Windows Internals: Advanced Exploit Mitigation

Fabian Nguyen

Abstract

We take a look at general techniques used by attackers to compromise Windows systems and some fundamental defense mechanisms against them. This paper will provide an overview of Microsoft's latest additions to the security concept of the Windows Operating System [OS] , analyse inherent flaws in their design and take a brief look at already existing attacks.

1 Introduction

In an increasingly digitalized world an overwhelming amount of private and/or safety-critical information and data is stored on computers. Windows is by far the most used operating system and therefore the main target of attackers to compromise data or computer systems. One common intent of attackers is to steal an individual's passcode, e.g for an online-banking website. Naturally, as the amount and complexity of attacks rises, OS vendors are forced to put an increasingly high amount of effort into mitigating existing weaknesses and deny attackers of further possibilities to compromise their OS. Even though this is the case, the amount of potentially abusable vulnerabilities in Windows has been increasing, instead of decreasing. [1]

Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	XSS	Bypass something	Gain Information	Gain Privileges
2015	57	4	19	6	6		10	5	26
2016	172	6	47	23	7		19	31	82
2017	268	32	50	16	2	1	18	108	19
2018	257	21	45	19	1	1	39	72	1
2019	357	28	124	101	6	1	10	73	2
Total	1111	91	285	165	22	3	96	289	130
% Of All		8.2	25.7	14.9	2.0	0.3	8.6	26.0	11.7

Figure 1: Amount of documented vulnerabilities in the Windows Operating System.
Note that the spike in "Gain Information" vulnerabilities in 2017 is inflated by a family of attacks widely known as Meltdown/Spectre

We can see that the three largest groups of vulnerabilities consist of "Gain Information", "Code Execution" and "Overflow". Of course these three categories aren't entirely separated from each other. An attacker that is able to execute code on a machine often does so in order to gain information and overflows are often the reason why an attacker can execute code in the first place.

2 Overflows

An overflow happens when a program writes data to memory beyond the limits of the intended data structure. One common example of this is a stack buffer overflow caused by an incorrect use of the function *strcpy*[1]. More precisely, an overflow can occur when the given input is longer than the buffer one writes too. Relatively small-in-size overflows are not always easy to spot and often remain unidentified if they don't cause immediate errors. Besides causing faulty program behaviour, this also provides a critical attack surface. To see why this is the case, let's take a look at a typical stack layout with only one buffer present right at the beginning.

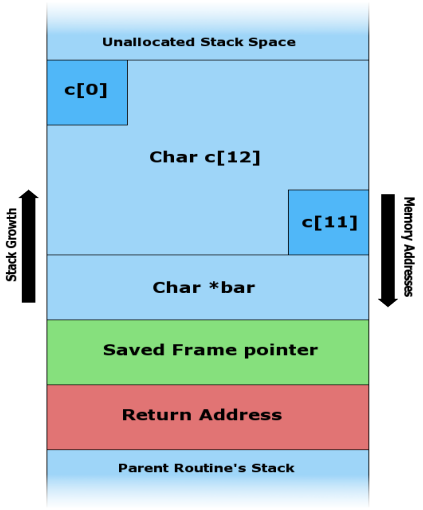

Figure 2: A typical stack layout

As we can see, the stack will usually contain a return address right at the bottom, a frame pointer on top of it and then local data that is used by the current function. Obviously, one will also need to keep the

[1] "char * strcpy (char * destination, const char * source)" is a C function that copies a string into the specified memory

parent function's data saved below (stack grows upwards). In our example there is only one character buffer of size 12 and a pointer to it present. Suppose one wants to copy user-input string into this buffer, e.g by using *strcpy*. The user may now unknowingly or perhaps purposely overflow this buffer by entering a string that's longer than 12 characters. As we observed earlier, this will result in a memory-write beyond the buffer's bounds.

