Using Hypervisors to Overcome Structured Exception Handler Attacks

Asaf Algawi[1], Michael Kiperberg[3], Roee Leon[1] and Nezer Zaidenberg[2]
[1]University of Jyväskylä, Finland
[2]College of Management Academic Studies, Rishon LeZion, Israel
asaf@trulyprotect.com
michael@trulyprotect.com
roee@trulyprotect.com
nezer@trulyprotect.com

Abstract: Microsoft windows is a family of client and server operating systems that needs no introduction. Microsoft windows operating system family has a feature to handle exceptions by storing in the stack the address of an exception handler. This feature of Microsoft Windows operating system family is called SEH (Structured exception handlers). When using SEH the exception handler address is specifically located on the stack like the function return address. When an exception occurs the address acts as a trampoline and the EIP jumps to the SEH address. By overwriting the stack one can create a unique type of return oriented programming (ROP) exploit that force the instruction pointer to jump to a random memory address. This memory address may contain random malicious code. Multiple Microsoft Windows applications are particularly vulnerable to this type of exploit. Attacks on Microsoft Window application that exploit these mechanisms are found in many common windows applications (including Microsoft Office, Adobe Acrobat, Flash and other popular software). These attacks are well documented in CVE database in numerous exploits. We previously described how hypervisors can be used to white list an end point and provide application control for a workstation and servers and protect against malware and viruses that may run on the end point computer. In this work we extend the protection mechanism for end points and servers that uses the hypervisor to white list the machine. The hypervisor detects permission elevation from user space to kernel space (system calls invocation) and detects anomalies in the software execution. The hypervisor based mechanism allows for detection and prevention of SEH return oriented exploits execution. Our hypervisor based SEH-exploit prevention mechanism was tested on multiple well documented CVE vulnerabilities. Our hypervisor was found to prevent a large collection of different types of SEH exploits in multiple applications and multiple flavours and versions of Windows OS in both 32 and 64 bit environments

Keywords: SEH, rootkit, application control, hypervisor

1. Introduction

Structured exception handlers (SEH) are a family of Return oriented Programming (ROP) (Roemer et al 2012)attacks that infest windows hosts.

Hypervisors (Zaidenberg 2018) can be used for various security purposes. One common usage for hypervisors for system security is to protect End points and allow only predefined software will receive execute permissions and allowed to run as was shown by Seshadri et al (2007) and Resh et al (2017) and others.

However even by controlling the entire guest system memory the hypervisor is still not capable of protecting against return oriented programming. In this work we would like to briefly describe prior attempts at using hypervisors to control and attest the guest system and explain why they are not sufficient for preventing SEH and other ROP attacks.

We will describe our method that allows a thin hypervisor to provide the guest operating system protection against SEH attacks.

2. Background

Hypervisors are modern software components that are designed to run multiple operating systems on a single hardware device. Thus the hypervisor (called the host) is catching all memory access and all hardware interrupts and delivers them to the operating systems that it runs (called guests). Thus the hypervisor has a relationship with the guest that is similar to the relationship that the normal operating system has with a process.

However a special type of hypervisor has been proposed the thin hypervisor. The thin hypervisor supports only one operating system and leaves all (or most) of the handling of hardware interrupts, memory allocation and other system events to the guest the operating system. The thin hypervisor is only a monitor that supervise a

specific set of predefined event. TrulyProtect thin hypervisor for execution protection or anti reverse engineering (Averbuch et al 2013, Kiperberg et al 2016) as well as SecVisor for end point security and the blue pill are examples for such thin hypervisor.

However when implementing thin hypervisor for system protection we discovered a class of attacks against code that does not run as an application but as complex class of ROP attacks called SEH.

2.1 Using hypervisor for forensics

Hypervisors have long been used as development tools. (Khen et al 2011, Khen et al 2013) They can even be used in order to detect security weaknesses on a target systems (Zaidenberg et al 2015). Later Kiperberg (Kiperberg 2019) has shown that the hypervisor can inspect the entire guest memory. It follows such a tool can be used in ensuring the end point security. Thus the goal is, given an end-point which is running off the shelf operating system. The hypervisor can be verified and found to be Trustworthy (Zaidenberg et al 2015). Thus a chain of trust be formed. I.e. can a remote 3rd party ensure that the end point is running code from pre-defined white list.

