Under Cover of Darkness: Hiding Tasks via Hardware Task Switching

Kyeong Joo Jung
Stonybrook University, Ajae.dll
Incheon, Republic of Korea
kyeongjoo.jung@stonybrook.edu

Bang Hun Lee
Woosuk University, Ajae.dll
Jeonju, Republic of Korea
kindbangu@gmail.com

Yeon Nam Gung
BoB, Ajae.dll
Seoul, Republic of Korea
namska@gmail.com

Jun Seok Kim
Ahnlab Inc., Ajae.dll
Cheongju, Republic of Korea
kjsgood21@gmail.com

Hyung Suk Kim
BoB (Best of the Best) KITRI, Ajae.dll
Seongnam, Republic of Korea
kimgudtjr68@gmail.com

Ju Seong Han
Bluehole Inc., Ajae.dll
Seongnam, Republic of Korea
allmnet@naver.com

Tomaspeter Kim
BoB (Best of the Best) KITRI, Ajae.dll
Seongnam, Republic of Korea
migam2015@gmail.com

Bong Jun (David)Choi
Stonybrook University
Incheon, Republic of Korea
bjchoi@sunykorea.ac.kr

March 8, 2018

Technical Report

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands.

Abstract

Recently, malicious mining using CPUs has become a trend – mining where the task is not detected by the user is even more of a threat. In this paper, we focused on discovering IA-32 vulnerabilities over the last couple of months and found an undetectable task using hardware task switching method. It is possibly undetectable to the operating system and thus hidden from system user. Although hardware task switching methods are replaced by more convenient software switching methods in the recent years, they still exist on modern computer systems. By manually manipulating hardware task switching, which are directly managed by the CPU, we show that it is possible to create a hidden scheduler aside from the ones created by the operating system. We demonstrate using a simple CPU consumption example that these hidden tasks have potential to evolve into more sophisticated malicious attacks that can go unnoticed by users.

The important point of this research was that you can conceal these attacks from the user. Proof of the concealment will be shown with figures and tables. We also show that it is difficult to defend against hardware switching attacks because there are currently no tools that detect when Global Descriptor Table has been modified.

Table of Contents

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands

List of Figures

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands

List of Tables

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands

1 Introduction

Recently, cryptocurrencies have become a center of attention, and malicious mining cases are being newly discovered. Also, mining can be even a bigger threat in terms of CPU usage if not detected by the user.

In this paper, we propose a new method to create a task using hardware task switching which does not be traced by operating system (OS) from the user. This can be hidden from the user perspective view because the current method to switch tasks is software task switching under OS boundary. Currently, software task switching method is mostly used for convenience and faster performance than the hardware task switching method which uses the CPU directly. However, we use hardware task switching method. This means that task management won't be able to display the task from hardware task switching method because it is executing outside of the OS. In other words, we can execute another task besides from the OS scheduler and use a different scheduler which the CPU usage is undetectable in the OS. Proposed research method could be potential attacks. In normal user perspective, if the users feel the PC slow, they think the problem as overuse of CPU or virus and rootkits. The way for the normal user to check the CPU usage is through task management and a way to detect virus is to use a generally used anti-virus tools. However, since the way we suggest is not existing way of attacking yet, it won't be able to be detected through the tools.

Following steps are the way to manually operate hardware task switching. First, we implement newly made driver to a computer. Then, we modify values of GDT [6], TSS [6] using the driver that has the ring 0 authority – which is the highest authority of privileged level of computer. Last, it starts to keep on switching tasks between normal task and a malicious task. In the following sections, we present the way how hardware task switching works and show how it can work secretly.

Proposed method is fundamentally different from the existing rootkit. Prior to this research, trials to use task switching method for rootkits were none. Existing rootkits tried to handle just the

address or the link state of the processes [4]. In addition, fundamental approach of manually using the vulnerability of OS using the hardware task switching can suggest dangers and further research to prevent it.

The paper is outlined as follows. In section 2, we present the proposed method in relation to existing works. In section 3, we present our proposed method and show that a task can be hidden from the OS. In section 4, we discuss about the result of the proposed method and in section 5, we show advantages, limitations and further research this proposed method can have. At last, we conclude the paper.

2 Background

2.1 Task Switching Methods

Global Descriptor Table (GDT) is a structure from Intel processors which mostly shows the memory area used during execution of a task. The memory area is also called as a 'descriptor'. One of the descriptors that GDT contains is 'Task State Segment (TSS) descriptor' which tells about the base address of the TSS structure. Hardware task switching method uses this special structure called Task State Segment (TSS). To be specific, this structure consists of data segments that contains the state of the CPU from each task. For each time of task switching, CPU uses the corresponding TSS's information of the task to switch back. CALL or JMP instructions are most used during the task switching.

