Stack-Based Buffer Overflow Attack

Austin Wolfe
CIT-485-001
April 10, 2021

Introduction

Listed in, “2020 CWE Top 25 Most Dangerous Software Weaknesses”, on cwe.mitre.org, is the, “Out-of-bounds Write [3],” category. Listed under this category is stack-based buffer overflow. Stack-based buffer overflow occurs when a buffer allocated on the stack is being overwritten. The buffer overflow vulnerability is a classic memory attack showing up throughout the history of computer security:

> The Morris worm in 1998 took advantage of a buffer overflow problem in the finger application, aiding the worm’s attack on remote systems [6]. The Code Red Worm in 2001 took advantage of unknown buffer overflow in a tool called ISAPI used by Microsoft’s Internet Information Services (ISS) version 4 and 5, which was the access point used for the worm which infected more than 250,000 systems in nine hours [7]. SQL Slammer in 2003 exploited a buffer overflow problem in the MS-SQL monitor service leading to a drop in CPU utilization on infected systems, hindering performance [8]. In 2015, seven bugs in the Android operating system (version 2.2) called Stagefright acted as “backdoors for remote code execution and privilege escalation [11]”.

The topic of this paper is about classic stack-based buffer overflow, with the intention to help understand a classic attack that still can happen today – especially with older or outdated devices – and to gain the knowledge needed before learning even more advanced topics on buffer overflow in the future.

A buffer is “a contiguous block of computer memory that holds multiple instances of the same data type [2]”. To overflow a buffer means to write more to the buffer than it’s intended initialized size, overwriting past the bounds of the buffer and into contiguous areas of memory. The exploit of a buffer overflow condition can lead to memory corruption which can cause software to crash or allow the attacker to inject and execute some code that they choose. C and C++ are the languages most common to buffer overflow vulnerabilities, so the example code in this paper will be mostly in C, and the buffer conditions will be most commonly in the form of buffer arrays. We will look at programs allocated in stack memory, witnessing what can potentially happen to programs compiled containing a buffer overflow condition.

This paper examines the memory structure that allows for buffer overflows – particularly in the stack memory region – and examines scenarios and C functions that lead to these vulnerabilities in programs. We also examine how to exploit the buffer overflow vulnerability to execute code. We do this by finding the data needed to create a shellcode designed to return a root owned shell to the attacker (we won’t write out all the shellcode in this paper), then insert the shellcode instructions into memory during the buffer overflow. Lastly, we look at coding, operating system, and compiler countermeasures that are implemented to stop these buffer overflow vulnerabilities, while also looking at ways for attackers to get around these protective countermeasures.

Memory Layout

The way buffer overflow attacks work can only be fully understood by first examining how a process's data is organized in memory. C programs are divided into five segments: text segment, data segment, BSS segment, heap, and stack.

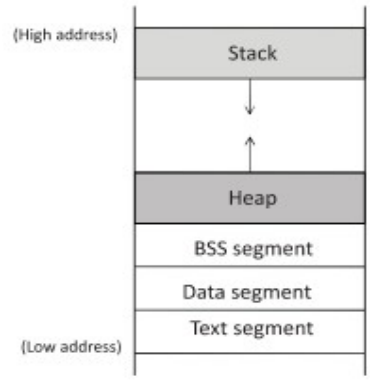


Figure 1: Program Memory Layout

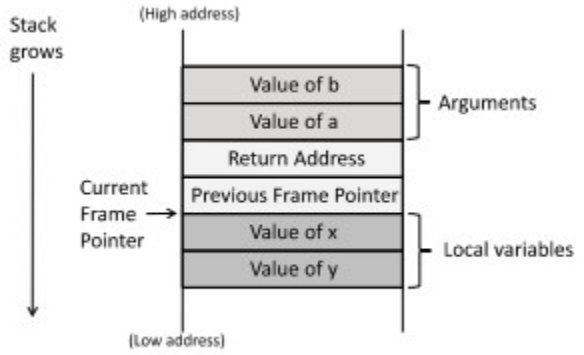
Source: [1]

Descriptions of the segments in Figure 1 are as follows: The text segment is usually read only and contains the binary executable code of the program. The data segment stores static/global variables initialized by the programmer. The BSS segment stores uninitialized static and global variables with zeros being stored for the uninitialized variables by the operating system. The heap is used as a space for allocating dynamic memory, and the stack is used for storing the local variables inside functions and the data that pertains to that function – like return addresses and arguments. The stack segment will be a major focus of this paper.