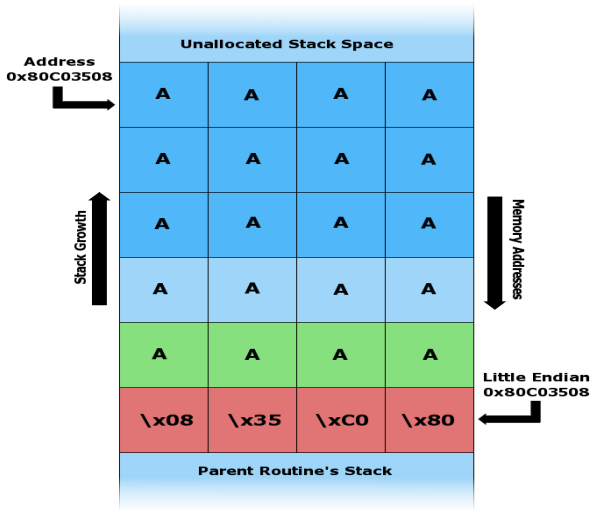

Figure 3: The user-input is too long for the given buffer

In this example we can see that our pointer to the buffer is overwritten, as well as the stack frame we had saved on the stack earlier. Most importantly though, the return address is also corrupted. In the given figure, the input is constructed so that the part that's written over the return address resembles an address itself. In this case, its just the address of the buffer again. At some point, our given function will attempt to return to this address. However, once it does so, all it will be able to read from this address is a swarm of 'A's which certainly doesn't resemble a valid sequence of code. The program will fault and terminate. This problem has existed for as long as the concept of the stack itself so naturally techniques to (try) prevent this from happening were implemented long ago.

3 Data Execution Prevention

One easy, and naive way to address the aforementioned issue is the use of Data Execution Prevention (DEP). DEP follows a very simple approach to prevent the execution of code from malicious addresses. Note that an attacker that uses overflows can usually only manipulate the stack which is typically used for non-executable data only (this is also true for heap overflows). Since we know that, we could easily mark the entire memory-area[2] (with an additional attribute for memory pages) that is used as a stack as *non-executable* or "NX" in short and that is basically what DEP does. Whenever an attempt to run code is made a check will ensure that the "NX"-Bit for the according page is not set.If it is, the program will terminate immediately. This approach offers an easy solution to prevent the *generation* of executable code on the stack. However, it does *not* prevent an attacker from redirecting control flow to already existing code. Attacks that abused this flaw could therefore still compromise or even take over a program.[3]

4 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) was implemented to address this issue. Since an attacker that uses pre-existing code needs to be able to tell where the code he wants to use is, an easy solution is to prevent them from addressing said code. ASLR does this by randomizing the arrangement of segments like the stack, heap and DLLs in memory.[4]

5 Return Oriented Programming

Return oriented programming describes one of the most popular approaches to manipulating programs by hijacking control-flow through the manipulation of return addresses. In essence, there are 3 steps to do :

1. Finding a security vulnerability that allows one to manipulate memory
2. Writing code (also called Payload or Shellcode[5] on the stack
3. Manipulating the return address to point to the injected code

Windows did not offer any kind of protection against ROP attacks until 2004 when DEP was introduced. However, as already mentioned, attackers quickly

[2]Sometimes code needs to be run directly from the stack, e.g just-in-time compiled JavaScript code, so this is a very simplified approach

[3]A well known way to do this is a return-to-libc attack

[4]32 bit Windows randomizes 8 of the address' bits, 64 Bit Windows can randomize a total of 19 bits.

[5]The name Shellcode comes from the fact that such attacks often inject code to open a Shell on the target system

overcame this barrier by using already existent code which would not be marked non-executable. One proposed "solution" to this was to store the first argument of each function in a register instead. Registers are not as easy to manipulate due to not being explicitly writeable in theory, though attackers easily overcame this restriction as well. Instead of using complete functions they would now use only small portions of a function that ended in a return instruction. Obviously, instruction sequences that allowed to manipulate registers were especially useful, in order to invoke complete functions again. However, this is not needed as the usage of such so called "gadgets" is already turing-complete given a *big enough* [2] program. This is made even easier by the fact that one can use instruction sequences that weren't supposed to be in the program to begin with. To see how this is possible, recall how machine code is written and read by the CPU. Note that in contrast to natural language, machine code doesn't include any white space (e.g spaces or slashes) so one may start reading wherever they want. The following figure shows an example where 2 instructions are split into 4 just by removing 1 Byte at the start.

```
1 F7 C7 07 00 00 00   -  test $0x00000007, %edi
2 0F 95 45 C3         -  setnzb -61(%ebp)
3
4 Missing the first Byte (F7) :
5
6 C7 07 00 00 00 0F   -  movl $0x0f000000, %edi
7 95                  -  xchg %ebp, %eax
8 45                  -  inc %ebp
9 C3                  -  ret
```

Figure 4: An example of an unintended use of machine code [3]

In theory ASLR should prevent such attacks completely since no attacker should be able to locate any specific code to begin with. In reality though vulnerabilities in both DEP and ASLR (even combined) have been found and successfully exploited. How this was done won't be covered here, but it's important to see that these techniques weren't sufficient protection after all.