Figure 1 gives an intuition of hardware task switching. To explain, address '0x28' is the base address of TSS descriptor and it usually is set to first address of a TSS. In addition, it is mostly indicated as 'Busy' because there is always a task that needs to keep on running. Furthermore, other TSS descriptor will be set to 'available', meaning that rest of the TSS descriptor is ready to be switched. Then, the task (task 1) of corresponding TSS is executed using the state information (Extended instructor pointer (EIP) – shows the address of a task) of TSS. When the task needs to be switched, task saves the

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands

current state at the corresponding TSS. Next, when the saving is done, CPU points to the address '0x50' of GDT which refer to the base address of TSS descriptor. Lastly, TSS descriptor points to corresponding TSS and EIP points to task 2 to execute. This is how the normal hardware task switching occurs. As shown in Figure 1, TSS contains lots of information to keep every state of the task. This can be inefficient when only certain amount of the information is important, and others are similar to most of the tasks.

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands

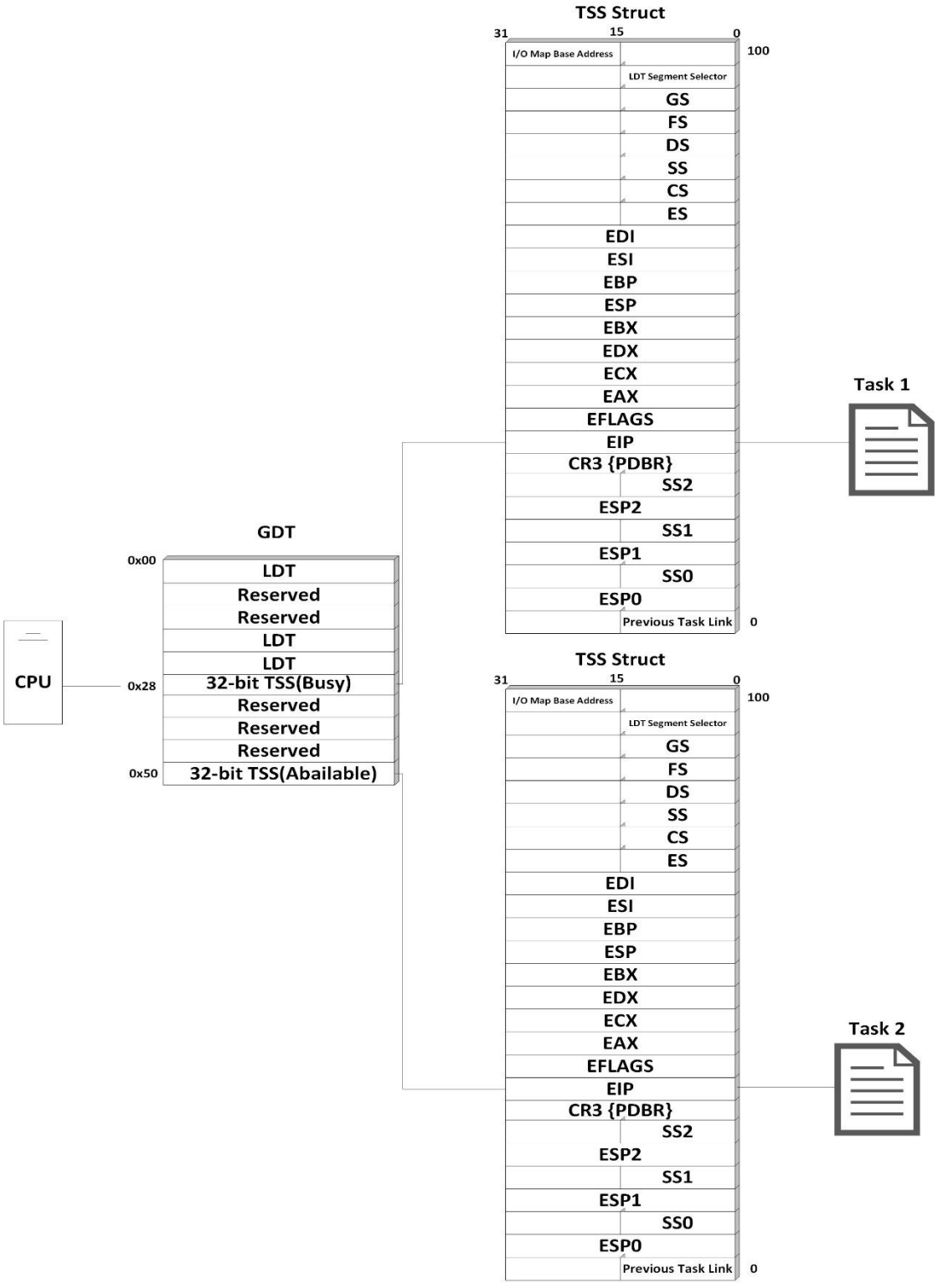


Figure 1 Hardware task switching method.

However, software task switching (a.k.a software context switching) saves and loads only the state that is needed when executing a task. This method uses the function that can manage current stack pointer to save and reload the task. If the function is called, current stack pointer which used to be pointed by current instruction pointer is stored in the old stack. In contrast, if the function is returned, new instruction pointer points to new stack pointer which will pop off. During this process, there will be only necessary information for the task execution at the stack. Most of the OS uses the software context switching instead of hardware task switching. Unfortunately, hardware task switching is mostly not used but it still exists on computers because of the compatibility. To explain, there are still servers or computers still using old OS such as Windows 95, Windows 98, etc. and that is the reason that it still exists.

