An Insight into Buffer Overflow Attacks and Kernel Security in Operating Systems

L. Joshua Crotts
Department of Computer Science
University of North Carolina at Greensboro

***Abstract*—Attacks at the kernel of an operating system leave it vulnerable to the curious enthusiast market, but likewise to the malicious black-hat hacker group of users. Buffer overflow attacks, in particular, aim at poorly-maintained software with the desire to take advantage of unguarded sections of memory, since direct manipulation to the kernel and operating system-level modes are generally locked under tight control. However, some user-level programs change and alter data at such a low level that it opens holes and potential for damage and security risks. This paper investigates the history of buffer overflow attacks, how it plays into the development and security of operating systems, as well as some insight into how these methods of attack work. We use three papers as a reference for the figures, and relevant information.**

***Index Terms*—Buffer overflow, kernel, security, operating systems, Linux, vulnerability, C programming, integer overflow, software engineering**

I. Introduction

PER Cowan et al., buffer overflow attacks constitute the greatest security vulnerability in the decades past [1]. Their interest to hackers and those to infiltrate a system is relatively self-explanatory: with such a common and easy way to exploit a system, it is no wonder why they choose this form of attack over other, more complex alternatives. As suggested, buffer overflow attacks allow malicious users to change the state of a program or an operating system to do one of several tasks, ranging from the collection of unintended/protected data, to the desire of crashing the system altogether. When attacking the kernel of an operating system (or the operating system internals in general), one is attacking the trusted computing base of a system [2]. Current and previous research focuses on detecting buffer overflow attacks (as well as kernel vulnerabilities and security risks), but preventing or mitigating problems in the wild is a harder task than one may originally believe. Dalton et al. state that many worms and viruses take advantage of buffer overflow in unprotected software [3].

According to Chen et al., the aforesaid previous studies and research attempted to experiment with *theoretical* patches and the good intention of encapsulating an operating system to the point where such important risks are not exposed to the user in any way[2]. In addition, they also explore what these vulnerabilities are, and how they affect the integrity of an operating system. The authors note, however, that these papers lack substance in understanding what techniques are used to *resolve* the issues [2]. An operating system such as Linux is very complicated to change, and while it may be open-source (allowing for the non-malicious users to find and understand bugs that cause security flaws), this opens the potential for those willing to perform more egregious acts. Moreover, because Linux is written in the non-safe language of C, it is understandable that some would prefer rewriting the kernel and structure in a language that enforce improved paradigms which reflect higher-level programming languages like bounds checking, but continues to operate at the lower level required by system programmers. Of course, this feat is not trivial, as the kernel is millions of lines of old and new code. As a result, the challenge for writing secure and effective code in an old language continues to dominate the Linux kernel source, but as vulnerabilities are found and patched, developers learn to investigate such problematic code and protect what they write with the appropriate safeguards and measures.

II. Buffer Overflow

Before we continue, let us define what a buffer overflow attack is, in a more formal sense. Buffer overflow attacks are, as the name implies, an attack on a piece of software that allows for potential arbitrary code execution, root access, and other upper-level permission tasks. These attacks by nature attack a buffer in memory, which overwrites preexisting code or data, thereby changing the state of the program. In other words, according to Cowan et al., the overall goal of a buffer overflow is to subvert the state of a program to perform acts that were not intended by the programmer [1]. There are numerous ways to "overflow" the "buffer", but it depends on what is being overflown, as well as what data is overflowing said destination.

Several of these vulnerabilities are present in older C code, with functions such as `char *strcat(char *dest, const char *src)`. String manipulation and functions that operate on strings or char pointer arrays are a prime target for weaker C libraries and code in the kernel. This particular example copies the `const char` pointer `src` to the end of the destination array `dest`. Therefore, the `dest` buffer, ideally, must have enough allocated bytes to store both its original string (provided it exists; it may also be an empty buffer), and the `src` buffer, concatenated onto the end of the original. The naive implementation without an enforcing size limit copies all data from `src` to `dest`, without care for if it overwrote data elsewhere in the code. This poses a few problems: firstly, we may experience a segmentation fault;

accessing data/memory outside the program space results in an "exception" being thrown (a signal in C/operating system terms, also known as `SIGSEGV`). If this does not occur, however, then it may be the case that the attack fills the buffer, but results in an attack that remains within the scope/memory bounds of our program. In other words, the overflown written-to memory is declared in the current memory stack.