6 Shadow Stack

The idea of a Shadow Stack follows a different approach to the issue. Instead of trying to prevent the manipulation of the return address an additional check is built in to make sure it wasn't altered. In order to do this the Shadow Stack saves all return addresses in a separate memory location. Every call instruction in a program pushes the return address of the parent routine on the normal stack *and* on the Shadow Stack. Every return instruction pops one address off the Shadow Stack and compares it to the address on the normal stack.If they're the same the program continues normally, if they differ a (Control Protection) Exception is raised and the program is terminated immediately.

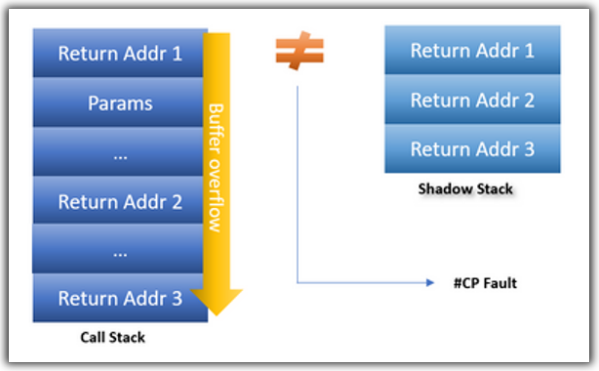

Figure 5: Shadow Stack Overview

In order to do this a new register called Shadow Stack Pointer(SSP) and an additional memory page attribute are used. Writes to the Shadow Stack are restricted to control transfer instructions(*call/ret)* and Shadow Stack management instructions for obvious reasons[6]. Note that this approach only provides integrity of "backward jumps" (also called Backward Edge)."Forward jumps" may still be manipulated while a Shadow Stack is being used. We will come back to this in a bit.

6.1 Shadow Stack revisited

One main problem with the Shadow Stack is that virtually no one is able to make use of it as of now. Since Microsoft's implementation of the Shadow Stack requires an additional register it is only usable with certain CPUs (more specifically all those that use a chipset that supports Intel Control-Flow-Enforcement technology). Intel's first generation of CPUs that use this chipset are yet to be released[7] and so are AMD's. ARM follows a different approach all together and thus won't be releasing any compatible CPUs. Additionally, once compatible CPUs are released it will still take a couple of years for them to be used widely. Considering that a software version of the Shadow Stack (dubbed Return Flow Guard) was already in development it

[6]There's no point in having a separate stack to save data when its explicitly writeable

[7]Intel Tiger Lake CPUs are announced for late 2020

is not quite apparent why this approach was not further pursued. According to unofficial sources [8] development of the RFG was discontinued because a vulnerability in the implementation was found. Surely something could've been done about this in the meantime though. An imperfect mitigation is better than no protection at all. One good aspect is that optimized Shadow Stacks have a runtime overhead of at most 5% and negligible memory overhead. Note that the Shadow Stack only validates return addresses. Function parameters are (intentionally) not protected. This is essentially a trade-off between security and runtime/memory overhead. It is non-trivial when or whether this trade-off is worth it and surely depends on the intended environment. According to Microsoft the use of Shadow Stacks will be opt-in "at first" per linker-flag for apps and DLLs for compatibility reasons.

7 Control Flow Guard

To take care of the Forward Edge the Control Flow Guard was shipped with Windows 8.1 Update 3. More specifically, CFG aims to prevent the misuse of indirect jumps. To do this developers can compile their programs with the CFG flag and ensure that their program will only run on memory areas that have been marked "safe" earlier. Here "safe" means that the address of every indirect jump refers to a valid function in the program. In order to check this efficiently at run time a bitmap of the starting addresses of every function is created at compile time where every bit in the bitmap corresponds to 8/16 [9] bytes in the address space. If a function starts within the block of addresses that correspond to a given bit it is set to 1, otherwise it is 0. A guard instruction is then inserted before every indirect jump which validates that the matching bit for the jump's target address in the bitmap is set to 1. If it is not, the program is terminated immediately. If a CFG compatible program is run on a Windows version that doesn't support CFG the call simply does nothing. To increase performance, system processes which are run as "protected" processes use shared bitmaps for DLLs. Note that dynamically generated functions cannot be protected this way as explicity allocated and "executable" marked memory has its corresponding bits in the bitmap implicitly set to 1. This is because dynamically allocated memory can not be considered at compile time.