2.2 Rootkit

Hiding task can be seen from a rootkit as well. A rootkit is a set of programs and codes which can attack and be undetectable on a computer [1] by hiding from the user and the OS [2]. There are various methods to create a rootkit, such as LKM (Loadable Kernel Module) [3], Hooking, DKOM (Direct Kernel Object Manipulation) [4], AL-DKOM (All Link - Direct Kernel Object Manipulation) [5], and Kernel Mode rootkit [7]. Most of the rootkits are implemented using the clandestine programs or inserting the rootkit process in the OS scheduler. No matter which method is used to create a rootkit, the rootkit is still inside the OS managing boundary. Therefore, any tasks executed by the rootkit will be traced by the OS scheduler because the OS scheduler shows every running process. Therefore, using the existing detection method, existence of the rootkits can be detected. However, proposed method is fundamentally different from the existing rootkit. This is because it uses the hardware task switching from outside of the OS boundary. Furthermore, it is not detected by the detection method since there is no detection approach to find malicious hardware task switching. Prior to this work, there was no attempt to sue the task switching for creating rootkits nor hiding tasks in general. In addition, existing

rootkits tried to handle just the address or the link state of the processes [4]. Therefore, the proposed approach of manually using the hardware task switching is fundamentally different and can lead to further research on more sophisticated attacks and countermeasures.

Kernel mode rootkit, also known as kernel level rootkit is one of the rootkit types that work at the kernel level [7]. Kernel is the deepest level of OS and it also needs higher authority to use the resources. Kernel mode rootkit itself modifies the system to compromise the target computer. It gains the access privileged authority of kernel (a.k.a ring 0) as shown in Figure 2 and uses the CPU's resources.

Most of the rootkits today use kernel mode rootkits [8] because it performs hidden behaviors easily and have access to the ring 0 authority. Ring 0 authority can also modify system values that shouldn't be modified such as GDT, and TSS values because it can cause a huge conflict in the OS. Ring 0 authority can be easily achieved just by installing a driver which is also known as '.sys' files. In this paper, using the ring 0 authority and unlike previous rootkits that modifies just the links between processes, we show modification of the system to operate hardware task switching method manually by installing '.sys' file.

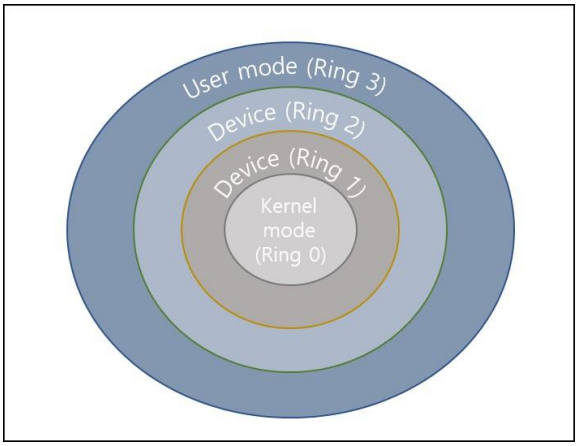


Figure 2 Privileged level of computer.

2.3 Rootkit Detection Methods

2.3.1 Behavior-Based Rootkit Detection

It attempts to detect the effects of the attack [9]. This detection method has the advantage of being able to detect the rootkits that were not detected before. In this method, the detector uses 3 steps [22] as shown in Figure 3 – Data collection, Interpretation, Matching algorithm. Data collection will collect the raw data of behavior of each program or process. Next, interpreter categorizes the programs or processes that have the similar behavior. Last, using the Matching Algorithm, when the rootkit does the unfamiliar behavior, it detects the rootkit as an attack. The drawbacks of this method are that False Positive Ratio is high and the amount of time to scan takes lots of time.

