Address Space Layout Randomization & Data Execution Prevention in Modern Windows Operating Systems

By

Andrew L Kerby

An Assignment
Submitted to the Faculty of
Mississippi State University
In Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

April 2017

ADDRESS SPACE LAYOUT RANDOMIZATION AND DATA EXECUTION PREVENTION IN MODERN WINDOWS OPERATING SYSTEMS

Many attacks on modern Windows (7/8.1/10) systems exploit vulnerabilities in memory management to execute malicious code [5]. In an effort to reduce the size of the attack surface, address space layout randomization (ASLR) and non-executable memory spaces are used in many modern operating systems. Modern Windows systems use ASLR and a technology called Data Execution Prevention (DEP) to reduce the vulnerability to these memory exploitation attacks [2][6]. The details of ASLR & DEP in modern Windows operating systems will be discussed, as well the limitations of these technologies in successfully preventing memory corruption attacks.

The goal of including address space layout randomization and/or data execution prevention in a modern operating system is to prevent the hijacking of non-malicious code sitting in main or secondary memory for the purpose of the execution malicious code or creating maliciously purposed code from non-malicious code [2][6]. One such attack can be carried out using a technique known as return-oriented programming (ROP) [6]. Using the concept of ROP, it is possible for an attacker to chain together small pieces of code sitting in main or secondary memory to achieve a malicious purpose, despite the fact that the original code may be a very benign program [6]. A simple program such as *notepad.exe* on the Windows Operating system, though not designed with malicious intent, contains code that executes on the processor nonetheless. In fact, when examined,

it was found that even relatively small programs of a few kilobytes could contain enough useful assembly code instructions to be successfully exploited using ROP [6].

For example, as a first step, an attacker may find an instruction that adds the contents of the register *EAX* with some arbitrary value. Continuing to keep in mind the ROP principles, an attacker may handpick a few instructions of assembly code, that, for example, pushes contents of the register EAX onto the stack [6]. In a third step, the attacker might decide to call the assembly code instruction *ret*, which pops the value of EAX at the top of the stack off, then begins executing memory at that location. The attacker has just used this *ret* instruction to jump to a place in memory, where a malicious piece of executable code may be, such as a shell like the Windows command prompt, *cmd.exe*. With this the attacker has gained a significant amount of access to the system. The use of the *ret* instruction, short for return, is where the term Return-Oriented Programming gets its name [6].

In order to prevent this, a technology like Microsoft's Enhanced Mitigation Experience Toolkit (EMET), included in modern Microsoft operating systems, can use a technology such as Address Space Layout Randomization to prevent or significantly reduce the chances that an attacker could carry out a ROP attack [2]. Without EMET, there is a specific compile flag that must be set in order for the address space layout of a compile program to be randomized, but EMET enforces mandatory ASLR, even on programs that were not compiled with this specific compile flag, although this only occurs in Windows 8.1 and above [2]. The core principle behind address space layout randomization is to obscure from the attacker the exact memory locations where instructions that could be used in a ROP-based attack might be located [2]. If the attacker

does not know or cannot possibly predict the address of the memory location where a useful piece of code, then ROP attacks become difficult to carry out [6]. For example, in code with a static address space, a user with malicious intent needs to analyze which address she needs to push on the stack, then call the *ret* instruction to pop this value off the stack and jump there. However, in a modern operating system like Windows 7, 8.1, or 10, Microsoft's implementation of EMET, which randomizes the way executable code is stored in memory, there is a very low probability that the ROP sequence that was used as a successful exploit on static code will succeed on a randomized address space, as the hexadecimal memory address values will mostly likely not result in the same, carefully-crafted sequence of changes in control-flow, so the ROP exploit will fail [2][6]. Microsoft's EMET also secures against ROP exploits by protecting vulnerable Windows API's such a *VirtualAlloc* [2], in addition to using ALSR.

There is a type of attack that called Heapspray Allocation that increases the chances of landing on injected shellcode successfully by placing many, small areas of malicious code repeatedly throughout the heap, rather than just using one copy of the malicious code, potentially reducing the effectiveness of ASLR [2]. EMET also has a method of mitigating this type of attack by allocating in advance the types of data pages that are commonly used, reducing the area on the heap onto which shellcode could be sprayed [2]. This an example of how Microsoft is constantly responding to new exploits as the security industry moves forward, building on the back of ASLR & DEP.