The Stack

The stack – a static location where programming instructions and data are stored – is used to determine what instructions a program should run, and when they should run. **Functions** are pushed onto the stack in **stack frames** when they are called, then popped off of the stack on returns. A stack frame holds the function parameters, local variables, and the data telling how to return to the previous stack frame [2]. In the Figure 1 example, it should be noted that the stack is shown growing down towards lower addresses (this is the way the stack grows on computers with Intel processors). This means that the growing part of the stack is at lower memory addresses. Since the stack is growing down, the bottom part – the bottom part being a fixed address that is dynamically adjusted by the kernel at run-time [2] – exists at lower memory addresses. The stack grows or shrinks when the CPU uses instructions to "push" and "pop" data onto and off the stack in a last in first out (LIFO) queue.

To demonstrate the layout of a function's stack frame, Figure 2 is provided, followed by the code snippet to which it corresponds.




```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

Figure 2: Function's stack frame layout

Source: [1]

When the **func()** function is called, the memory space for the stack frame is allocated onto the stack with regions for the arguments, return address (containing the address of the location in which to return after the function finishes), previous frame pointer, and the local variables. These memory segments are pushed onto the stack frame in that order.

The previous frame pointer section on the stack holds the previous frame address (the one before the current function call), and this value comes from being stored in the frame pointer register, which points to a fixed point (an address) on the stack frame. The reason the frame pointer register points to a fixed place on the stack is because compilers cannot predict the runtime status of the stack, and since memory addresses are needed to be known to access the data inside of and pertaining to functions, the frame pointer register points to a fixed location and is used along with an offset to calculate the arguments and the local variable addresses of functions [1]. In x86 (32 bit) architecture, the frame pointer register is called **ebp**. The **ebp** register always points to the stack frame of the current function, but it allocates the previous stack frame into the previous frame pointer segment on the stack when jumping to another function, so on return from a function, the previous stack frame is not lost. There is also the **%esp** register (stack pointer) which points to the top of the stack.

Stack Buffer-Overflow Attack

The point of a buffer overflow attack is to write past the memory allocated on the stack that is reserved for a buffer, therefore overwriting higher memory addresses with the intent to rewrite these stack segments with our own code. In Figure 2, where you see Return Address, using the stack buffer overflow attack we can change that address to one that points to the address of the attacker's malicious code.

Susceptible C Functions

There are some C standard library functions that allow for the copying of too much memory into a buffer than what is intended by the programmer. The programmer may have intended the buffer to be a smaller byte size, but C functions like **strcpy**, **sprintf**, **strcat**, and **gets** – which copies data from one location to another [1] – are susceptible to buffer overflow attacks. The problem lies in how these functions know to stop copying data, which is by using certain unique characters. One example is the use of the NULL character '\0' in **strcpy()**. Since the length of the data is controlled by some unique character and not a specified maximum byte size (buffer length), the data that is being copied can be altered in a way that allows even bigger sized data to be copied, therefore allowing for overflow on the stack. In the buffer overflow examples in this paper, **strcpy()** will be used to copy data into a buffer array.