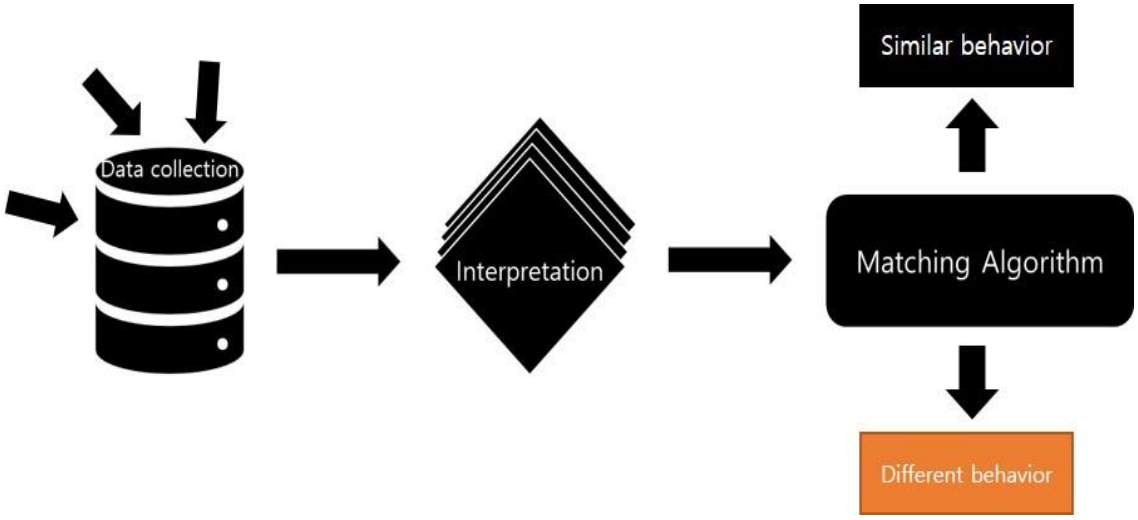


Figure 3 Behavior based detection

2.3.2 Signature-Based Rootkit Detection

It is also known as Fingerprint Identification. This method uses the pattern matching to find the attack. The pattern indicates the unique byte pattern or the signature of a rootkit [23]. The unique signature of rootkits is from already known rootkits [10] Therefore, the advantage of this method is that

it has high accuracy of detecting the rootkit. However, it also has a critical drawback that it can only detect only the known rootkits.

2.3.3 Integrity-Based Rootkit Detection

It finds out a change of system files or OS components that was unauthorized [9]. To find out the change of the file, it checks the CRC values. In addition, the detector keeps on calculating and comparing the CRC among the initial system files. The advantage of this detection is that it can detect what exact damage the rootkit has done. However, the drawback of this method is that comparing the CRC between the original and the modified can sometimes be different due to the change of the original system file itself. A typical program using Integrity-Based Detection is Tripwire.

2.3.4 Hooking-Based Rootkit Detection

[11] is relatively an easy detection method. This tries to scan the computer and find where the hooking has happened. For the service or interrupts, hookings such IDT, SSDT has function pointers in certain memory area. In addition, when rootkit modifies the hook for the malicious action, it goes out of the certain memory area that it was supposed to be which makes it easy to detect.

3 Methodology

This section presents how to create an independent scheduler besides of the OS scheduler using hardware task switching method. In addition, using independent scheduler can be a flaw of the computer and we show how this is threatening. Task Switching method can be categorized in 2 ways. First is software context switching which Windows OS currently provides. The other is hardware switching which basically Intel CPU provides. OSes use software context switching to save only the state value of the necessary ones and hardware task switching method is left abandoned mostly and be used only at the special cases such as using old OS. Therefore, we researched what happens when software context switching and hardware task switching coexists at the same time. Since everything needed for hardware task switching already exists on the computers, the only thing we need to do is to use the

tables (GDT, TSS) and system values. Currently, these values are just in simple numbers and its objective to task switch is not operated. Therefore, to use the tables and system values, we implemented newly designed driver that has a task that we created and code that makes to use the tables and systems. The reason to implement the driver is to obtain the ring 0 authority in order to use the tables and values.

In Figure 4, we set Timer value to 1 second to make each of the task switching method to work alternatively. If we don't, only one method would keep on running. This result makes the users to feel that the both switching methods execute simultaneously. In addition, results have shown that OSes cannot detect CPU scheduler using hardware task switching method. We used this undetectable feature to make the task stealthy and used EIP value in TSS to trigger the task in the implemented driver. In the following sections, we present the environment of proposed method, and show that it is executing undetected.

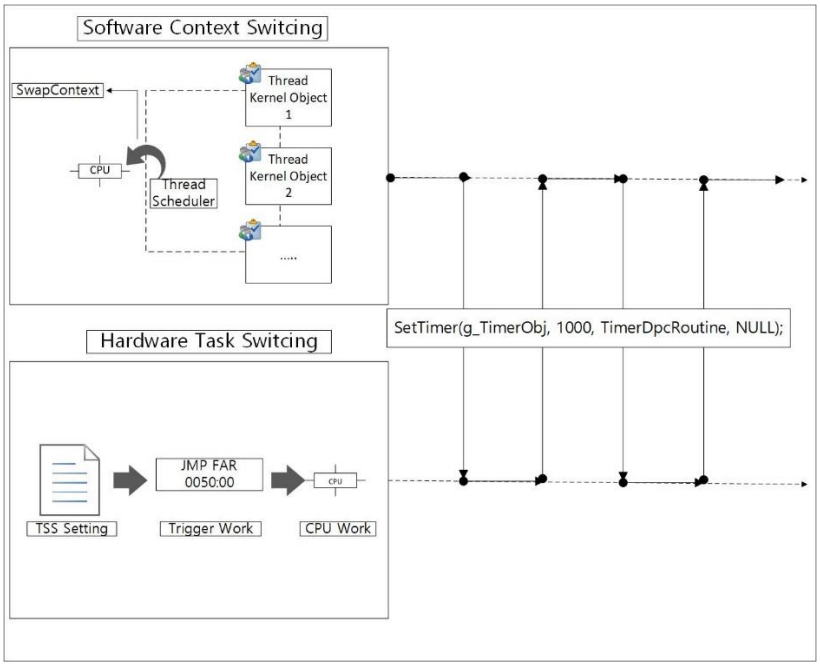


Figure 4 Adapt Timer value at implemented driver for software context switching and hardware task switching tasks to look as if they are executing simultaneously

