kernel so that the hypervisor can trap the kernel's key operations. To distinguish different operations that are associated with different watchpoints, each watchpoint passes a unique identifier to the hypervisor. In our implementation, we insert 4 watchpoints in the related kernel functions, which include `kernel_thread()`, `do_fork()`, `copy_process()`, and `dup_task_struct()`. In this way, the hypervisor could determine whether the kernel component follows the normal execution path to create a kernel thread. If not, it indicates that the kernel's execution could be hijacked. Thus, the newly created kernel thread cannot be registered to the whitelist.

In order to reduce modifications to the existing programs and OS kernel, we only enforce the process registration after the protected system is fully booted. In initial, the whitelist contains all the runnable processes. When a process terminates, the registered information will be removed from the whitelist.

3.2 Process Scheduling

Once the processes are registered, the scheduler is responsible for selecting a feasible process to get executed. In Linux, there are two kinds of process: non-real-time process and real-time process [7].

To prevent the unregistered non-real-time process from being scheduled, we need to modify the CFS scheduler in

```
 1 static struct sched_entity
 2 *pick_next_entity(struct cfs_rq *cfs_rq){
 3   struct sched_entity *se;
 4   struct task_struct *task;
 5   struct rq *rq;
 6 selection:
 7   se = NULL;
 8   //ensure the leftmost node exists
 9   if(first_fair(cfs_rq)){
10     //select a entity from the cfs_rq
11     se = __pick_next_entity(cfs_rq);
12     //ensure the entity contains a task
13     if(entity_is_task(se)){
14       //get a task from the entity
15       task = task_of(se);
16       while(task is not in the whitelist){
17         //generate an alert
18         alert(task);
19         //get a runqueue from the cfs_rq
20         rq = rq_of(cfs_rq);
21         //remove the task from the runqueue
22         dequeue_task_fair(rq,task,1);
23         //update the runqueue's related info
24         update_info(rq);
25         //reselect a schedule entity
26         goto selection;
27       }
28     }
29     //set related information to the entity
30     set_next_entity(cfs_rq, se);
31   }
32   return se;
33 }
```

Fig. 3 Non-real-time process scheduling algorithm.

Linux. Different from the conventional timesharing-based schedulers that assign each process a timeslice, the CFS scheduler assigns each process a proportion of the processor so that the scheduling fairness could be improved. To help modeling the multitasking processor, the CFS scheduler introduces the vruntime variable that stores the virtual runtime of a process. To decide which process to run next, the scheduler will select the one with the smallest vruntime. To facilitate the process selection, the scheduler utilizes a red-black tree to maintain the list of runnable processes. By locating the leftmost node in the tree, the scheduler can get the process with the smallest vruntime, which will be scheduled next. Before performing the process switching, our modified CFS scheduler will check whether the selected process is in the whitelist. If it is, the CFS scheduler will handle it by its default routine. Otherwise, the scheduler will pick another process to run. For this purpose, we need to hook the kernel function `pick_next_entity` in the CFS scheduler.

Figure 3 shows the specific scheduling algorithm. The key functionality of this algorithm is to set the scheduler entity (i.e., `se`) that the CFS scheduler depends on for scheduling. Initially, we set the scheduler entity to NULL (Line 7). If the CFS runqueue (i.e., `cfs_rq`) does not contain a scheduler entity, the returned `se` value remains to be empty. As a result, the CFS scheduler will schedule the idle process. On the other hand, if a scheduler entity is selected (Line 11), the scheduler should check whether it contains a task (Line 13). If it is, the scheduler need to further judge whether the task is in the whitelist. If not, an alert will be generated (Line 18). Then, the scheduler removes this task from the runqueue (Line 22). Accordingly, the runqueue's related information (e.g., the number of runnable processes and the CPU load factor in the runqueue) should get updated (Line 24). After that, the scheduler will reselect another process that could be in the whitelist (Line 26).

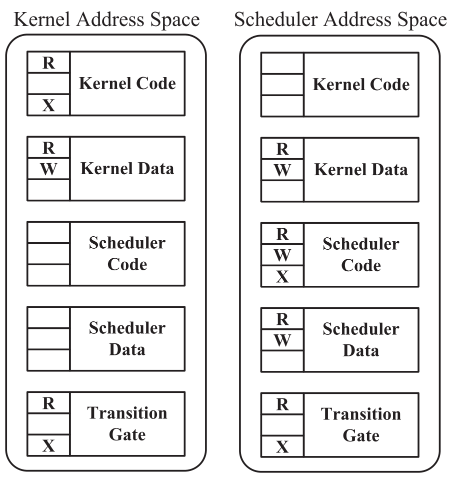

Fig. 4 Memory protections in the kernel and schedule address space.

Since the CFS scheduler and the whitelist both reside in the kernel space, advanced attackers may hijack the scheduler's execution or modify the whitelist to bypass our protection by exploiting the kernel's vulnerabilities. To tackle this problem, a traditional virtualization-based method is to move the scheduler and whitelist into the underlying hypervisor for protection. However, doing so may impose considerable performance overhead in that each process scheduling will cause the world switch between the guest OS and hypervisor. Instead, we take the advantage of the SIM framework [6], which leverages the hardware virtualization technology to achieve efficient In-VM protection. Specifically, we exploit the Shadow Page Table (SPT) management subsystem in the hypervisor to create a separate protected address space that we call scheduler address space, and then put the CFS scheduler and whitelist in it. Compared with the original kernel address space, the scheduler address space has the same memory mapping but different access permissions. In this way, the scheduler and whitelist are well isolated from the OS kernel for protection.