2.2 Using hypervisors for end point security

A remote host can attest the hypervisor and verify the remote system correctness using side effects of computation (Zaidenberg and Resh 2015). This was shown in principle by Kennell (Kennell et al 2002) and in practice on modern x86 processors. (Kiperberg et al 2013, Kiperberg et al 2015) once the end point is running a trusted hypervisor it was shown by Seshadri et al 2007 and Resh et al 2017 that the end point can be counted on to only intentionally run only code from predefined white lists. However, code can come from many sources not only intentionally. Self-modifying code and Return oriented programming code can modify in memory and thus execute code (as an attack) that was not originally present in the white list. W^X or Data execution prevention (DEP) may prevent some attacks of self-modifying code but on a modern window host some pages are required to have write (W) and execute (X) permissions thus W^X is impossible for those pages. Some of these memory pages are SEH pages which are thus vulnerable to exploit even if a protecting hypervisor is running. Furthermore, These SEH pages are well known as part of a complete class of attacks against various software. Thus, it is required to provide windows hosts protection against SEH attacks.

2.3 Windows SEH handlers and SEH attacks

In Intel's x86 instruction set architecture an exception is an event that is triggered during the execution of a program. The exception requires the immediate execution of code outside the normal flow of control. In x86 architecture there are two kinds of exceptions: hardware exceptions and software exceptions. Hardware exceptions are initiated by the CPU. They can result from the execution of certain instruction sequences, such as division by zero or an attempt to access an invalid memory address. Software exceptions are initiated explicitly by applications or the operating system. For example, the system can detect when an invalid parameter value is specified. The exception mechanism is employed by many programming languages ranging from native languages such as C++ to interpreted or JIT languages such as Java and Javascript. In windows Microsoft had developed a mechanism called SEH, which stands for Structured Exception Handling, this mechanism allows native windows code to handle both software and hardware exceptions. The SEH mechanism allows developers for the native platform to use _try, _except & _finally keywords to handle exceptions. It is worth noting that unlike other C++ exception mechanisms, SEH allows for _finally which is not standard exception handling in RAII languages like C++.

SEH mechanism works in the following manner, each TIB (Win32 Thread Information Block) hold a pointer to a list of structures called *_EXCEPTION_REGISTRATION_RECORD as shown in Figure 1*

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
    PEXCEPTION_REGISTRATION_RECORD Next;
    PEXCEPTION_DISPOSITION Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

Figure 1 Windows _EXCEPTION_REGISTRATION_RECORD

Each link on the list has two pointers, the first one is a pointer to the next link on the chain, the second is a pointer to the function *_except_handler3* which in turn calls the exception _except and _finally blocks for the developer.

SEH is entirely implemented in the Microsoft window's visual studio compiler. When Microsoft (visual studio) compiler "sees" a *_try* block it will generate a call to a function called *EH_PROLOG.* The EH_PROLOG function will in turn create the *_EXCEPTION_REGISTRATION_RECORD* structure on the user stack and put it at the head of the chain. Once the Microsoft compiler "sees" an exit from the protected block the compiler will generate a call to *EG_EPILOG* which will remove the aforementioned structure from the stack. Every SEH chain in Windows ends with *kernel32!UnhandledExceptionFilter*, a predefined function which shows the "A Program had stopped working" dialog and kills the program's process.

The SEH mechanism is notorious for its vulnerability for security exploits. Since SEH handlers and pointers are written to the stack, it is possible for a user to control these pointers after an invalid memory operation or assignment such as memcpy or strcpy without (or with incorrect) bound checking. The basic idea is to control the pointer of the second link in the chain, making it jump into some random shell code inside the user stack. This is shown in Figure 2 below.