3.1 Environment Establishment

Development environment we used is Microsoft C/C++ Compiler Driver named c1.exe for compiler, Microsoft Incremental Linker named link.exe for linker and Windows DDK for driver development. Proposed method was created in 32-bit Windows XP OS and was tested through loading Windows Driver.

```
[boot loader]
timeout=10
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/fastdetect /debugport=COM1 /baudrate=115200
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=optin /fastdetect
```

Figure 5 Windows XP Kernel Debugging Environment

Table 1 Development environment

For	Name
OS	Windows XP
Compiler	Microsoft C/C++ Compiler Driver (c1.exe)
Linker	Microsoft Incremental Linker (link.exe)
Driver	Windows DDK

32-bit Windows XP was used but it can be extended to any other 32-bit OS. Windows XP was used in this research because most of the anti-rootkit tools are working in Windows XP well and Windows XP has the least restriction for implementing rootkits that can make suitable for testing environment. The reason we use anti-rootkit tools is because the way we are hiding the task acts just like rootkits. The most important key point of the proposed method is not the version of OS. Instead, acquiring ring 0 authority through loading the driver is the key point of this research. In addition, for setting the Windows XP kernel debugging, we added '/debugport=COM1 /baudrate=115200 in C:\boot.ini'. Brief information of development environment is shown in Figure 5 and Table 1. Figure 5 is showing the kernel debugging environment and Table 1 is showing the brief development environment

of the research. Figure 6 is the structure of TSS and Windows has total of 4 for each Intel processor for 32-bit.

```
typedef struct _TSS_32 {
        WORD    prev_task_link, res0;
        DWORD   esp0;
        WORD    ss0, res8;
        DWORD   esp1;
        WORD    ss1, res16;
        DWORD   esp2;
        WORD    ss2, res24;
        DWORD   cr3;
        DWORD   eip;
        DWORD   eflags;
        DWORD   eax;
        DWORD   ecx;
        DWORD   edx;
        DWORD   ebx;
        DWORD   esp;
        DWORD   ebp;
        DWORD   esi;
        DWORD   edi;
        WORD    es, res72;
        WORD    cs, res76;
        WORD    ss, res80;
        WORD    ds, res84;
        WORD    fs, res88;
        WORD    gs, res92;
        WORD    ldt_set_selector, res96;
        WORD    t_flag, io_map_base_address;
} TSS_32, *PTSS_32;
```

Figure 6 TSS Structure

In Windows OS, one of the four TSS structure is existing at the address '0x28' and it is set to 'busy' bit. To be specific, set by 'busy' bit means that corresponding TSS structure is currently being used and at the same time, other TSS structures are set to 'available' bit which presents that they are the candidate to be used when task switching occurs. In the research, we use the address '0x50' for task switching between address '0x28'. We set timer value during the task switching operation so that it does not conflict with the original OS scheduler. If there are no timer values set, software context switching and hardware task switching cannot coexist together because only one of the 2 methods will occur not yielding themselves to be switched to the other scheduler.

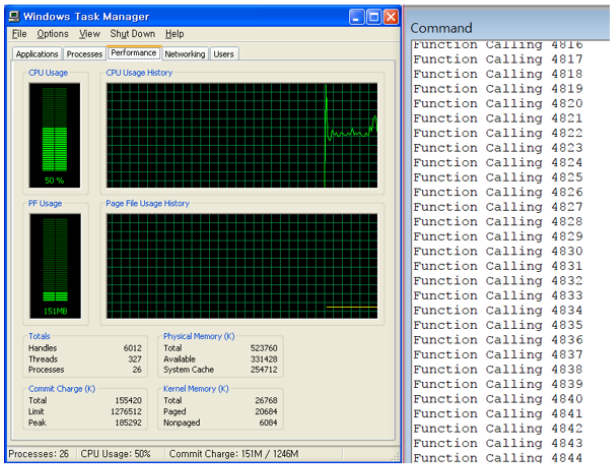
```
__asm
{
  // jmp FAR 0x50:0x00000000
  cli
  _emit 0xEA
  _emit 0x00
  _emit 0x00
  _emit 0x00
  _emit 0x00
  _emit 0x50
  _emit 0x00

}
__asm
{
  // jmp FAR 0x28:0x00000000
  cli
  _emit 0xEA
  _emit 0x00
  _emit 0x00
  _emit 0x00
  _emit 0x00
  _emit 0x28
  _emit 0x00

}
```

Figure 7 Jump statement for Hardware Task Switching

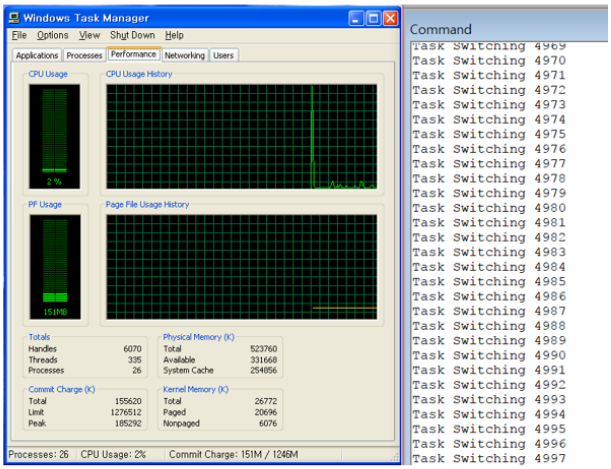
In Figure 7, it is an operation code (opcode) of jump (Jmp) to run hardware task switching. Jump statement is used so that the machine can jump to one TSS to other TSS.

