Automated Buffer Overflow Exploitation and Profiling

Aidan Walsh
ECE 580: Hardware Security
Sharad Malik

Abstract

Memory bugs in low-level programming languages are an ongoing issue. These bugs include Buffer Overflow vulnerabilities that enable a user to overwrite values located on the stack. This overwriting can lead to unintended program execution that can ultimately result in malicious code being executed. Finding such vulnerable programs and verifying that these vulnerabilities can be exploited can be a time consuming and difficult process. To speed up the process and enable vulnerability verification at a larger scale, this work attempts to construct a program that automates the construction of an exploit that "pops a shell" for given binaries. Furthermore, this program profiles the given binaries to assist security researchers in developing their own exploit or bolster the security of the program. This program, in its current form, is able to automatically "pop a shell" for simpler 32 bit executables that do not use ASLR or a Stack Canary, but can be modified to handle these defenses.

Introduction and Motivation

Memory corruption attacks have existed since the beginning of computer systems. Payer et. al's *SoK: Eternal War* i*n Memory [3]* discusses the plethora of attacks and vulnerabilities that are present. Many of these memory bugs occur in C and C++ programs because these languages provide the developer low-level features such as memory manipulation. These features are provided to enhance the performance of these programs since they are frequently used for operating systems and performance-critical systems. These attacks can modify pointers, change code execution, inject code, leak information, control the flow of the program, or corrupt the code being executed. Such attacks that are able to do all of this are called Buffer Overflows. Conducting these attacks can be time consuming and difficult for adversaries and developers hoping to verify the safety of their programs. Conducting these attacks requires inspection of the stack, memory addresses being used, the memory layout of the program, assembly instructions that the attacker has access to, and defenses that must be bypassed. Thus, having a tool that is able to profile the binaries and give the attacker assistance in constructing an exploit would save time and enable the exploitation of many more programs. Also, having an automated method of exploitation would let a developer run this automated method on many programs, enabling them to verify the security of these programs quickly and easily. There exists many ways to exploit vulnerable programs. Handling all of these ways can be burdensome and may result in a fruitless exploit. Thus, automated exploit construction can prove the ineffectiveness of some methods and a profiler can help the developer construct the most effective attack.

Although the process for constructing an exploit can be long and difficult, it is sometimes repetitive and algorithmic. For example, some defenses can automatically shut out some avenues of attack, and inspection of the stack during program execution is always performed. Thus, putting the process of exploitation construction into a program may be effective, generalizable and applicable to many programs.

Background and Setup

Buffer Overflows

Buffer Overflows overwrite memory on the stack by taking advantage of certain c-library functions. These functions include gets(), printf(), strcpy(), and strcat(). This project primarily focuses on the use of gets(). Functions such as gets() take in an unbounded amount of user input, and so this user input can overwrite values on the stack if the given buffer is too small. These overwritten values can be local variables, function arguments, and return addresses. As a result, the values of variables change, malicious code can be injected, and program execution can be modified. Typical Buffer Overflow attacks overwrite the return address of functions with the address of malicious code so that the malicious code may be executed.

Mitigations

There exists many mitigations to prevent Buffer Overflows from overwriting important values on the stack and changing the intended execution of programs. First, there exists Address Space Layout Randomization (ASLR) which randomizes the offsets of the different memory segments of the Process Address Space. So, when the process to run the executable begins, the location of the Stack, shared memory, and executable code are randomized (their offsets are, so their relative positioning inside each memory segment remains the same)[2]. Position Independent Executable Code (PIE) complements ASLR by randomizing the location of the .text, .plt, .got, and .rodata sections, alongside the random offset of the Executable Code section generated by ASLR. There also exists NX which makes the Stack non-executable. There is a mitigation called W-X which, in addition to making the Stack non-executable, enforces the following invariant in the Process Address Space: no memory section can be both writable and executable. To even prevent the Stack from overflowing, there are Stack Canaries[1]. These Stack Canaries are randomized values placed before the saved Base Pointer and Return Address of the function Stack Frame and whenever the function returns, the integrity of the Stack Canary is checked. This makes it very difficult to bypass the Stack Canary unless it can be leaked by a string vulnerability or brute-forced in a 32 bit system. The current state of this project focuses on bypassing the following vulnerabilities: PIE and W-X on 32 bit systems, but fails to address ASLR and Stack Canaries.