```
char *strcpy(char * destination, const char * source);
```

We now will see an example of how an attacker would create a buffer overflow in **strcpy()** without injecting our own malicious code (we will look at how to do that in the next section). In the following example, the pre-built VM (discussed in [1]) is used (Ubuntu, Version 16.04.2, 32-bit). The compiler is GCC version 5.4.0.

The following program, called **stack.c** [1], is a Set-UID root program that contains a buffer overflow vulnerability. The goal of exploiting the Set-UID root program's buffer overflow vulnerability is so that malicious code can be injected and ran with root privileges. The code to be injected into this program – called shellcode – will run a root shell. The program is compiled with gcc to the executable binary called **stack.** We turn off current buffer overflow countermeasures on this VM for the program (will be discussed in more detail in countermeasures section), and then the ownership of the compiled program is changed and made into a root owned Set-UID program.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");

    /* the first arg in fread is a pointer to a buffer to store what
       is being read in from the file */
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

This program reads 300 bytes from the binary file **badfile** by using the **fread()** function and stores it at the **str** pointer. The pointer address (**str**) is then supplied to the **foo()** function, and then the array starting at this pointer address is copied into **foo**'s local buffer array called **buffer** by using the **strcpy()** function. If the number of bytes read to this array is of size 100 or less, then the program runs normal, but copying in the 300 bytes causes buffer overflow.

Putting malicious code onto the stack

Since the source code **stack.c** is known, and we know it is a root owned Set-UID program, it is easier to exploit. Within this experiment, in order to write some shellcode, it is better to determine where in memory the function **foo()** – which is the function containing the buffer overflow condition – is allocated**,** so that the shellcode can be written to correspond to the correct memory addresses.

Compiling **stack.c** with the debugger flag turned on allows us to use gdb to set a break point at the **foo()** function with the **b** command. We then use the **run** command to run and then stop the program inside the function **foo()** [1, p. 76]. Once there, the **p** command can be used to obtain the base pointer (**ebp**) address and the address of the buffer array called **buffer**. We can see in the stack diagram of the program **stack,** in Figure 3, just where the data of the **foo()** function is allocated.

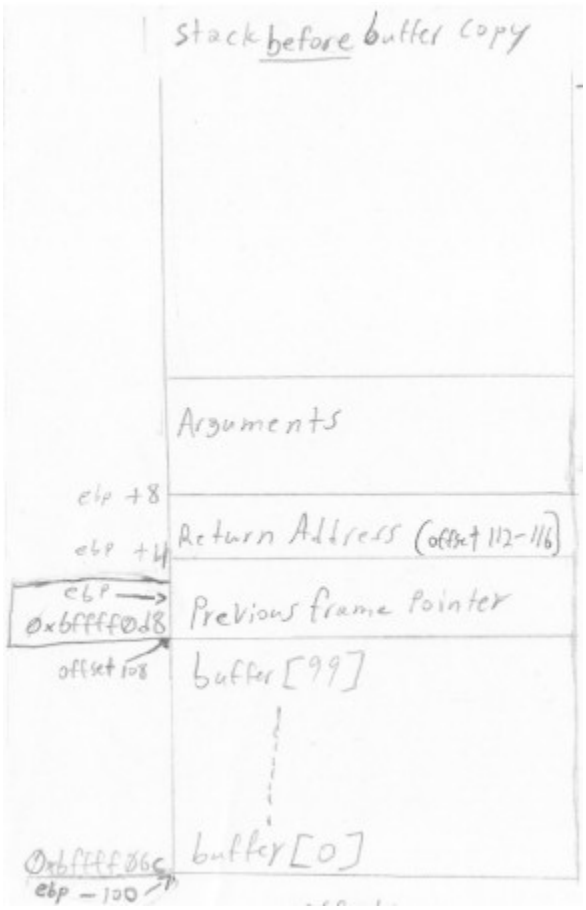


Figure 3: Stack before buffer copy

Source: Adapted from [1]