3.2 Undetected task using Hardware Task Switching

In this section, we present how hardware task switching method is used as an undetected task. To explain, task executed by hardware task switching method is not detected by the OS. To prove that the created task is undetectable, we created two different scenarios where tasks are executed either with or without the proposed hardware task switching. Task can be referred to as a function in a code. Task in both scenarios repeatedly print lines until the task switching happens. Next, we verify through Windbg program to see whether the function was working properly or not and checked the CPU usage by running the task management tool provided by the OS.



(a) Normal task function call shows 55% of CPU usage.



(b) Hardware task switching call shows 2% of CPU usage.

Figure 8 Normal function call and Hardware Task Switching function call

As shown in Figure 8, the normal function call in (a) prints 'Function Calling number' which is a normal task and hardware task switching function call in (b) prints 'Task Switching number' which is a task operated through hardware task switching method. Result show that normal function call shows 55% of CPU usage and the hardware task switching function call shows almost no CPU usage below 2%. The small CPU usage is due to some other normal system tasks running in the background. In summary, execution of normal task call shows the usage of CPU but execution of same function through hardware task switching shows no CPU usage from the view point of OS. This shows that the task through hardware task switching method can use the CPU's resources without OS noticing about it. In addition, modifying EIP value to point the address of my own task using TSS can lead not to expose the source of the actual action of my own task. To be clear, as an example, with this method, computer user might know somehow that something suspicious is happening but cannot know from where this is happening. This is because the OS cannot find the source of the actual action.

Table 2 Result of detecting proposed method through existing detection tools

Detection method	Tool	Detection
Signature based detection	Icesword	X
	Malwarebytes anti-rootkit	X
	Sophos Virus Removal Tool	X
	TDSSKiller	X
Behavior based detection	nProtect Online Security	X
Integrity based detection	Afick	X
Hooking detection	Gmer	X
	Radix	X

Furthermore, using the proposed method, we used rootkit detection tools to see whether it catches the creation. Among the existing tools, proposed method was undetectable. As shown in Table 2, Icesword [12], Malwarebytes antirootkit [13], Sophos Virus Removal Tool [14], and TDSSKiller [15] are signature-based detection tools. nPortect Onlince Security [20] is a behavior-based detection tool and Afick [16] is an integrity-based detection tool. Finally, Gmer [11, 17], and Radix [18] are hooking detection tools. As a result, we found out that these tools could not detect the proposed method which uses CPU secretly from the OS.

4 Result

In this section, we present how the proposed method can be used in the real world. The key feature of the proposed method is that it executes tasks secretly from the OS. We show how the OS works during the proposed method by measuring the execution time and CPU usage. If the task through hardware task switching occurs, only the CPU utilization from OS task is calculated, not the task from the driver task. We tried to find out how the software context switching and hardware task switching can coexist. To be specific, we experimented tasks of the OS scheduler and hardware scheduler to see how the task from hardware scheduler affect OS scheduler. To find out the relation, we measured the execution time of a task of approximately 55% of CPU usage executed in OS scheduler. At the same time, we made hardware task switching happen simultaneously for many cases. For each case, we made

the task more complicate to increase the CPU utilization. When measuring the time, we calculated the total time of execution counted by the OS timer and the online timer. This is because we assumed that if the total amount of CPU utilization of tasks in both (OS and hardware schedulers) goes above 100%, then the OS might stop at that moment. In addition, this explains why OS timer stop as well, and this makes the actual time (online time) different from the OS time. Then, we calculate the average time of each OS times and online times from 100 trials. For the task run in hardware, we incremented the printed lines in the task to increase the utilization of the CPU.

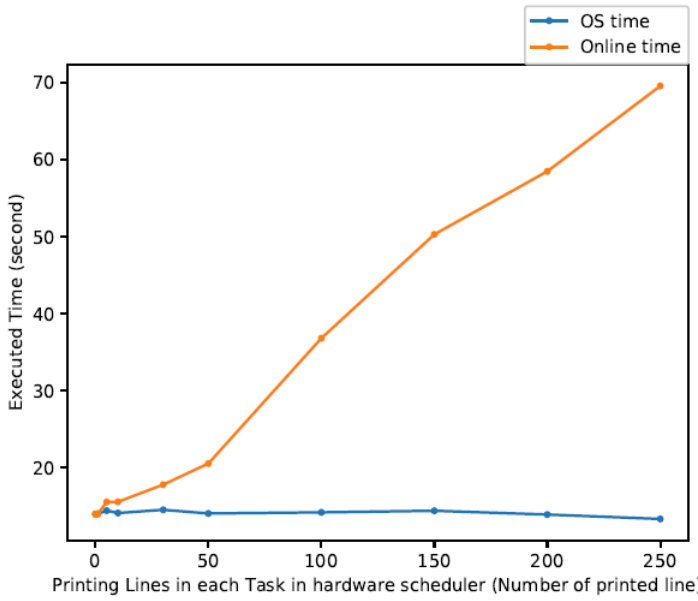


Figure 9 Executed time comparison between OS time and online time

In Figure 9, x-axis shows the number of printed lines in task from hardware scheduler which the CPU utilization increases as the lines increment. In addition, y-axis shows the seconds of OS timer, and online time it took to execute a task in OS. As the graph shows, the difference between the OS time and online time begin to increase from the point where printed lines are 50 in the task from hardware. The larger the gap between the OS time and online time becomes, there will be high possibility for the user of the OS to notice that there is something suspicious from the computer because the computer will become slow at certain time.