“Popping a Shell”

As an adversary or developer attempting to test the security of their program, the ultimate goal of a Buffer Overflow can be to “pop a shell”. This is where the execution of a

program changes to begin a shell session. This can be done by calling the function Execve("/bin/sh") or by using the assembly instruction "int" for interrupt with the correct arguments in the registers. In the real-world setting, an adversary would be communicating with a server that would be running a program, let us call this program authenticate.c. This server could be a Google server that is storing employee credentials. When the adversary is communicating with the server, the server could be authenticating the identity of the adversary, making sure that the adversary is an employee for Google. The server could have a buffer overflow vulnerability so that if the adversary is able to "pop a shell", then the adversary would have access to a shell session on the server. As a result, the attacker would have administrator privileges and could navigate around the files and data within the server.

Return-into-libc Attack

One method of buffer overflow that can "pop a shell" is a return-into-libc attack [4]. During the linking phase of compilation, functions from Libc are linked to the binary. Libc contains a plethora of useful functions such as Execve() which can be used to "pop a shell". Libc is not always linked, however, but in most cases it is. To conduct a Libc attack, the return address of a function is typically overwritten to point to a function in Libc that is used to "pop a shell". Refer to figure 1 for the structure of a Libc attack.

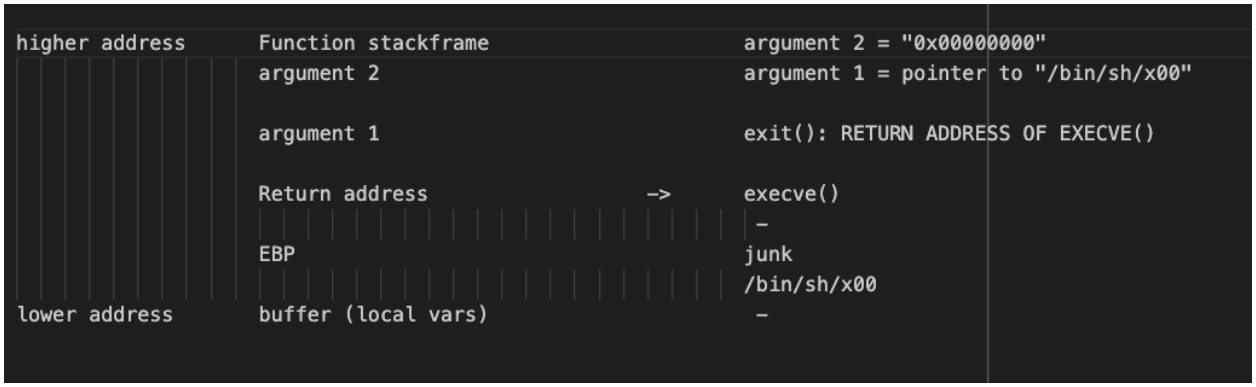

Figure 1: On the left, we have the stack layout before a Libc attack, and on the right, we have the stack layout after a Libc attack. This is to "pop a shell" for 32 bit executables.

The stack is overwritten with "/bin/sh/x00" at the beginning, then filled with junk up until the address of Execve (that overwrites the original return address), followed by the address of Exit(). This inclusion of Exit is the return address for Execve(), so Exit() will be called after the shell is "popped". This makes the attack harder to detect since the program exits safely. The address of Exit is then followed by the arguments for Execve(): a pointer to "/bin/sh/x00" at the beginning of the buffer, then 0x0 which indicates that no more arguments should be used. The inclusion of this second argument is essential. This is exactly how this paper's automated exploitation performs the Ret-To-Libc attack. We note that to have the address of /bin/sh/x00, that we must have the address of the stack, thus ASLR is deactivated in the current state of this project.

