Buffer Overflow Exploits and Alleviations

Sonu Sureshchandra Gupta

Pune, Maharashtra, India

sonugupta4636@gmail.com

***Abstract*— Over a decade ago, buffer overflow has caused immense security vulnerability and still continues. Just using few tools hackers are able to exploit the software applications and merge their attack code in applications. In this paper, I will be discussing buffer overflow exploits and various mitigation techniques for windows and Linux platforms. Apart from that, I will be discussing how we can prevent buffer overflow vulnerabilities by using combinations of different mitigation techniques while preserving the functionality and performance of the system.**

***Keywords*— Buffer Overflow, Security, Malicious, Prevention, Secure Code, Pointers**

I. INTRODUCTION

Buffer overflow is one of the most commonly found security vulnerability that allows attackers to give control of host machine. It gives the attacker liberty to inject and execute his attack code in user's application. This malicious code can now run with the privileges of user's vulnerable program and allows the attacker to bootstrap whatever functionality is needed to control the host machine.

Buffer overflow vulnerabilities and attacks come in various forms, which I will be describing in section II. Along with this, mitigation techniques against this exploit also comes in a variety of forms which I will be describing in section III in detail. Section IV discusses combinations of mitigation techniques in Windows and Linux platform. Section V presents the conclusion.

II. BUFFER OVERFLOW VULNERABILITY

A. Code in program's address space

The main crux of buffer overflow attack is to wreak havoc on the function of the privileged program. Doing this will give an attacker the control of that program. If the program is sufficiently privileged, it can further give access to the host machine.

Two sub-goals need to be achieved to do this.

- Arrange for suitable code to be available in program's address space.
- Get the program to jump to that code, with suitable parameters loaded into register and memory.

There are two ways to make the presence of attack code in program's address space i.e, either Inject it or use what is already there.

1) To Inject

The attacker provides a string to the program which acts as an input and which is stored in a buffer.The string contains native CPU instructions for the platform being attacked. This buffer can be located anywhere on the stack, on the heap or in the static data area. The attacker doesn't have to overflow any buffer to this; sufficient payload can be injected in perfectly reasonable buffers.

2) Already there

The malicious code of attacker is already present in the program address space. All attacker needs to do is parameterize the code and then cause the program to jump to it. This can be exemplified by the case when attacker executes "exec("/bin/sh")". Since there exists code in libc that executes "exec(arg)" where "arg" is a string pointer. The attacker only needs to change this pointer to point to "/bin/sh" and jump to the appropriate instructions in libs library.

The basic method to overflow a buffer that has weak or non-existent bounds checking. By overflowing the buffer, the attacker can easily overwrite the adjacent program state with shellcode/attack code.

- Activation Record:

An activation record is a just data structure having information about called function and return address. Each time the function is called, the activation record is placed on the stack.By corrupting this return address in activation record, attacker causes the program to jump to attack code.

- Function pointers:

These can be allocated anywhere(stack, heap, static area). The attacker only needs to find an overflowable buffer adjacent to a function pointer. Overflowing that buffer results in changing the function pointer. When sometime later, this function pointer is called, it will result in jumping to attacker's desired location.

- Long jump buffers:

'C' includes a simple checkpoint/rollback system called setjmp/longjmp. The idiom is to say "setjmp(buffer)" to the checkpoint and say "longjmp(buffer)" to go back to the checkpoint. However, if the attacker can corrupt the state of the buffer, then "longjmp(buffer)" will jump to the attacker's code instead. Since function pointers, longjmp buffers can be allocated anywhere, this gives the attacker liberty to find an adjacent overflowable buffer.

III. MITIGATING BUFFER OVERFLOW EXPLOITS

There are plenty of techniques that can be put in place by the developer such as secure coding practices, stack cookies, SafeSEH, etc. Most compilers and linkers nowadays enable

most of those features by default (except for "secure coding", which is not a feature of course), and that is a good thing. Unfortunately, there are still a horrible amount of applications that are not protected and will rely on other protection mechanisms. And I think you will agree that there are still a lot of developers who don't apply secure coding principles to all their code. They rely on OS protection mechanisms (see next), and just don't even care about secure coding.

Luckily for the zillions, Windows end-users, a number of protection mechanisms have been built-in into the Windows Operating systems implicitly such as:

– Stack cookies (/GS Switch cookie)

– SafeSEH (/SafeSEH compiler switch)

– Data Execution Prevention (DEP) (software and hardware based)

– Address Space Layout Randomization (ASLR)

A. Stack cookies (/GS Switch cookie)

When an application starts, a program-wide master cookie which is 4 bytes (Dword), unsigned int is calculated. This cookie can be any pseudo-random number and it is saved in the '.data' section of the loaded module. In the function prolog, this program-wide master cookie is copied to the stack, right before the saved EBP and EIP. (between the local variables and the return addresses)

During the epilog, this cookie is compared again with the program-wide master cookie. If it is different, it concludes that corruption has occurred, and the program is terminated.

