

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.

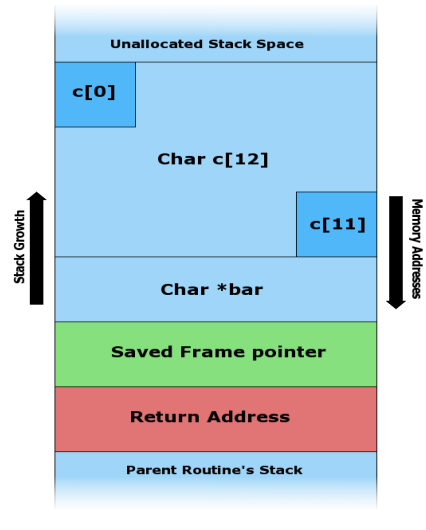
Note that the spike in "Gain Information" vulnerabilities in 2017 is inflated by a family of attacks widely known as Meltdown/Spectre

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.



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 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.

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

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.

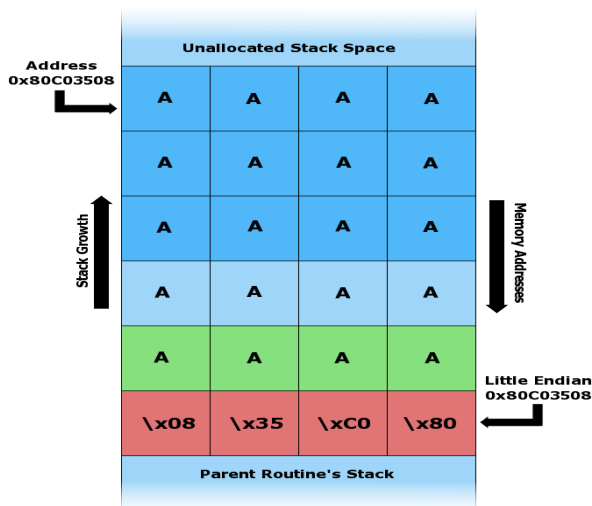


Figure 2: The user-input is too long for the given buffer

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, it's 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-area² (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 memory segments like the stack, heap and DLLs

5 Approach

6 Conclusion

References

- [1] cvedetails.com. Windows 10 security vulnerabilities.

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

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