Return Oriented Programming (ROP)

ROP is where sequences of assembly instructions are chained together to execute code that will achieve an ultimate goal, such as “popping a shell”. An instruction used for a ROP chain is known as a gadget. Sequences of instructions that can “pop a shell” on different systems can be found at https://shell-storm.org/shellcode/index.html. Some sequences can be as short as 23 bytes. If Libc is not present then ROP chaining can be used to execute the adversary’s intended code. If ASLR or PIE are activated, then this will make things significantly more difficult, especially on a 64 bit system. On a 32 bit system, the random offsets created by ASLR can be feasibly brute forced, and so using gadgets from linked libraries is most typically done. Refer to figure 2 for a diagram of the stack before and after a ROP chain.

Figure 2: On the left, we have the stack layout before a ROP chain attack, and on the right, we have the stack layout after a ROP chain attack. This is to “pop a shell” for 32 bit executables.

We see that when we reach the original return address, we instead are pointed to an add and return instruction. That return instruction will then pop the next value off the stack and set the Instruction Pointer (IP) to that address. This next address points to a mov and ret instruction. This process continues until the adversary’s intended code is executed. The current state of this project does not utilize ROP chaining.

Current Setup and Limitations

This project currently only exploits 32-bit programs that have the W-X and PIE mitigations activated. This is because the automated exploitation, as of now, only utilizes a Ret-To-Libc attack which does not utilize the Executable Code section of the process’s memory and does not write malicious code to the stack to be executed. It does not exploit binaries that have ASLR or/and Stack Canaries. For 32 bit programs, ASLR and Stack Canaries can be feasibly bypassed and so the recommendation system still recommends a Ret-To-Libc attack if ASLR is activated - it will just take a longer time due to brute-force. If a Stack Canary is present, the program’s recommendation system first recommends bypassing the Stack Canary by a leak or brute-force, then will continue (after bypassing it) as it would without the Stack Canary. Thus,

this program can be furthered by adding functionality that attempts to leak the Stack Canary and can brute-force ASLR and the Stack Canary for 32 bit systems.

The crux of the program's recommendation system is the fact that it assumes a 32 bit binary - in a 64 bit system, brute-forcing the Canary or ASLR is unfeasible. It also bases its recommendation system and automated exploitation primarily on whether or not the exploitable program has Libc linked. If it doesn't it goes down the path of either executing code on the Stack or using a ROP chain. Otherwise, the program will attempt a Ret-To-Libc attack. This program can be extended to handle 64 bit binaries - the recommendation system will just need to change and the automated exploitation will need to be updated to handle 64 bit values. It cannot be simply used on 64 bit binaries because the methodology for calling a function is different - instead of directly using values on the stack as arguments, the arguments are stored in registers. Thus, calling Execve() in an exploit for a 64 bit binary may require manipulation of register values. This can be feasibly done.

Methodology

To construct this automated exploitation, I first began with a bottom-up approach of manually exploiting a simple program with no defenses and converting the whole process into code. Then, after successfully constructing the automated exploit, updating it to be able to handle more complex execution of the exploitable binary (such as multiple times that gets() is called, as opposed to once, or execution dependent on the input from the adversary). Generally, successfully exploiting a binary happens in two stages: probing and exploit construction. This is exactly what this program does to successfully execute automated exploitation. Firstly, though, none of this is possible without the use of several tools.