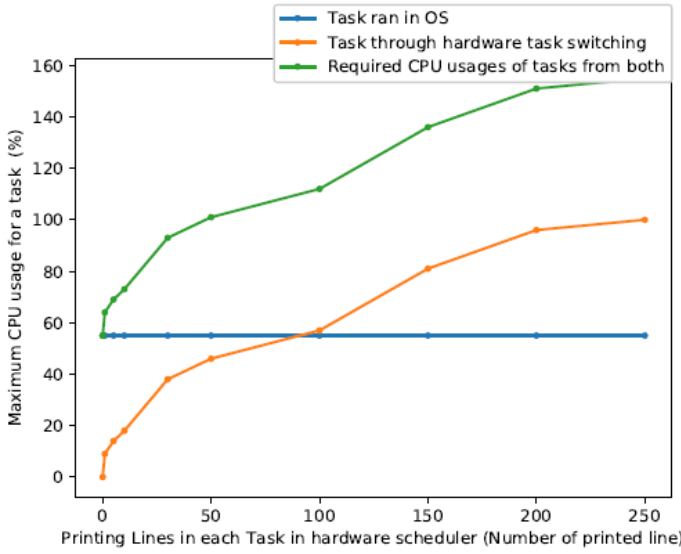


Figure 10 CPU utilization graph for normal task and task through hardware task switching

Next, to check whether the computer is becoming slow due to the task executed through hardware task switching, we measured the actual CPU utilization percentage using the task management provided by the OS. Since only one task is executed in the OS that was made to use CPU utilization of maximum 55 %, the percentage is the same during the other trials as well. The problem is measuring the CPU utilization percentage of task through hardware task switching because OS cannot detect the task of hardware scheduler. Therefore, in the experiment, we calculated the CPU of the same task in driver that does not use hardware task switching method. It allows OS to detect the task and measure the CPU utilization percentage again. As shown in Figure 10, x-axis is showing the printed lines from the task and y-axis shows the maximum CPU usage for each case. The green line shows the required CPU usages from both tasks, orange line shows the maximum usage from task using hardware task switching, and blue line shows the maximum usage from task in OS. As shown in Figure 10, printed lines of 50 is the point where the sum of percentage of CPU utilization goes above 100 %. To explain, this indicates lots of meaningful signs.

Table 3 Result of task execution

Number of printed lines in task in hardware	0	1	5	10	30	50	100	150	200	250
CPU utilization of task in hardware (%)	0	9	14	18	38	46	57	81	96	100
CPU utilization of task in OS (%)	55	55	55	55	55	55	55	55	55	55
Required CPU utilization of both tasks (%)	55	64	69	73	93	**101**	**112**	**136**	**151**	**155**
Execution time of task using OS time (sec)	13.99	14.03	14.44	14.13	14.55	14.07	14.21	14.42	13.93	13.36
Execution time of task using online (sec)	13.96	13.99	15.56	15.58	17.81	20.55	36.8	50.27	58.45	69.53
Time difference of both time (sec)	-0.04	-0.04	1.12	1.45	3.26	**6.48**	**22.58**	**35.85**	**44.52**	**56.17**

In Table 3, required CPU utilization can be calculated by adding the results from executing the tasks from both the OS and hardware schedulers. If the sum of CPU usage of both (Required CPU utilization) tasks become over 100%, it would make the user feel that the computer is becoming slow. This is because as the required CPU utilization reaches over 100%, the time difference of the execution time between the OS timer and the online timer increases. In this case, user will become suspicious about the computer and will start to actively seek for solutions. However, this is not the behavior of the user that attacker wants. Therefore, as long as the CPU usage stays below 100%, it will not change the user's quality of experience. Moreover, this approach can be extended to dominate the CPU resource of servers. Since many users share a server, it will become more difficult to detect as the users may think that the resource is simply occupied by other users. Another approach is to use low CPU resources from many computers, such as Zombies, so that users are completely unsuspecting the attack. A similar work was done previous in [19]. However, [19] is fundamentally different from our method that it used the hypervisor authority to use the CPU and it is for the virtual machine environment. In addition, hypervisor authority is an authority which is hard to achieve while the proposed method needs the ring 0 authority.

5 Discussion and Recommendation

In this section, we discuss about the limitations, the possible detection method and further research. Proposed method has some advantages such as not being detected from the OS and a totally different approach that has not been used for making rootkit previously.