The previous frame pointer segment and the return address segment is 4 bytes each, and the address of ebp is known; it is now easy to find the offsets for the return address and the arguments segment, which is the first address that can be jumped to in order to reach where the malicious code will get stored. The distance between the beginning of the buffer and the return address is also important for knowing at what point in the shellcode to store the new return address (the address pointing to the malicious code). This is easy to know, since all that needs to be done is to subtract the buffer address from the ebp address and then add 4 to get the distance (distance = 0xbffff0d8-0xbffff06c + 4 = 112). With these addresses now known, the shellcode can be constructed. The Idea behind the shellcode is to overwrite the stack at particular areas when **badfile** is read into the buffer to run the malicious code. The problem of getting to the malicious code can be made easier with NOP instructions that advance the program counter to the next location. All the return address needs to do is hit one of the NOP instructions and the malicious code will eventually execute, thus creating multiple entry points for the injected malicious code. Figure 5 compares the stack after the buffer copy, with and without NOP instructions.

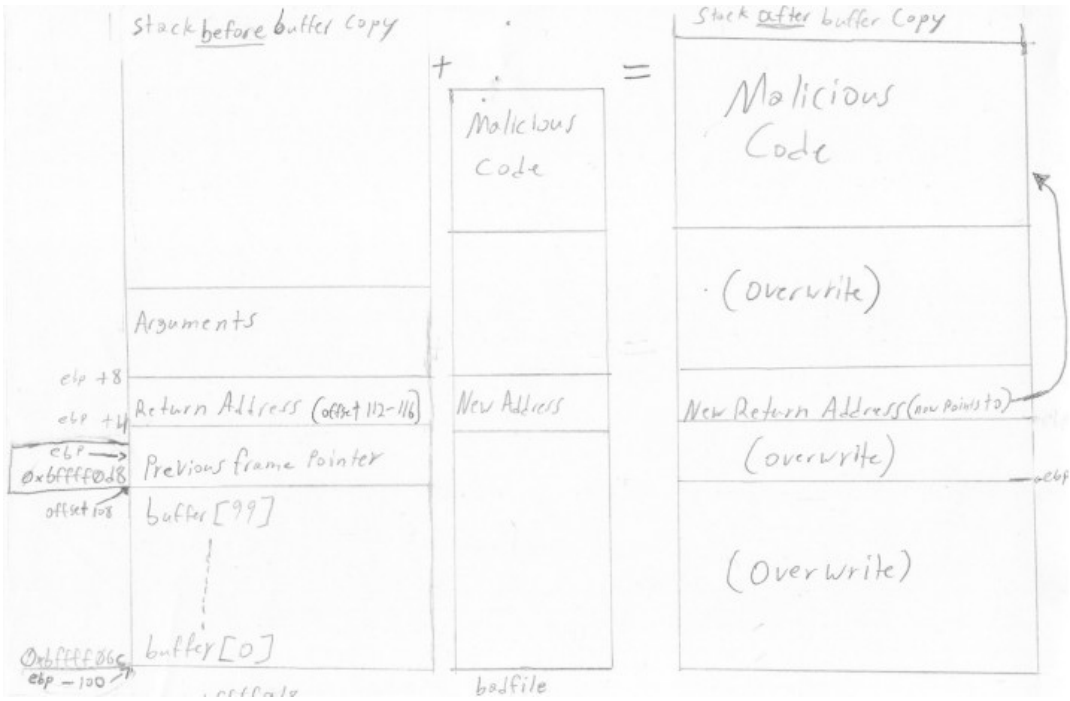


Figure 4: Before and after buffer copy

Source: adapted from [1]

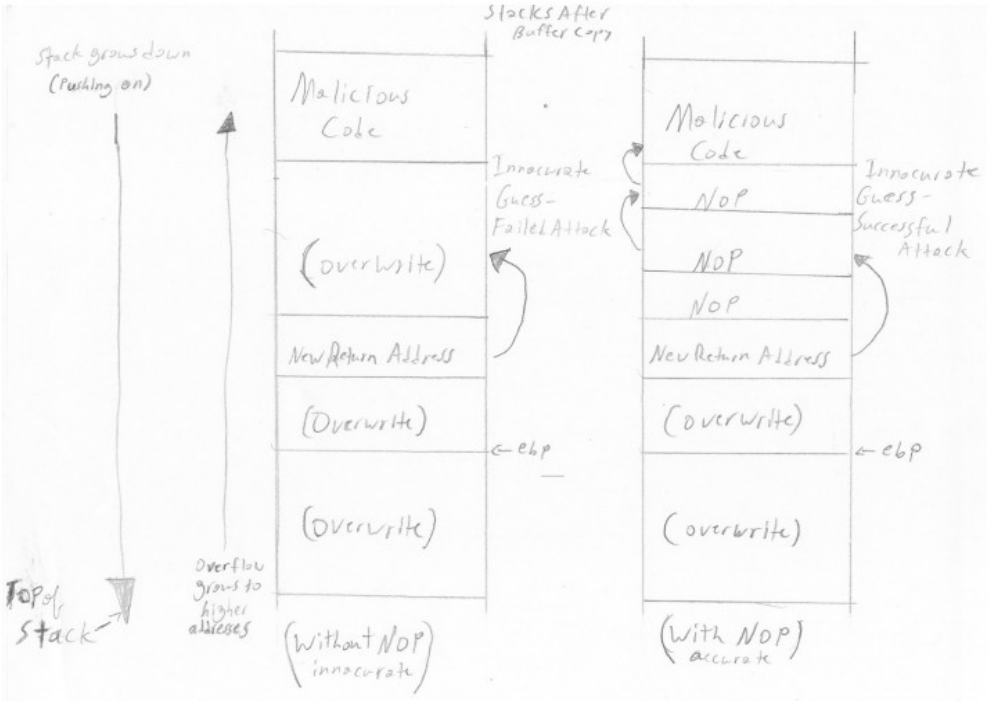


Figure 5: Stack with and without NOP instructions

Source: Adapted from [1].

Countermeasures

Tools and techniques have been created to counteract stack-based buffer overflow attacks. These can be categorized into these groups: static-analysis, compiler modifications, operating system modifications, and hardware modifications [10]. These categories are often combined to help prevent buffer overflow attacks due to attackers creating exploitative countermeasures for the defensive countermeasures that were created over time.

Checking Source Code for Vulnerabilities

One way to prevent buffer overflow attacks is to make sure that there are no buffer overflow conditions in the source code. One way of doing this is to use programming languages that check for buffer overflow, like Python and Java, which have automatic boundary checking [1][10]. But because C is used when low level data manipulation is needed, a good way for uncovering software flaws mentioned in [10], is to do source code review.

In [13], a secure code review is said to be meant for identifying “specific security related flaws within the code that a malicious user could leverage to compromise confidentiality, integrity, and availability of the application.” And mentioned as an example of this type of security flaw is a buffer input that has not been checked, leading to buffer overflow that could “allow a malicious user to execute arbitrary code with escalated privileges”. In [13], it is also mentioned that, though secure source code reviews may not find all flaws, “it should arm developers with information to help make the application’s source code more sound and secure,” and that it is no “silver bullet,” but a “strong part of an overall risk mitigation program to protect an application”.