As shown in Fig. 4, the scheduler cannot access the kernel code and data in the scheduler address space, while the OS kernel is not allowed to access the memory regions of the scheduler and whitelist (stored in the scheduler data area) in the kernel address space. To ensure the normal execution transfer between the scheduler and OS kernel, the transition gate is introduced to switch address spaces and is always executable in both address spaces. Similar to the SIM framework, the transition gate utilizes the recent hardware feature [8] (i.e., CR3_TARGET_LIST) to change the CR3 register, which stores the root of a shadow page table for address space switching. By doing so, the execution transfers between two addresses spaces will not be trapped by the hypervisor, and the performance will be greatly improved.

On the other hand, to block the unregistered real-time process getting scheduled, some modifications to the real-time scheduler are needed. Basically, the scheduler has two different policies (i.e., SCHED_FIFO and SCHED_RR) to schedule real-time processes. Both of these policies apply a simple FIFO (first-in, first-out) algorithm to select a real-time process to run. The difference is that a SCHED_FIFO process does not have a timeslice but a SCHED_RR process does. As a result, a SCHED_FIFO process can run indefinitely until a higher priority process preempts it, while a SCHED_RR process can run only until it exhausts a predetermined timeslice.

To select a runnable process from the real-time runqueue, the kernel function `pick_next_task_rt` in the real-time scheduler is hooked. In particular, an additional judgment is added to this function for checking whether the selected real-time process is in the whitelist. If not, the scheduler will pick another process that is included in the whitelist to run. Similar to the CFS scheduler, we also apply the SIM framework to protect the real-time scheduler and its data from kernel-level attacks.

4. Evaluation

All the experiments are performed on a Dell PowerEdge T410 Server with a 2.13 G Intel Xeon E5606 CPU and 4 GB memory. The Xen hypervisor version is 3.4.3. We use Fedora 12 (2.6.31 kernel) as Dom0 system and Debian 4 (2.6.24 kernel) as DomU system with HVM mode enabled.

4.1 Effectiveness

We evaluate the effectiveness of VAW to block execution of the processes that are not included in the whitelist. For the user space evaluation, we first introduce a valid but vulnerable process that is forked by the bash program. Then, we exploit the vulnerability of this process to create a new process for executing a long-stay functionality. The experiments show that the new created process is not registered to the whitelist, and thus it cannot get scheduled. Regarding the kernel space evaluation, we load a malicious kernel module into the kernel space to fork a new kernel thread. Similarly, our system does not schedule the kernel thread because the kernel module does not have a valid key to register the kernel thread to the whitelist.

4.2 Performance

To evaluate the efficiency of our system, we test its performance with both micro and macro benchmarks. For the micro benchmark experiments, we use the LMbench tool to measure the execution time of process creation and context switch operations. The results, shown in Table 1, indicate our system introduces moderate performance overhead for these operations. For the macro benchmark experiments, we first apply the gunzip program to decompress the standard Linux kernel source package (linux-2.6.24.tar.gz), and then use the gcc program to compile the Linux kernel. As illustrated in Table 2, our system incurs low overhead as compared with the original Linux system running on the normal VM.

Table 1 Micro benchmarks.

Micro Operation	Normal VM (μs)	VAW (μs)	Overhead (μs)
Process fork+exit	121.35	234.51	113.16
Process fork+execve	325.81	482.59	156.78
Process fork+/bin/sh -c	728.62	963.78	235.16
Context switch	2.13	4.52	2.39

Table 2 Macro benchmarks.

Macro Operation	Normal VM	VAW	Overhead
Kernel decompression	35,827 ms	41,265 ms	5,438 ms
Kernel build	2,951 s	3,379 s	428 s

5. Conclusion

In this paper, we present VAW, a novel application whitelisting system based on virtualization. We exploit the process scheduling mechanism to enforce that only the processes from the whitelist can get scheduled. Moreover, we apply the SIM framework to protect the scheduler and whitelist from being compromised. Our evaluations show that VAW can prevent unauthorized processes from being executed effectively with a little performance cost.

References

[1] O. Sukwong, H. Kim, and J. Hoe, "Commercial antivirus software effectiveness: An empirical study," Computer, vol.44, pp.63–70, 2011.

[2] A. Apvrille, S. Hallyn, M. Pourzandi, and V. Roy, "DigSig: Run-time authentication of binaries at kernel level," Proc. 18th Large Installation System Administration Conference (LISA), 2004.

[3] Y. Wu and R.H.C. Yap, "Towards a binary integrity system for windows," Proc. 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2011.

[4] C. Gates, N. Li, J. Chen, and R. Proctor, "CodeShield: Towards personalized application whitelisting," Proc. 28th Annual Computer Security Applications Conference (ACSAC), 2012.

[5] A.M. Azab, P. Ning, E.C. Sezer, and X. Zhang, "HIMA: A hypervisor-based integrity measurement agent," Proc. 28th Annual Computer Security Applications Conference (ACSAC), 2009.

[6] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," 25th ACM Conference on Computer and Communications Security (CCS), 2009.

[7] R. Love, Linux Kernel Development Third ed., Addison-Wesley, 2010.

[8] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manuals, http://www.intel.com/Assets/PDF/manual/253669.pdf, 2013.