At first glance, it may appear that this does not harm the program or the end-user; if the buffer is filled with "garbage" to the computer, then theoretically nothing occurs. However, what happens if the program is to execute code via a system call or `exec` immediately after the buffer data is read? Then, it may be the case that the buffer overwrote that executable code to perform something else; the unintended consequences of the developer not guarding against large input. We will further elaborate on this later. The solution to this function, in particular, as we will also examine later with a few other important C data-retrieval functions, is to limit how much data goes into either buffer. Thus, a safer function namely `char *strncat(char *dest, const char *source, size_t bytes)` exists to combat such issues. This function copies `n` bytes from our `src` to the `dest` buffer, and returns the result, bypassing the potential for an attack.

A. Buffer Overflow in C Functions

Several other C input functions pose problems similar to what `strcat` introduced. We will list a few and examine their security risks, and then mention their safe(r) counterparts.

1) `int scanf(const char *format, ...args)` - An input function similar to the standard output `printf`, `scanf` reads in a formatted string, alongside pointer arguments to store the result of the requested string. For example, `scanf("%10s", &buf)` reads a string that is ten bytes long into the pointer `buf`. Suppose that, for arguments sake, we read in two strings with the following code: `scanf("%10s", &buf)`, `scanf("%s", &buf2)` from standard input. Further suppose that the first string s_1 = `"1234567890ABCDE"`, s_2 = `"ABCDEFGHIJ"`. The important detail here is that, with the current implementation, the second call to `scanf` will not properly execute. `scanf` ignores trailing newline characters, meaning that our second call consumes all remaining data in the input stream after the first ten bytes are read from s_1. The safe alternative to `scanf` is to use a different function for reading input data altogether, which we will present next.

2) `char *gets(char *src)` - Another data input function with the goal of reading in a string into a buffer, as opposed to any arbitrary formatted input. This function is simpler than `scanf` because, in order to perform a buffer overflow, all one needs to do is create a buffer of n bytes, then supply a string with n_0 bytes such that $n_0 > n$. This, in turn, fills `src` with more data than it is intended to hold, which may overwrite other segments in the code stack or cause a segmentation fault. The solution for this function is to use the bounded-buffer function: `char *fgets(char *s, int size, FILE *stream)`, which guards against overflow by only reading `size` bytes from the `stream` pointer, which is then stored in the pointer `s`.

3) `char *strcpy(char *dest, const char *src)` - String copy function with almost identical consequences to the previously-mentioned `strcat` function. Copying more bytes than allocated into the `dest` pointer results in overflow. The solution is to use `char *strncpy(char *dest, const char *src, size_t n)`, copying only `n` bytes from `src` to `dest`.

III. Methods of Attack

Now that we have explained several types of buffer overflow risks and functions in C, we will describe key components to an attack, and what makes one successful. There are a few styles to buffer overflow attacks, each with their own goals and methods of approach.

Cowan et al. propose two "sub-goals", describing two possible candidates for starting a buffer overflow attack [1]:

1) Arrange the malicious code such that it exists in the program space.
2) Arrange the malicious code such that, when performing the overflow, a jump instruction (of some variation) leads to the execution of said code.

We will examine and explain the two approaches, and fill in some of our own details.