Tools

This program's probing and exploitation construction involved the following tools: objdump, GDB, checksec, ROPgadget, and Pwntools. Objdump is a command-line tool that was used to disassemble the assembly of the binary. It was used to extract the .text functions that could be vulnerable, and to see which functions use vulnerable functions such as gets(). GDB was used to simulate execution and view the memory of the exploitable binary. All of the memory was logged into files and parsed by this paper's program. From viewing and logging the memory layout at specific times of the binary's execution, the program is able to infer the requirements necessary to exploit the binary. Using GDB, we are also able to infer if ASLR is activated (by viewing the location of memory on process startup) and we can see where all Libc functions are located if ASLR is deactivated. Checksec is a command-line tool that is able to infer if the following defenses are activated in the binary: PIE, Stack Canary, and NX. It also tells us if we are dealing with a 32 bit binary. Parsing this information (and using the ASLR information from GDB), the program is able to recommend a method of exploitation and attempt to construct an exploit. ROPgadget is a command-line tool that inspects the process's address space to look for strings and assembly instructions that are helpful for a ROP chain. This tool is not directly used in this paper's program, but will be used in future to be able to exploit binaries that are not linked with Libc. Pwntools is a command line tool, but, most importantly, a Python

library (imported as pwn) that allows us to construct a program that automatically constructs an exploit and provides an exploit recommendation. Pwntools lets the user start the binary and attach to it, letting the user send any information to the process whenever they want. Pwntools also lets the user attach GDB to it, enabling easier exploit construction and logging for the automated exploitation to work properly. We are also able to package bytes nicely so that it is sent properly to the running process. Overall, Pwntools is a great library to use for debugging, probing, and conducting Buffer Overflow attacks.

Probing

The purpose of the probing is to understand the binary to enable the program to provide a recommended method of exploitation and to correctly conduct an automated attack on the binary. To achieve this, our probing asked and answered the following questions:

1. Does the program take input?

Pwntools lets us attach to the process and gives us an exit code which indicates when the program terminates as we do/do not send it input.

2. How many inputs does the program take?

We send inputs using Pwntools until an exit code of 0 is returned from the process.

3. What security protocols is the program using?

We simply use Checksec.

4. Can the inputs overflow (seg-fault?) How long until a seg-fault?

We construct input and increase the size of the inputs 1 byte at a time until we reach a large enough value that indicates that a buffer overflow is highly unlikely. We know that we had a segmentation fault depending on the returned error code (-11).

5. Is ASLR activated? Can Libc be used?

We use GDB to start the process multiple times. We ensure that the disabled randomization is off for GDB and we print the locations of places outside of Executable Code. If they change on every program run, then ASLR is activated. If Libc functions have an address, such as Execve(), then Libc is linked.

6. Can we perform a Buffer overflow on main?

32 bit binaries have an interesting functionality at the end of main() where they use an indirect reference (“lea”) to the saved return address instead of the typical “leave, ret”. This makes it very difficult to properly overflow main() such that it returns to a different function or code (especially if ASLR is activated). Thus, if there is no “leave”, then the main function cannot be properly overflowed.

7. What can we learn from the inputs and all functions in the program?
    a. For all functions ask and answer the following questions:

For all functions found in the .text section, we maintained a data structure that kept track of all of these values for each function. We used a dictionary to do so where the key is the name of the function (such as “main”) and the value is a tuple of all the values given below.

i. Does it contain a vulnerable function? (gets, strcpy, ...)

A boolean value is stored. We used objdump to inspect the assembly of each function in the .text section.

ii. If so, what functions are they?

An array of string values of the name of the functions. For example, [“gets”, “gets”].

iii. What is the offset of the call instruction from the start of the function?

An array of integers where index i is used for the same index of the instruction found at index i in part ii (the part before). This value is used for GDB to break at this address.

iv. What is the offset of the next instruction from the start of the function?

Another array of integers. This value is also used for GDB to break at this address.

v. What is the offset of the “ret” assembly instruction?

An array of integers is used to store. This value is also used for GDB to break at this address.

vi. What is the size of the buffer up to the return address?