An example of something simple to look for during a source code review of C code is unsafe memory copy functions; this is because functions like **strcpy**, **strcat**, **sprintf**, and **gets** don’t require programmers to specify the maximum length of the data that they are working with [1]. Instead, the safer versions of these functions should be used: **strncpy**, **strncat**, **snprintf**, and **fgets**. These safer functions don’t make it impossible for buffer overflow – “If a developer specifies a length that is larger than the actual size of the buffer, there will still be a buffer overflow vulnerability” [1,p. 86] – but using these functions does make them less likely.

Libsafe

Libsafe helps with the previously mentioned unsafe functions. It is a dynamically loaded library that gets preloaded along with processes that it needs to protect [14]. During the preloading, the libsafe library is “injected between the program code and the dynamically loadable standard C library functions”, and because of this is able to “intercept and bounds-check” arguments in those unsafe functions before letting them execute. Libsafe can be bypassed because “It gives a limited amount of protection

against attacks[13, p. 110]" – monitoring only the unsafe functions, but not detecting buffer overflow "if a string is copied byte by byte".

Address Layout Randomization (ASLR)

For attackers to exploit a buffer overflow on the stack, they need to know the location of the stack in memory – this is so that they can more accurately guess where to tell a program to jump to – to run their injected malicious code. ASLR is used at the operating system's loader program [1]. For older devices not using ASLR, the stack was placed at a fixed address, making it easier for attackers to guess the addresses they needed to know to get the program to return to their injected code. But, the stack doesn't have to be at a fixed address: When source code is compiled into binary code, the addresses for all the data on the stack are not hard-coded into the binary[1]. These addresses are actually represented based on the offset to the frame pointer (%ebp) or the stack pointer (%esp) – not the start of the stack – so the stack can start at any place in memory as long as the %ebp and %esp registers hold the correct values. ASLR takes advantage of this and randomizes the position in memory of not only the stack, but the heap and libraries used by a program. This makes it harder for attackers to know the address of their malicious code in memory [12].

There have been techniques developed to counter ASLR: using non-ASLR modules, partial %eip overwrite, and brute force [12]. Using non-ASLR modules can cause the countermeasure to not work properly in the case where processes that use ASLR load in non-ASLR modules that allow an attacker to run shellcode with the **jmp %esp** command. In a partial %eip overwrite, part or the eip register can be overwritten or "use trustworthy information disclosure in the stack" [12, p.109] to find what the real %eip is, therefore using it to calculate a precise location (this technique also relies on non-ASLR modules).