Furthermore, there is a need for a caution that ring 0 authority has much higher authority than it should have. This is because by only achieving ring 0 authority, you can modify TSS, GDT values which could make a possibility of making another scheduler. There shouldn't be any scheduler except the OS scheduler because it could conflict the system itself. There should be an effort to reduce the authority of ring 0.

Limitation of the proposed method has is that it can only work for 32-bit OS. However, there are still many servers using 32-bit OS and theoretically 64-bit OS is also possible to make another scheduler besides the OS scheduler using software context switching method. In addition, extension to 64-bit Windows and other operating systems is in progress.

6 Conclusions

In this paper, we proposed a new way to create an undetectable task using hardware task switching. The ring 0 authority can be achieved easily through installing a newly created driver. Being able to create a new scheduler outside of the OS management boundary can make the system vulnerable and can give rise to various new attacks. Attacks can dominate server's CPU usage or mine cryptocurrencies from the users' computers without being noticed. Therefore, further research effort is needing to detect such attacks by designing more sophisticated detection methods that also examines the GDT and TSS structure. We also recommend that the authority of ring 0 needs to be reduced or be redesigned to consider various prevailing scenarios.

7 Acknowledgements

This research was funded by the MSIT, Korea, under the "ICT Consilience Creative Program" (IITP-2017-R0346-16-1007) supervised by the IITP, by the KEIT, Korea, under the "Global Advanced Technology Center" (10053204), and by the NRF, Korea, under the "Basic Science Research Program" (NRF-2015R1C1A1A01053788).

8 References

[1] Greg Hoglund and Jamie Butler. 2005. Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional.

[2] John Harrison Spencer Smith. 2012. Rootkits. (2012). https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/rootkits-12-en.pdf.

[3] J. M. De Goyeneche and E. A. F. De Sousa. 1999. Loadable kernel modules. IEEE Software 16, 1 (Jan 1999), 65–71. https://doi.org/10.1109/52.744571.

[4] J. Butler. 2004. DKOM (Direct Kernel Object Manipulation). (2004). http: //www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[5] Seong-Jin Byeon Dongho Kim Ji-Won Choi, Sang-Jun Park. 2017. Dual-Mode Kernel Rootkit Scan and Recovery with Process ID Brute-Force. Advanced Science Letters 23, 3 (March 2017), 1573–1577. https://doi.org/10.1166/asl.2017.8624.

[6] Intel 2016. Intel 64 and IA-32 Architectures Developer's Manual, chapter 6, chapter 7. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html.

[7] Jestin Joy, Anita John, and James Joy. 2011. Rootkit Detection Mechanism: A Survey. Springer Berlin Heidelberg, Berlin, Heidelberg, 366–374. https://doi.org/10.1007/978-3-642-24037-9_36.

[8] Sungkwan Kim, Junyoung Park, Kyungroul Lee, Ilsun You, and Kangbin Yim. 2012. A Brief Survey on Rootkit Techniques in Malicious Codes. Journal of Internet Services and Information Security (JISIS) 2, 3/4 (11 2012), 134–147.

[9] Jamie Butler Sherri Sparks. 2005. SHADOW WALKER Raising The Bar For Rootkit Detection. (2005). http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf.

[10] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh. 2013. A survey on heuristic malware detection techniques. In The 5th Conference on Information and Knowledge Technology. 113–120.

[11] Thomas Martin Arnolda. 2011. Comparative analysis of rootkit detection techniques. Master's thesis. The University of Houston Clear Lake.

[12] Softonic. 2016. Icesword. (2016). http://icesword.en.softonic.com

[13] Malwarebytes. 2017. Malwarebytes anti-rootkit beta. (2017). https://www. malwarebytes.com/antirootkit.

[14] Sophos. 2017. Sophos Virus Removal Tool. (2017). https://www.sophos.com/en-us/products/free-tools/sophos-anti-rootkit.aspx.

[15] Kaspersky lab. 2017. TDSSKiller. (2017). https://support.kaspersky.com/viruses/disinfection/5350#block3.
[16] Afick. 2017. Afick. (2017). http://afick.sourceforge.net/index.html.
[17] Przrmyslaw Gmerek. 2016. Gmer. (2016). http://www.gmer.net.
[18] Unique Security Software. 2008. Radix. (2008). https://www.usec.at/faq.html.
[19] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. 2011. Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing. In 2011 IEEE 10th International Symposium on Network Computing and Applications. 123–130. https://doi.org/10.1109/NCA.2011.24.
[20] nProtect. 2017. nProtect Online Security. (2017). https://nos.nprotect.com/landing/woodhuston/.
[21] A. Juels, M. Jakobsson, and T. N. Jagatic. 2006. Cache cookies for browser authentication. In 2006 IEEE Symposium on Security and Privacy (S P'06). 5 pp.– 305. https://doi.org/10.1109/SP.2006.8.
[22] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," Journal in Computer Virology, pp. 251–266, 2008.
[23] P. Gutmann., 2007 "The Commercial Malware Industry.", University of Auckland, https://www.educause.edu/ir/library/pdf/CYB07002.pdf.

This paper was presented at HITBSecConf 2018, April 9–13, 2018, Amsterdam, Netherlands