When arranging the code such that it exists in the program space, this implies that we can inject malicious code into the program to do what we desire. One approach to this is to use preexisting code and "parameterize" it to satisfy our needs. In other words, we alter the original input to the code (provided that it exists), and substitute in our code. We alluded to this previously with the `exec` example. Suppose that we have a program that acts as as shell, and we read in the next command via a call to `gets(...)`, which stores the string from standard input into a buffer b. Then, we immediately call the Linux function `execvp(...)` in which we pass this string, without parsing the input for validity. Further assume that we overflow the buffer, and that our implementation of the code uses this buffer when calling `execvp`. The attack is then trivial because we can overflow b by overwriting the argument space for the parameter of `execvp`, and store what we want to execute, thereby changing the pointer. An example like this is crucial to understanding the complexities and significance of protecting against bad input and using functions that do not have negative and unintended consequences, as we previously suggested.

On the other hand, we can inject code that is *not* in the currently-executing victim program, but rather what we wish to execute. Storing the attack code in this way allows the attacker to use it later, perhaps in a jump instruction via

a function pointer or misguided return address overwrite. These buffers may be allocated in any segment of memory, including the stack and the heap (static and dynamic memory respectively).

Oppositely, we may arrange the malicious code such that we alter the program control flow, bypassing whatever logic currently exists in the code in favor of the arbitrary code injected into a buffer. There are several methods to this, and we will describe them as follows:

1) Stack-smashing: When a function is pushed onto the call-stack, an *activation record* is pushed in its place, containing information such as the stack pointer, frame pointer, return address, parameters pushed to the stack, and space for temporary and local variables. For our purposes, we will illustrate these commands with MIPS assembly. Note that the MIPS code below was self-written, whereas the x64 presented later is compiler-generated.

```
    .text
_main:
    .globl main
    ; allocate 40 bytes for frame.
    subu $sp, $sp, 40
    ; save return address.
    sw $ra, 32($sp)
    ; save current frame pointer.
    sw $fp, 28($sp)
    ...
xf_main:
    ; load old frame pointer
    lw $fp, 28($sp)
    ; load return address
    lw $ra$, 32($sp)
    ; deallocate stack frame.
    addu $sp, $sp, 40
.data
    ...
    ;$
```

Note the load word (`lw`) instruction with the annotation that it reloads the old return address off the stack frame. Malicious code from a buffer overflow attack could change this value. A buffer overflow will overwrite the space allocated for the string and exceed into the space for both the frame pointer and the return address if used properly, thereby resetting the location to somewhere it should not be. Thus, upon returning from the function and destruction of the activation record, the instruction pointer is misplaced, pointing at potentially malicious code. According to Cowan et al., this is the most common buffer overflow attack.

2) Another variation listed in the previous research uses function pointers in C to redirect program control. A *function pointer* is a pointer to a memory address that acts as the starting instruction of the function it points to. A similar comparison in modern programming languages are *callback functions*. Therefore, a buffer overflow attack may rewrite this address to point to a function or segment that executes arbitrary code. The difference between this and many other forms of attack is the time of execution; it is not necessary for a function pointer to be executed at a point in time, or even at all depending on the program. Though, even allowing for a rewrite to the address it points to is a significant security flaw and oversight. Below is an example of a function pointer, and the output in x64 assembly. For this, we compile via `gcc` with the `-S` flag (all comments generated by the compiler are omitted). Note that we create a function `void f(int *n)` that simply reassigns the value pointed to by `n`:

```
void f(int *n) {
    *n = 5;
}

int main(int argc, char *argv[]) {
    /* Create a pointer to the function f
    called fn_ptr that returns void and accepts
     an integer pointer. */
    void (* fn_ptr)(int*) = f;
    int n = 125;
    /* Call our function pointer and pass the
    address of n. */
    fn_ptr(&n);

    return 0;
}
```

```
        .section
        .globl          _f
        .p2align          4, 0x90
_f:
        .cfi_startproc
        pushq           %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq            %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movq            %rdi, -8(%rbp)
        movq            -8(%rbp), %rax
        movl            $5, (%rax)
        popq            %rbp
        retq
        .cfi_endproc

        .globl          _main
        .p2align          4, 0x90
_main:
        .cfi_startproc
        pushq           %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq            %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq            $32, %rsp
        movl            $0, -4(%rbp)
        movl            %edi, -8(%rbp)
        movq            %rsi, -16(%rbp)
        leaq            _f(%rip), %rax
        movq            %rax, -24(%rbp)
        movl            $125, -28(%rbp)
```

```
leaq        -28(%rbp), %rdi
callq        *-24(%rbp)
xorl        %eax, %eax
addq        $32, %rsp
popq        %rbp
retq
.cfi_endproc
;
```