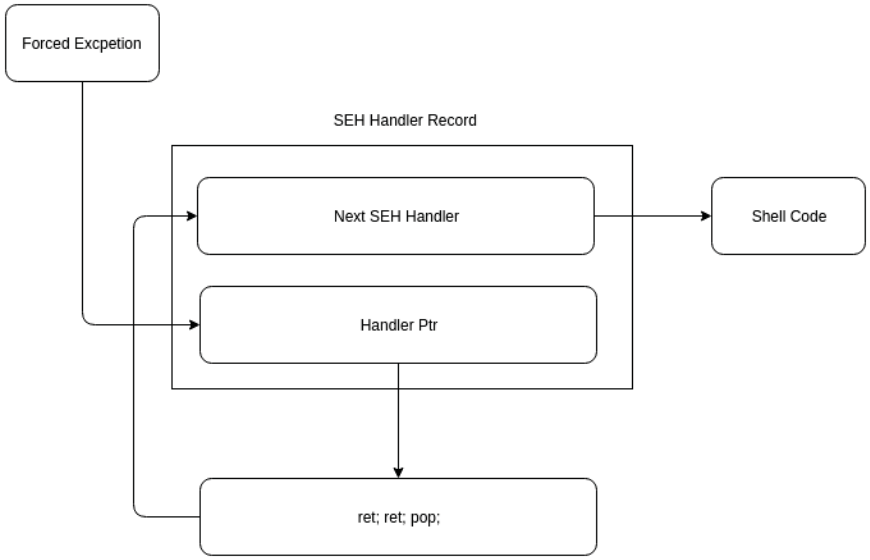

Figure 2: SEH attack operation logic.

We utilize the fact that the prolog function *EG_PROLOG* is called with the stack setup as follows, the top of the stack contains a ptr to the second link in the chain + 8 bytes, therefore issuing the sequence *pop; pop; ret;* will setup the stack in such manner the 8 bytes are gone, the top of the stack contains a ptr to the second link in the chain to which we return.

The attacker has to prepare a payload which contains the following:

[PAD][Next SEH Record][SEH Record][Shell code]

While we remember that each record is exactly 8 bytes, the pad is the distance between the buffer and the top of the chain. The exact payload obviously differs between each program but the structure is mostly the same, the pad can be any junk, the `Next SEH Record` contains a JMP instruction with an address of the appended shell code (which should fit in 8 bytes), the `SEH Record` part of the payload is a pointer to an address inside the program containing the special *pop; pop; ret;* code. Figure 2 demonstrates SHE operation logic

Some well-known attacks that utilize the above system are CVE-2018-9059, Exploit DB 38486, Exploit DB 38516 and Exploit DB 44218 and many others. All the above exploits use the same principal highlighted above for similar attacks on different applications.

3. Solution

In order to circumvent attacks based on manipulating the SEH chain we use a special hardware features available as part of Intel's (and AMD's) hardware virtualization instruction set. The hypervisor can specify certain 'events' which cause an exit (VMEXIT) from the guest code to the hypervisor exit handler. One such event that causes a VMEXIT is the CPUID instruction. The thin hypervisor uses this feature to force an exit from guest to host whenever the function *KiUserExceptionDispatcher* starts. We catch this function's execution because it is responsible for unwinding the SEH chain during an exception. Once we catch that function we can inspect the entire SEH chain and make sure no function contains the malicious *pop;pop;ret* sequence or any sort of code located on the stack, especially such code which does a near jump to the stack. Once such template of malicious code is found, the Hypervisor can easily close the program safely without the malicious payload executing.

3.1 Performance impact

Systems such as SecVisor or TrulyProtect bears some performance cost from the sole reason of running a hypervisor.

Since the systems require look up two layers of memory indirection (OS and Hypervisor level) as opposed to only single level some memory performance penalty is expected. This penalty was measured by VMWare and other sources and was found to be between 0 and 5% depending on application.

The proposed system still suffer from this performance penalty. (Since the system is using a thin hypervisor it suffers from slightly reduced penalty when compared with full hypervisor such as VMWare ESXi but some penalty must exist)

Furthermore, slight additional penalty is added due to the loading of programs to memory and calculating the page hashes when the software is first loaded to memory but not on standard usage. This penalty is also associated with the general hypervisor-based protection and not with the costs associated with the protection system proposed here-in.

