

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), analyze 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 on desktop PCs [1] and therefore a main target of attackers to compromise data or computer systems. A common intent of attackers is to steal an individual's private/confidential data or install ransomware to extort money. Naturally, as the amount and complexity of attacks rise, OS vendors are forced to put an increasingly high amount of effort into mitigating existing weaknesses and deny attackers further possibilities to compromise the OS. Even though this is the case, the amount of potentially abusable vulnerabilities in Windows has been increasing, instead of decreasing.

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 [2].

Note that the spike in *Gain Information* vulnerabilities in 2017 is inflated by a family of attacks widely known as Meltdown/Spectre.

In Figure 1, we can see that the three largest groups of vulnerabilities consist of *Gain Information*, *Code Execution* and *Overflow*. These three categories are not 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. We will take a quick look at how these issues play together and what has been done against them in the past before taking a deeper dive into Microsoft's newest defense additions.

2 Overflows

An overflow happens when a program writes data to memory beyond the limits of the underlying 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 to.

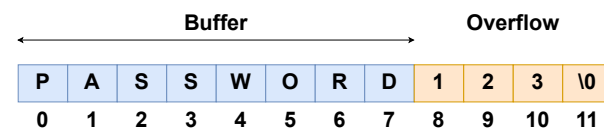


Figure 2: The password's length is too large to fit into the designated buffer. However, `strcpy()` will not check for valid sizes so it will continuously write beyond the buffer.

Relatively small-in-size overflows are not always easy to spot and often remain unidentified if they do not cause immediate errors. In addition to causing faulty program behavior, this also provides a critical attack surface. To see why this is the case let us take a look at a typical stack layout with only one buffer present right at the beginning.

As we can see in Figure 3 (A) 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. One will also need to keep the parent function's data saved below (stack grows upwards in this example). There is only one character buffer of size 12 and a pointer to it.

¹`char * strcpy (char * destination, const char * source)` is a C function that copies a string into the specified destination buffer

one can use instruction sequences that were not 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 does not include any white space (e.g., spaces or slashes) so one may start reading wherever they want. The following figure shows an example where two instructions are split into four just by removing one 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 $0xf000000, %edi
7 95                - xchg %ebp, %eax
8 45                - inc %ebp
9 C3                - ret

```

Figure 4: An example of an unintended use of machine code^[7]. Ignoring the first byte results in a sequence of four instructions instead of two.

5 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 they want 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. This means that using a fixed address to reference memory will most likely result in a segmentation error or the wrong code being targeted at best.⁵ However, a single pointer leak may compromise the location of any of the segments and thus break ASLR completely. (Note that this has happened quite often already)

In theory, ASLR should prevent ROP attacks completely since no attacker should be able to locate any specific code to begin with. In practice though, vulnerabilities in both DEP and ASLR (even combined) have been found and successfully exploited [8]. How this was done will not be covered in this report, but it is important to see that these techniques were not sufficient protection after all.

⁵32 bit Windows randomizes 8 of the address' bits, 64 Bit Windows can randomize a total of 19 bits.

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 was not 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.

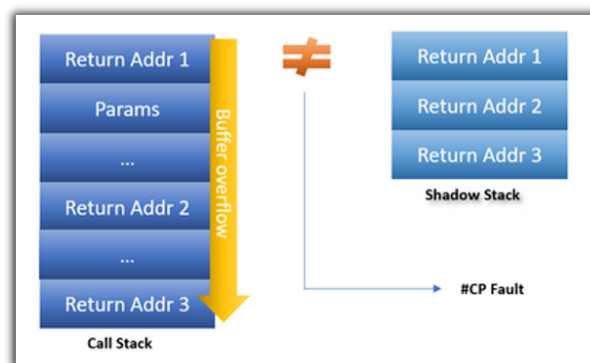


Figure 5: Shadow Stack Overview. Every return address on the stack is matched by a copy of that return address on the shadow stack.

In order to implement this check, 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.⁶ Note that calls to exception handlers and *longjumps* may cause issues as calls and returns are not necessarily perfectly matched if they occur [9]. One solution to this is to continuously pop addresses off the shadow stack until a matching address is found. In this case, a program would be terminated if the shadow stack runs empty without match. Implementing special treatment for these troublesome calls instead could increase runtime performance but include a higher memory overhead and increase complexity. Using a parallel shadow stack removes the necessity of considering mismatches between call/ret instructions. Parallel shadow stacks are placed at a fixed offset relative to the main stack and all return addresses are placed parallel to the return addresses on the main stack (hence the name parallel shadow stack).

⁶There is no point in having a separate stack to save data when its explicitly writable.

Parallel shadow stacks have drastically reduced performance overhead (generally less than half) as it is no longer necessary to maintain the SSP (i.e. copying it to/from memory) but sacrifice some security to achieve this result [10]. Note that shadow stacks only provide integrity of "backward jumps" (also called backward edges). "Forward jumps" may still be manipulated while a shadow stack is being used.

6.1 Shadow Stack Revisited

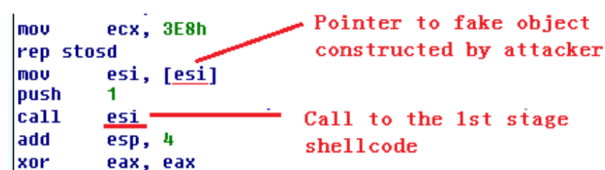
One main problem with the shadow stack is that no one (except for researchers and employees that have early access to hardware that is still in development) is able to make use of it as of now. Since Microsoft's implementation of the Shadow Stack requires an additional register it is only usable with certain CPUs (more specifically those using a chipset that supports Intel Control-Flow-Enforcement technology) [11]. Intel's first generation of CPUs that use this chipset are yet to be released⁷ and so are AMD's.

ARM will not be releasing any compatible CPUs [12]. 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 (RFG) [13] was already in development it is not quite apparent why this approach was not further pursued. According to unofficial sources⁸ development of the RFG was discontinued because a vulnerability in the implementation was found. Surely something could have been done about this in the meantime though. An imperfect mitigation is better than no protection after all. Performance overhead can range from as low as 3.5 (parallel) to roughly 10% (traditional shadow stacks) [10]. As Microsoft's implementation of the shadow stack will probably be neither parallel nor a basic traditional shadow stack we can expect the overhead to be anywhere in between these bounds. Note that the shadow stack only validates return addresses though. Function parameters are (intentionally) not protected. This is essentially a trade-off between security and run-time/memory overhead. It is non-trivial when or whether this trade-off is worth it and surely depends on the intended environment. Lastly, according to Microsoft the use of Shadow Stacks will be opt-in "at first" per linker-flag for apps and DLLs for compatibility reasons. This will further increase the time

needed for CFG to spread.

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 8/16⁹ 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 target address in the bitmap is set to 1. If it is not, the program is terminated immediately. If a CFG com-