[8] https://www.techrepublic.com/article/windows-10-security-how-the-shadow-stack-will-help-to-keep-the-hackers-at-bay

[9] 8 bytes per bit at first, later changed to tuple per 16 bytes

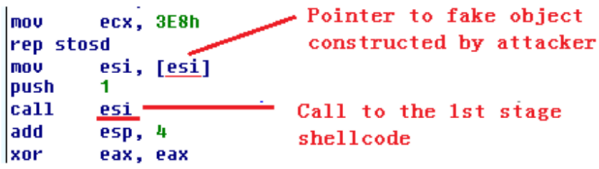

Figure 6: Example code without CFG [4]

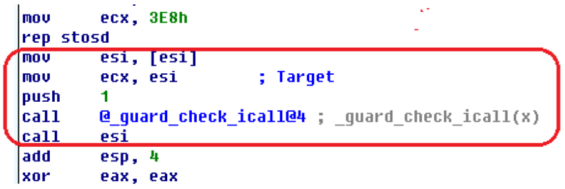

Figure 7: Example code with CFG enabled [4]

7.1 Control Flow Guard revisited

One major flaw in the initial design of CFG lies in one of its main assumptions: That the addresses of functions are aligned to 8 bytes. While this assumption holds true for the compiled program itself (since the compiler can enforce the alignment) it certainly is not for DLLs. To make up for this issue the CFG uses a mapping of 2 bits (a tuple) for every 16 bytes since Windows 10 instead where (0,0) means no function starts in the corresponding address block, (1,0) means a function starts at the start of the block and (1,1) means a function starts anywhere in the block. Note that this still provides a sort of margin for attacks when alignment is not enforced as an attacker can still manipulate a jump to point to anywhere within a (1,1) block

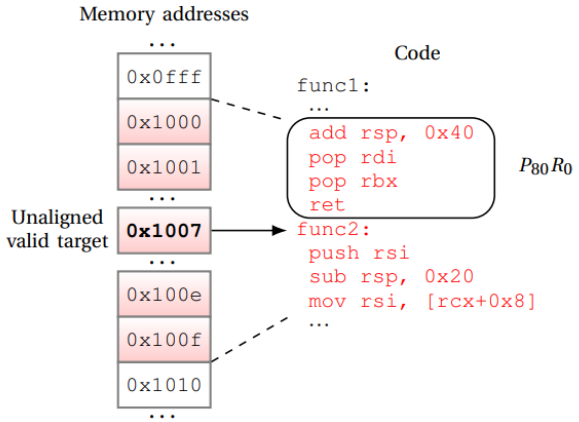

Figure 8: An example of why unaligned addresses can be a major vulnerability [4]

Another oversight that was found rather quickly: The guard function wasn't properly protected. In 2015 Zhang Yunhai showed that by using a (custom-)heap-allocation and freeing it afterwards one may make its memory area writeable again under certain circumstances. [5]
Last but not least another major issue that has not been fixed as of now: Lots of Windows system DLLs are not compiled to use CFG (145 in Windows 10 1511 / still 50+ in build 18362). [6]
Note that any program, even if it is compiled to use CFG, is vulnerable when it uses a DLL that is not since the guard calls are only present for the cfg compiled program. This means that as long as there are unprotected modules present every part of the program is vulnerable.

8 Approach

9 Conclusion

References

[1] https://www.cvedetails.com/product/32238/Microsoft Windows-10.html, "Windows 10 security vulnerabilities."

[2] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX conference on Offensive Technologies*, pp. 7–7, USENIX Association, 2012.

[3] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552–561, 2007.

[4] J. Tang and T. M. T. S. Team, "Exploring control flow guard in windows 10," *Trend Micro Blog*, 2015.

[5] Z. Yunhai, "Bypass control flow guard comprehensively," *Black Hat USA*, 2015.

[6] https://improsec.com/tech-blog/bypassing-control-flow-guard-on-windows-10-part ii, "Bypassing cfg on windows 10."