The brute force technique is made possible by measuring the available randomness in an address space, and one way to measure this is through entropy [1, p. 90]. A memory region's base address can take 2^n possibilities for locations, all equally probable. On 32-bit Linux devices using static ASLR, stacks have 19 bits of entropy, meaning the base address of the stack can have 2^19 = 524,288 possible locations. This number is low enough to be exhausted quickly using brute force. This can be done by writing a script to repeatedly run the buffer overflow attack until the memory address is correct, causing the attackers malicious code to run.

Stackshield and StackGuard

Since compilers turn source code into binary, they therefore control the layout of the stack. It is because of this that compilers can insert countermeasures into the binary that check for or eliminate some of the steps an attacker would implement in a buffer overflow attack [1]. Two of these countermeasures are called Stackshield and StackGuard, and they both check if the return address has been changed before a function returns.

Stackshield copies the return address into some other place – a shadow stack – so it can't be overflown [1]: The compiler puts instructions into the binary that copies the return address into the shadow stack so that a comparison can be made between the return address saved in the safe place and the one on the stack before the function returns. This comparison is done to see if there is a difference that would indicate buffer overflow. Stackshield does not protect %ebp, and since %ebp is copied to %esp after a function return, can be used to exploit programs [12].

Another compiler technique that checks if the function return address has been changed is called StackGuard. StackGuard places a special value, called a guard – or "canary"[12] – onto the stack between the return address and local variables (where a buffer would be located). This is used to detect when a return address is overwritten [1].

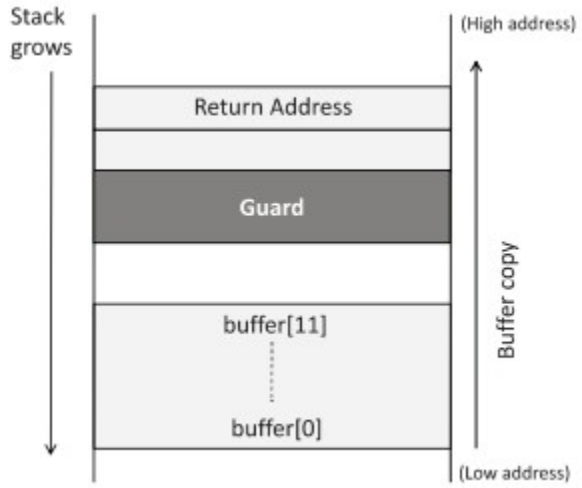


Figure 6: "The idea of StackGuard"

Source: [1, p.24].

The guard is a non-predictable value, and before a function returns, the value is checked to see if it has been changed. If the guard has been changed, it is a signal that the return address might be overwritten; although, that doesn't necessarily mean that the return addresses value has been modified.

Since the compiler is adding the code to a function, it is important for that code to be written so it can be added to and protect any function. The following is an example of how a compiler may add this code, which is a guard, to a function. Since the following is just example code implemented by a human and not by a compiler, we don't know for sure how close the variable defined first in the code will be to the return address. This is because the order in which variables are allocated to the stack is decided by the compiler [1].