The only cost that is associated with the system described herein is the cost involved from handling exceptions during standard operations. Since exceptions are not called regularly in normal applications (we measured less then 1 exception per second) the costs associated with the system are truly minimal. (less than 0.1% in our system testing and within random operation variance)

4. Conclusion

We have shown a critical weakness in using a well-known weakness family in hypervisor based end point protection. The aforementioned weakness allows execution of random code through a call for ROP. The ROP code execution is performed by exploiting the Windows SEH weakness. When using the weakness an attacker can inject random code to a hypervisor protected windows host. The code will execute despite the hypervisor based white listing as the memory pages are modified in memory.

We have also shown means to mitigate this weakness. By specifying the needed values on VMCS (Intel) or VMCB (AMD) structure hypervisors can mitigate this SEH attack and enforce the white list. This method is not preventing all ROP attacks not associated with Exceptions. DEP should still be used to defeat these attacks. However since there are several attacks against DEP as well (e.g. Stojanovski 2007) additional research is required to provide full defence against all forms of Return oriented programming (ROP).

References

Averbuch, A., Kiperberg, M., & Zaidenberg, N. J. (2013). Truly-protect: An efficient VM-based software protection. *IEEE Systems Journal*, *7*(3), 455-466.

CVE-2018-9059 "Stack-based buffer overflow in Easy File Sharing (EFS)" NIST https://nvd.nist.gov/vuln/detail/CVE-2018-9059

Exploit DB 38486 Tomabo MP4 Player 3.11.6 - Local Stack Overflow (SEH) https://www.exploit-db.com/exploits/38486

Exploit DB 38526 Easy File Sharing Web Server 7.2 - Remote Overflow (SEH) https://www.exploit-db.com/exploits/38526

Exploit DB 44218 IrfanView 4.50 Email Plugin - Buffer Overflow (SEH Unicode) https://www.exploit-db.com/exploits/44218

Khen, E., Zaidenberg, N. J., & Averbuch, A. (2011, June). Using virtualization for online kernel profiling, code coverage and instrumentation. In *2011 International Symposium on Performance Evaluation of Computer & Telecommunication Systems* (pp. 104-110). IEEE.

Khen, E., Zaidenberg, N. J., Averbuch, A., & Fraimovitch, E. (2013, July). Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation. In *2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)* (pp. 78-85). IEEE.

Kiperberg, M., & Zaidenberg, N. (2013) Efficient Remote Authentication. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*(p. 144). Academic Conferences Limited.

Kiperberg, M., Resh, A., & Zaidenberg, N. J. (2015, November). Remote Attestation of Software and Execution-Environment in Modern Machines. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing* (pp. 335-341). IEEE.

Kiperberg, M., Resh, A., & Zaidenberg, N. (2016). *U.S. Patent No. 9,471,511*. Washington, DC: U.S. Patent and Trademark Office.

Kiperberg M, Algawi A, Leon R, Resh A. & Zaidenberg N. J. Hypervisor-assisted Atomic Memory Acquisition in Modern Systems (2019) in Proceedings of 5th international conference on information system security and privacy ICISSP 2019

Resh, A., Kiperberg, M., Leon, R., & Zaidenberg, N. J. (2017). Preventing Execution of Unauthorized Native-Code Software. International Journal of Digital Content Technology and its Applications, 11.

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, *15*(1), 2.

Seshadri, A., Luk, M., Qu, N., & Perrig, A. (2007, October). SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Operating Systems Review* (Vol. 41, No. 6, pp. 335-350). ACM.

Stojanovski, N., Gusev, M., Gligoroski, D., & Knapskog, S. J. (2007, April). Bypassing data execution prevention on microsoftwindows xp sp2. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*(pp. 1222-1226). IEEE.

Zaidenberg, N. J. (2018). Hardware Rooted Security in Industry 4.0 Systems. *Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures*, *51*, (pp 135-151).

Zaidenberg, N. J., & Khen, E. (2015, November). Detecting Kernel Vulnerabilities During the Development Phase. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing* (pp. 224-230). IEEE

Zaidenberg, N., & Resh, A. (2015). Timing and side channel attacks. In *Cyber Security: Analytics, Technology and Automation* (pp. 183-194). Springer, Cham.

Zaidenberg, N., Neittaanmäki, P., Kiperberg, M., & Resh, A. (2015). Trusted Computing and DRM. In *Cyber Security: Analytics, Technology and Automation* (pp. 205-212). Springer, Cham.