The point of interest is the line `callq *-24(%rbp)`. If this address offset from the base pointer is manipulated via a buffer overflow, the program flow of control is likewise altered.

A. Integer Overflow

We will briefly touch on integer overflow, since it relates to the topic of buffer overflows in a similar manner. Integer overflow or underflow occurs when the size of an operand exceeds the storage capacity for that value. For instance, a value of 10,000,000,000 (ten billion) exceeds a 32-bit word sized memory location, and if 64-bit instructions are not present (as such in x86 assembly and others), using such a large value may result in overflow. However, many implementations guard against such inputs, so this generally does not result in a problem. What *can* result in a problem is when a binary operation is performed on two values that are very close to that threshold. Suppose we take two signed 32-bit values x and y, where $x = y$, and $x = 2,147,483,647$, the maximum value for a 32-bit integer. If we try to perform a signed `add` operation on these values, for instance, then store them in a 32-bit register, it is possible that a flag may be triggered, causing sign errors or buffer sizes to allocate erroneously [2].

IV. Solutions

At this point, we have reviewed some assembly and C code, and examined a few ways that buffer overflow attacks occur. In this section, we will describe a few solutions to the problem at hand.

One possibility, as suggested by several authors [1, 2, 3], is to *always* write clean and non-vulnerable code at the kernel/operating system level(s). Implying that buffers are maintained, checks are performed, and the programmer writes impeccable and impenetrable code. Of course, in plenty instances, this is easier said than done, primarily due to the aggressive performance paradigm that C enforces over program correctness at times. Combined with the fact that C is a moderately unsafe language to begin with (as demonstrated and elaborated on by the handful of above functions, as well as the fact that raw pointers are used with no automatic garbage collection), it is obvious why many consider C a dangerous language to write in. Allowing programmers to access memory directly, and providing the control of allocation and de-allocation grants flexibility in the power and potential of a program. Conversely, it leaves many, more novice (or even the experienced) programmers at risk of accidentally writing egregious code. Moreover, time is money to most software engineers, and as a result, must resort to writing fast code that may not catch all security risks and bugs. Consequently, tools analyzing source code (static and dynamic analyzers) exist to find bugs and problems in source code, as well as compiler-generated warnings when using a volatile (unrelated to the C `volatile` keyword) function. For example, when using `gets(char *buf)`, the compiler outputs the following warning, informing the programmer to consider choosing a different, non-antiquated function: `"warning: the 'gets' function is dangerous and should not be used."`. An issue with some static analyzers, though, is the sheer number of false positives encountered [2].

Another possibility is to check all array or pointer boundaries - meaning that every time a buffer is used or allocated, as well as any writing performed to the buffer, we check to make sure that the size of some input does not exceed the buffer size. Whether this is done at the user-level or compiler-level is a different debate altogether, but performing checks at both levels guarantees an extra needed layer of security instead of solely dedicating the job to a compiler.

Canaries are a compiler-generated solution to mitigating an attack as it occurs. Canaries are random values placed next to the return address in memory on the stack frame, and if a stack-smashing attack occurs, the program can continue in a way that ensures security and safety is maintained. A random 32-bit canary, for instance, is chosen due to the fact that it is hard to guess with a typical stack-smashing attack; if the attacker is unaware that a canary exists, they have no easy way to combat it.