An array of integers is used to store. Based on how we parse the addresses given by GDB, we are correct with the size of the buffer with an error of up to at most 3 bytes. This is due to byte alignment because, in a 32 bit program, buffers may be aligned to byte 0,1,2, or 3 for 4 byte words. Our program assumes proper alignment to every byte 0, but this is not the case, thus we have this small error.

vii. At what address does the buffer start at? (no ASLR)

An array of integers is used to store this stack address.

viii. What input number is this? (the first or second or... input?)

An array of integers is stored. We need to know which input this is, because an exploit is unique for each input. So, when we want to exploit input 2, we know how to construct that exploit, which will be different to input 3. This was achieved by creating an input file that sent in hex values in incrementing order starting at 0x01010101 and GDB searched for these unique hex values in the stack to tell us the order and location of buffers.

8. Extract addresses of Execve() and Exit()

Finally, we used GDB to print the addresses of the Libc functions Execve() and Exit(), if Libc is present.

Exploit Construction

After the probing, we have extracted all the information we need to best exploit as many inputs as possible. For all inputs, we construct an input such that it causes the stack to look like the stack given in figure 1 for a Ret-to-Libc attack. The automatic exploitation could also be used to construct an exploit that utilizes an executable stack or ROP chaining, or brute-forcing a Stack Canary or ASLR offsets but, as discussed earlier, this is room for future improvements. We use Pwntools to package the bytes all together and to send the exploit to the running binary. We know that we have "popped a shell" when we can type commands (such as "ls") as the program is running and get back the correct output. Due to the error discussed for question 7)a)vi), our exploit gives 4 attempts and one of these correctly "pops a shell". Using the data structure, we also must ensure that we send the correct exploit for the correct input. Question 7)a)viii) answers this for us and our exploit automatically handles this.

Testing and Results

After creating the program for the automated Buffer Overflow, we then test its effectiveness on 2 C files: simple_vuln.c[1] and name.c. The former is a simpler c file that calls gets() twice and the latter is a more complex program (but we still note that it is quite simple) that simulates a program that stores credentials to a database and asks for login credentials. From these 2 C files, we construct several binaries that can be executed: vuln1, vuln2, vuln3, vuln4, and name. Vuln1 is simple_vuln.c without any active defenses compiled 32 bit. Vuln2 is simple_vuln.c compiled 32 bit with NX. Vuln3 is simple_vuln.c compiled 32 bit with NX and PIE. Vuln4 is simple_vuln.c compiled 32 bit with NX, PIE, and a Stack Canary. To refer to the commands used for constructing these files, refer to environment.sh.

When running the program on vuln1, we see that a "shell is popped" on the program's 3rd attempt (offset of 2 from predicted start of buffer). The size of the buffers are the same for vuln1,vuln2, and vuln3, and since NX and PIE have no effect on our Ret-To-Libc exploit, a "shell is popped" on the program's 3rd attempt for vuln1,vuln2, and vuln3. We also see that the recommendation system recommended a Ret-To-Libc exploit. For vuln4, a Stack Canary is present, so we see that no attempt is made at constructing an exploit, we cannot properly segfault the binary, and a recommendation is made to first leak and/or brute-force the Stack Canary since we are dealing with a 32 bit binary. Additionally, we see that the probing successfully gave us how much we have to fill up the buffer until we change the return address. Notice how it is 4 more than the size given for vuln1,vuln2, and vuln3 - this is because there is a 4 byte Stack Canary at the end of the stack now that precedes the saved return address and base pointer.

All testing is done on Ubuntu 22.04, Cloudlab Wisconsin c220g2 server, which is an x86-64 machine with 2 10 core CPUs operating at 2.6 GHz. The 32 bit files are compiled into i386-32-little architecture and the 64 bit files are compiled into amd64-64-little architecture.

[1] All code can be found at https://github.com/Aidan-Walsh/Automated-Binary-Exploitation. Instructions for running the program are given in the readme.