Also, /GS is responsible for variable reordering. This variable reordering will prevent attackers from overwriting local variables or arguments used by the function, the compiler will rearrange the layout of the stack frame and will put string buffers at a higher address than all other variables. Thus, when a string buffer overflow occurs, it cannot overwrite any other local variables.

B. SafeSEH

Instead of protecting the stack (by putting a cookie before the return address), modules compiled with this flag will include a list of all known addresses that can be used as exception handler functions. If an exception occurs, the application will check if the address in the SEH chain records belongs to the list of "known" functions, if the address belongs to a module that was compiled with SafeSEH. Otherwise, the application will be terminated without jumping to the corrupted handler.

C. Data Execution Prevention (DEP)

Attack code is placed somewhere on the stack and then attempted to force the application to jump to attack code and execute it. Hardware DEP (or Data Execution Prevention) aims are preventing just that. It imposes non-executable pages (basically marks the stack/part of the stack as non-executable), thus preventing the execution of arbitrary shellcode.

Wikipedia cites "DEP runs in two modes: hardware-enforced DEP for CPUs that can mark memory pages as nonexecutable (NX bit), and software-enforced DEP with a limited prevention for CPUs that do not have hardware support. Software-enforced DEP does not protect from the execution of code in data pages, but instead from another type of attack (SEH overwrite). "[2]

In other words: Software DEP = SafeSEH Software DEP has nothing to do with the NX bit at all!

The concept of NX protection is pretty simple. If the hardware supports NX, the BIOS is configured to enable NX, and if the OS supports it, at least the system services will be protected. Depending on the DEP settings, apps could be protected too. Compilers such as Visual Studio C++ offer a link flag (/NXCOMPAT) that will enable applications for DEP protection.

D. Address Space Layout Randomization (ASLR)

Windows Vista, 2008 server, and Windows 7 offer yet another built-in security technique (not new, but new for the Windows OS), which randomizes the base addresses of executables, DLL's, stack and heap in a process's address space (in fact, it will load the system images into 1 out of 256 random slots, it will randomize the stack for each thread, and it will randomize the heap as well). This technique is known as ASLR (Address Space Layout Randomization).

The addresses changes on each boot. ASLR is by default enabled for system images (excluding IE7), and for non-system images if they were linked with the /DYNAMICBASE link option.

IV. Effective Combinations

When /DYNAMICBASE is enabled, a module's load address is randomized, which means that it cannot easily be used in Return Oriented Programming (ROP) attacks. When it comes to Windows applications, we recommend that all vendors use both DEP and ASLR, as well as the other mitigations outlined in the Windows ISV Software Security Defenses document.

On the Linux platform, ASLR does have a performance penalty. This penalty is greatest on the x86 architecture, and it is most noticeable in benchmarks. For an executable to be compatible with ASLR on Linux platform, it must be compiled with the Position Independent Executable (PIE) option.

The main goal of ASLR is to have executable code at an unpredictable address. But, there is a difference between the Windows and Linux implementations. It's important to note that ASLR compatibility on Windows is a link-time option, while on Linux it's a compile-time option.[4]

With Windows, the code is patched at runtime for relocation purposes. In the Linux and Unix worlds, this technique is known as text relocation. With Linux, ASLR is achieved in a different way. Rather than patching the code at runtime, the code is compiled in a way that makes it position

independent i.e, it can be loaded at any memory address and still function properly.

At least on the x86 platform, this position-independent capability is accomplished through the use of a general-purpose CPU register. But, even with one less register available to use by a program, it doesn't operate as efficiently. This limitation is most noticeable on architectures with a small number of registers, such as x86.

Why did the Linux developers choose this technique for implementing ASLR? Because text relocations involve patching. Thus, loading such a module would trigger copy-on-write, which subsequently increases the memory footprint of a system. But, the position-independent code does not require patching and therefore does not trigger copy-on-write.

Compiling your application with compiler flags which enables Stack cookies, safeSEH, DEP and ASLR will definitely make your application less vulnerable and difficult for a hacker to exploit and inject his attack code. Though these flags are here, the best practice of secure coding will definitely make the application more secure and less vulnerable to other exploits too.

V. Conclusions

I have presented a detailed categorization and analysis of buffer overflow vulnerabilities, attacks, and defenses. Buffer overflows are worthy of this degree of analysis because they constitute a majority of security vulnerability issues.

References

[1] Rafel Wojtczuk. Defeating Solar Designer Non-Executable Stack Patch. Bugtraq mailing list, http://geek-girl.com/bugtraq/, January 30, 1998.

[2] https://wiki2.org/en/Data_Execution_Prevention

[3] Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. http://www.cse.ogi.edu/DISC/projects/immunix.

[4] https://insights.sei.cmu.edu/cert/2014/02/differences-between-aslr-on-windows-and-linux.html