```

mov     ecx, 3E8h
rep stosd
mov     esi, [esi]
push    1
call    esi
add     esp, 4
xor     eax, eax

```

Pointer to fake object constructed by attacker

Call to the 1st stage shellcode

Figure 6: Example code without CFG [14]



```

mov     ecx, 3E8h
rep stosd
mov     esi, [esi]
mov     ecx, esi ; Target
push    1
call    @_guard_check_icall@4 ; _guard_check_icall(x)
add     esp, 4
xor     eax, eax

```

Figure 7: Example code with CFG enabled [14]

patible program is run on a Windows version that does not 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 explicitly 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.

⁷Intel Tiger Lake CPUs are announced for late 2020

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

⁹8 bytes per bit at first, later changed to tuple per 16 bytes

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 eight 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 two 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 (15 bytes of unintended viable targets in the worst case!).

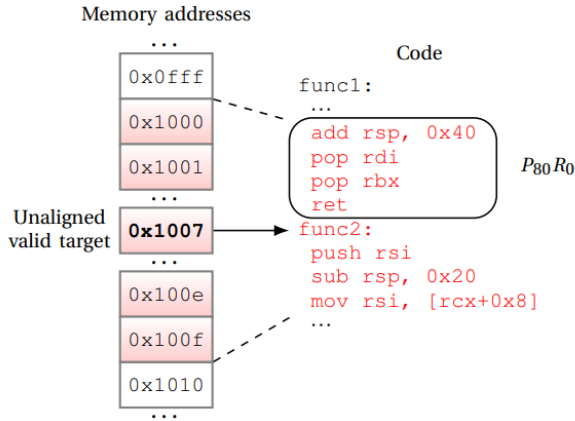


Figure 8: An example of why unaligned addresses can be a major vulnerability [14]

Another oversight that was found rather quickly: The guard function was not properly protected. In 2015 Zhang Yunhai showed that by using a (custom-)heap-allocation and freeing it afterwards one may make its memory area writable again under certain circumstances. [15] 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). [16] 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] Statcounter, “Desktop operating system market share worldwide.”
- [2] <https://www.cvedetails.com/product/32238/Microsoft-Windows-10.html>, “Windows 10 security vulnerabilities.”
- [3] S. Designer, “Getting around non-executable stack.”
- [4] Microsoft, “x64 calling convention.”
- [5] S. Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique.”
- [6] 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.
- [7] 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.
- [8] V. Katoch, “Bypassing aslr/dep.”
- [9] T. Zhang, “Shining light on shadow stacks.”
- [10] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566, 2015.
- [11] Intel, “Control-flow enforcement technology specification.”
- [12] M. Branscombe, “Windows 10 security: How the shadow stack will help to keep the hackers at bay.”
- [13] T. X. Lab, “Return flow guard.”
- [14] J. Tang and T. M. T. S. Team, “Exploring control flow guard in windows 10,” *Trend Micro Blog*, 2015.
- [15] Y. Zhang, “Bypass control flow guard comprehensively,” *Black Hat USA*, 2015.

- [16] [https://improsec.com/tech-blog/bypassing-control-flow-guard-on-windows-10-part](https://improsec.com/tech-blog/bypassing-control-flow-guard-on-windows-10-part-ii) ii,
“Bypassing cfg on windows 10.”