```
void foo (char *str)
{

   char buffer[12];
   strcpy (buffer, str);

   return;
}
```

Figure 7:Function Before Adding the Guard

Source: [1].

Figure 7 shows the function before the guard code is added and Figure 8 shows the same function with the guard code added.

```
// This global variable will be initialized with a random
// number in the main function.
int secret;

void foo (char *str)
{
   int guard;
   guard = secret;

   char buffer[12];
   strcpy (buffer, str);

   if (guard == secret)
      return;
   else
      exit(1);
}
```

Figure 8:Function After Adding the Guard

Source: [1].

The **guard** variable is initialized with a secret, which is a random number. The random number is generated inside a main function (not shown here), so that the secret number is different each time the program runs. Now, when there is overflow, and as long as an attacker does not know the guard's secret value, we know that if there was any kind of modification to the return address, then the guard will also have to be changed [12]. The only way to mod the return address without the guard giving it away is to overwrite the guard with the same value. This is why the secret number must not be guessable. The secret value should never be hard-coded into the code or placed on the stack where it can be overwritten; the secret can be stored in the heap and BSS memory segments. In Figure 8, the uninitialized global variable, **secret**, gets put in the BSS segment.

Non-executable Stack

CPUs part of the modern hardware architecture use something called the NX bit – which stands for No-execute – that is used to separate code from data [1]. Operating systems can use this to mark areas in memory as non-executable, which means code in an area like stack memory will not execute. When a stack is made non-executable, performing a stack-based buffer overflow attack to run malicious injected code – like the examples used in this paper – will no longer work. There is a countermeasure to a non-executable stack, called a **return-to-libc** attack, but is not discussed here. More information on this topic can be found in [1], in chapter 5.

Conclusion

Stack-based buffer overflows occur when a function allocated onto the stack containing a buffer with unchecked boundaries is overwritten. Overwriting contiguous memory spaces on the stack can cause a program to crash or allow an attacker to change a function's return address to redirect the flow of the program to execute the attackers injected malicious code. This type of attack can allow an attacker access to a shell with escalated privileges on a victim's computer. We have seen countermeasures to the stack-based buffer overflow attacks and some ways around those countermeasure – like the brute-force technique to get around 32-bit memory address randomization.

The stack-based buffer overflow attack was one of the most successful software attacks for a long time, particularly because it is easy to make coding mistakes [1], creating susceptible software. The creation and implementation of counter measures onto computers throughout time has helped combat these attacks; though, even now there are still exploits for memory corruption leading to code execution. The modern CPU has a built-in countermeasure that sets the stack to be non-executable by default, which prevents injected malicious code from running, but even this countermeasure can be defeated with a **return-to-libc** attack [1]. Not all machines in the world are up to date, and safe coding practices still need to be a priority, especially when using C and C++. This category of attack is listed even now on cwe.mitre.org as one of the most dangerous software attacks.

References

[1] Du, Wenliang. “Buffer Overflow Attack.” *Computer & Internet Security: A Hands-on Approach*, 2nd ed., Wenliang Du, 2019, pp. 63–98, https://www.handsonsecurity.net/files/chapters/buffer_overflow.pdf. PDF file.

[2] Aleph One. (n.d.). College of Engineering and Computer Science. http://cecs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html.

[3] *Common Weakness Enumeration*. CWE. (n.d.). https://cwe.mitre.org/data/definitions/1350.html.

[5] *Common Weakness Enumeration*. CWE. (n.d.). https://cwe.mitre.org/data/definitions/121.html.

[6] Orman, H. (n.d.). *The Morris Worm: A Fifteen-Year Perspective*. www.cs.umd.edu. https://www.cs.umd.edu/class/fall2019/cmsc818O/papers/morris-worm.pdf.

[7] Berghel, Hal. “The Code Red Worm.” *Doi-Org.Northernkentuckyuniversity.Idm.Oclc.Org*, Association for Computing Machinery, Dec. 2001, doi-org.northernkentuckyuniversity.idm.oclc.org/10.1145/501317.501328.

[8] “The MS-SQL Slammer Worm.” *ScienceDirect*, 1 Mar. 2003, linkinghub.elsevier.com/retrieve/pii/S1353485803003106.

[10] Kuperman, Benjamin A., et al. “Detection and Prevention of Stack Buffer Overflow Attacks.” *Communications of the ACM*, vol. 48, no. 11, Nov. 2005, pp. 51–56. *EBSCOhost*, search.ebscohost.com/login.aspx?direct=true&db=edb&AN=18740803&site=eds-live.

[11] Akcay, Fatma Kevser. “Android’s Performance Anxiety: An In-Depth Analysis of the Stagefright Bugs.” *Dash.Harvard.Edu*, 26 Mar. 2019, dash.harvard.edu/handle/1/38811442.

[12] SOLANKI, J., SHAH, A., & MANIK LAL DAS. (2014). Secure Patrol: Patrolling Against Buffer Overflow Exploits. *Information Security Journal (Print)*, *23*(1–3), 107–117.

[13] “Secure Code Review.” *The MITRE Corporation*, 10 Apr. 2015, www.mitre.org/publications/systems-engineering-guide/enterprise-engineering/systems-engineering-for-mission-assurance/secure-code-review.

[14] Baratloo, Arash, et al. “Transparent Run-Time Defense Against Stack Smashing Attacks.” *Web.Eecs.Umich.Edu*, web.eecs.umich.edu/~aprakash/security/handouts/baratloo00transparent.pdf.