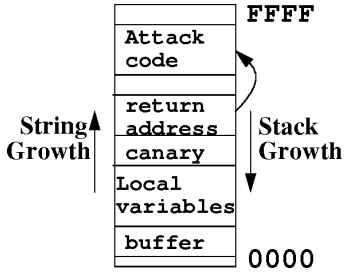

Fig. 1. Example of canary - random 32-bit word is placed directly next to the return address on the stack. Buffer overflows will haphazardly overwrite this value as well. This figure is used in paper [1].

The primary problem of a buffer overflow attack is that, when arbitrary code is inserted into the buffer, that code is executable in some way if no safeguards are implemented. So, a solution to this is to make all buffers non-executable [1]. This, however, proves to be futile because in modern programming and applications, buffers are used for optimization purposes and thus reduce backwards compatibility.

It is also feasible to use *memory tracking* [2] to determine when the kernel is using memory that it should not, or instructions that were not previously there (those from redirected control flow attacks).

As we stated before, Linux (and namely the Linux kernel) is written in C. Certain languages reduce as much contact to/with the operating system as possible by abstracting the core details, but with these abstractions comes a performance hit - something that many programmers at this level can-

not afford to have happen. Furthermore, rewriting the entire Linux kernel in another, safer language would be a nearly impossible feat, so it is easier to maintain the legacy code and patch/update the bugs as they are discovered by testers and quality assurance. In addition, fixing one exploit may open the door for other exploits, or even worse, a broader classification of exploits. Chen et al. describes different kernel designs such as Stanford's HiStar and Minix, both derivations of a microkernel architecture. These kernel designs attempt to move as much information and functionality into user space as possible, leaving minimal, direct interaction with the kernel. Though, such implementations are few and far between with not enough real-world experimentation [2].

V. Future Considerations

Future work into analyzing code and ensuring the safety of computer programs and the kernel is of utmost importance. Nowadays with the vast array of programming languages and paradigms out there, paired with the wide knowledge and expertise of programmers who are aware of such risks and vulnerabilities, problematic code ought to be kept to a minimum. Legacy code is a different story, however - and instead of rewriting an entire segment of code consisting of hundreds of thousands of lines, it is better to investigate problematic sections that allocate memory, work directly with buffers, or use the safe alternatives of functions (e.g. "n" versions of various C functions). It is interesting to see and research the different tools and programs that walk through programs and examine them for flaws, compared to the mundane task it is for a human programmer.

VI. Conclusion

In summary, buffer overflow attacks are severe and prevalent in today's computing software, despite being decades old. Operating system vulnerabilities play critical roles in tightening security since attacks at the kernel are detrimental to the integrity of a system and its data/components. We examined how previous research explains methods of attack, as well as several examples of where these vulnerabilities exist in legacy and Linux code. We further described solutions to said bugs and attacks, particularly focusing on their detection and prevention, rather than eliminating the threat in the act. Computer security in operating systems is a difficult and arduous adventure to explore, but is absolutely necessary for both programmers, as well as everyday users, since viruses and worms affect both parties.

Acknowledgements. : This research paper was written for my graduate Principles of Operating Systems (CSC 662) course at the University of North Carolina Greensboro in the Fall 2020 semester.

References

[1] C. Cowan, F. Wagle, Calton Pu, S. Beattie and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, Hilton Head, SC, USA, 2000, pp. 119-129 vol.2, doi: 10.1109/DISCEX.2000.821514.

[2] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. "Linux kernel vulnerabilities: state-of-the-art defenses and open problems," *In Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11)*, Association for Computing Machinery, New York, NY, USA 2011, Article 5, pp. 1–5. doi: https://doi.org/10.1145/2103799.2103805.

[3] Michael Dalton, Hari Kannan, and Christos Kozyrakis, Real-world buffer overflow protection for userspace & kernelspace, *In Proceedings of the 17th conference on Security symposium (SS'08)*, USENIX Association, USA, 2011, pp. 395–410.