Limitations and Difficulties

This automated buffer overflow exploit is very limited and so there is much room for improvement and development. As stated earlier, it is not able to bypass ASLR or a Stack Canary - it only provides recommendations to the adversary/developer. Our program is also only able to handle 32 bit programs and only performs automated Ret-To-Libc exploits. Furthermore, it has difficulty exploiting programs with complex execution. This means that programs that take special input or have multiple execution paths may not be properly exploited using our automatic Buffer Overflow. For example, a program that calls gets() in an exploitable function (outside of main() for a 32 bit binary) may only do so if the user inputs “yes” or “no” or a specific number. Our automatic Buffer Overflow would not detect this since it only inputs random values. To be able to handle this, our program could have more user interaction. Someone hoping to use our program may be required to reverse engineer the binary and give our program values that it must use to find the Buffer Overflow. The automated Buffer Overflow could also look at the text section of the binary and use any found strings as a starting point for exploring execution paths. Our program also has difficulty properly exploiting functions that call gets() multiple times in a function. This is because multiple calls to gets() could be using the same buffer or different buffers and our code does not account for this complexity. To fix this, our code would just need to be able to handle cases where a buffer will not necessarily begin at the highest place on the stack (an assumption we make in our code that simplifies things and makes exploitation possible for simpler programs). Furthermore, non-deterministic programs are not able to be exploited by our automated Buffer Overflow since our program takes advantage of a deterministic execution path to keep track of vulnerabilities and architectural characteristics such as buffer size. Finally, our program is only able to properly exploit uses of gets() - its primary focus is on gets(). Thus, our automated Buffer Overflow is very limited, but provides a great start for extending to more complex mitigations and programs.

During the development of this program, there were many difficulties encountered. The most important ones involved constructing the exploit. Firstly, when executing instructions on the Stack, sometimes a “Sig-ill” was returned - not a Seg-fault. This was because the binary’s execution was not properly aligned. For example, the exploit changed the binary’s execution to return to address 0x7ffff0003 on the stack, which is not properly aligned to 4 bytes. The address should be a multiple of 4. Also, when conducting a Ret-To-Libc attack that utilizes the stack for argument storage, there must be a 0x0 after the pointer to “/bin/sh/x00” because this tells the function call to discontinue looking at the stack for function arguments. Other difficulties were encountered when trying to use GDB to construct the exploit - the terminal had to be split before running, and the command “tmux” had to be run. Also, to correctly log output from GDB, pagination had to be turned off. Finally, since the buffers do not necessarily begin aligned to 4 bytes, this made predicting the buffer sizes more difficult and led to the 4 byte error stated earlier.

Conclusion

Memory vulnerabilities, such as Buffer Overflows, have existed since the advent of computer systems. The prevalence of C and C++ computer programs and need for high performance programs has enabled these vulnerabilities to remain. Properly exploiting these programs to cause a “shell to pop” can be cumbersome for adversaries and developers who hope to test the security of their programs. To make this process easier and faster, we developed an automated Buffer Overflow program that probes and exploits 32 bit programs and provides a recommended method of exploitation. Although successful on several programs that use NX and PIE defenses, this program is limited in its ability to exploit 64 bit programs, and programs that use ASLR and Stack Canaries. This program is also limited in its ability to exploit binaries with complex and non-deterministic executions. This program can be extended to handle all of these limitations and will likely require more interaction with the user, requiring the user to reverse engineer or have an understanding of the execution of the binary.

References

[1] BUTT, M. A., AJMAL, Z., KHAN, Z. I., IDREES, M., AND JAVED, Y. An in-depth survey of bypassing buffer overflow mitigation techniques. Applied Sciences (2022).
[2] SCHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. Association for Computing Machinery (2004), 298–307.
[3] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. 2013 IEEE Symposium on Security and Privacy (2013).
[4] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V., AND NING, P. On the expressiveness of return-into-libc attacks. International Workshop on Recent Advances in Intrusion Detection (2011), 121–141.