Another protection that Microsoft's EMET puts in place is DEP [2], which has the benefit of preventing malicious code from being executed on data structures such as the heap and the stack [1]. DEP has been available on versions of the Windows operating

system since Windows XP and Windows Server 2003 was released [1]. There are two types of DEP present on the Windows operating system, Hardware-enforced DEP and Software-enforced DEP [1]. Hardware-enforced DEP is the more powerful of the two, as it is built into the hardware itself by both Advanced Micro Devices (AMD) and Intel, although they use the terms no-execute page-protection and Execute Disable Bit to describe the same technology [1][3]. The reason hardware-enforced DEP is more powerful is that by default, all pages in a process are explicitly marked as non-executable via a bit in the page table unless there is legitimate executable code present in that page [1]. If there is code present, then DEP will isolate these sections as the areas in memory that are allowed to execute [3]. Therefore, if the no-execute bit is set in the page table, then a hardware exception will the thrown if the instruction pointer is made to point to this location [B][6]. Software-enforced DEP provides the protective benefits of DEP only to the Windows system binaries, so it is quite a bit more limited than Hardware-enforced DEP [1].

Despite all of the protection that DEP puts in place, there are still ways in which DEP can be disabled for an entire process, despite the no-execute bit being set initially on an area of non-code memory [4]. When Hardware-enforced DEP is enabled and the no-execute bit set, it would not be possible to inject malicious code into a process by a buffer-overflow attack and have code written to the overflowed memory area be executed without raising an exception [1][4]. However, Windows has a built-in API called *NtSetInformationProcess* that is called by the *ntdll* library that can be used to produce a successful exploit by allowing the no-execute bit to be disabled at runtime for a memory area, allowing malicious code to be executed [4]. However, the Windows API that allows

for the exploit can also be uses to guard against it. Processes can be set to never allow changes to update flags using by calling *NtSetInfomationProcess* with a specific flag, therefore preventing a second call to *NtSetInfomationProcess* to change the no-execute bit over the entire lifetime of the process [4].

In conclusion, it was shown that despite the existence methods of attack such as ROP that can utilize non-malicious code to be used maliciously, a tool such as Microsoft's EMET can enforce ASLR on applications that may otherwise be left vulnerable due to the oversight of leaving compile-time flags unset. ASLR has the ability to eliminate an entire attack surface in vulnerable applications, making it very worthwhile to employ. DEP, which is both built into the hardware architecture and supported by Microsoft's Windows operating system, can provide further protections where ASLR fails to successfully prevent attacks.

REFERENCES

[1] ”A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003” *Microsoft Corporation*, Revision 3, 2017, https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in-windows-xp-service-pack-2-windows-xp-tablet-pc-edition-2005-and-windows-server-2003 (accessed Apr. 26, 2017).

[2] ” Enhanced Mitigation Experience Toolkit 5.52 User Guide” *Microsoft Corporation*, 2016, https://www.microsoft.com/en-us/download/details.aspx?id=54265 (accessed Apr. 26, 2017).

[3] Lobo, D, et al., “Windows Rootkits: Attacks and Countermeasures”, *Second Cybercrime and Trustworthy Computing Workshop*, Jul. 2010. https://www.researchgate.net/publication/50875025_Windows_Rootkits_Attacks_and_Countermeasures (accessed Apr. 26, 2017).

[4] Miller, M, ” A Brief History of Exploitation Techniques & Mitigations on Windows”, 2007, https://repo.palkeo.com/repositories/repo.zenk-security.com/Techniques%20d.attaques%20%20.%20%20Failles/A%20Bried%20of%20Exploitation%20Techniques%20and%20Mitigations%20on%20Windows.pdf (accessed Apr. 26, 2017).

[5] Nemeth, Zoltan, “Modern Binary Attacks and Defenses in the Windows Environment – Fighting Against Microsoft EMET in Seven Rounds”, *13th IEEE International Symposium on Intelligent Systems and Informatics*, 2015, https://www.researchgate.net/publication/282284249_Modern_Binary_Attacks_and_Defences_in_the_Windows_Environment__Fighting_Against_Microsoft_EMET_in_Seven_Rounds (accessed Apr. 26, 2017).

[6] Schwartz, Edward, “The Danger of Unrandomized Code”, *;login:*, Vol 36, No. 6, Dec. 2011. https://www.usenix.org/system/files/login/articles/105516-Schwartz.pdf (accessed Apr. 26, 2017).