

Stack-Based Buffer Overflow

Memory exceptions are the operating system's reaction to an error in existing software or during the execution of these. This is responsible for most of the security vulnerabilities in program flows in the last decade. Programming errors often occur, leading to buffer overflows due to inattention when programming with low abstract languages such as `C` or `C++`.

These languages are compiled almost directly to machine code and, in contrast to highly abstracted languages such as Java or Python, run through little to no control structure operating system. Buffer overflows are errors that allow data that is too large to fit into a buffer of the operating system's memory that is not large enough, thereby overflowing this buffer. As a result of this mishandling, the memory of other functions of the executed program is overwritten, potentially creating a security vulnerability.

Such a program (binary file), is a general executable file stored on a data storage medium. There are several different file formats for such executable binary files. For example, the `Portable Executable Format` (PE) is used on Microsoft platforms.

Another format for executable files is the `Executable and Linking Format` (ELF), supported by almost all modern `UNIX` variants. If the linker loads such an executable binary file and the program will be executed, the corresponding program code will be loaded into the main memory and then executed by the CPU.

Programs store data and instructions in memory during initialization and execution. These are data that are displayed in the executed software or entered by the user. Especially for expected user input, a buffer must be created beforehand by saving the input.

The instructions are used to model the program flow. Among other things, return addresses are stored in the memory, which refers to other memory addresses and thus define the program's control flow. If such a return address is deliberately overwritten by using a buffer overflow, an attacker can manipulate the program flow by having the return address refer to another function or subroutine. Also, it would be possible to jump back to a code previously introduced by the user input.

To understand how it works on the technical level, we need to become familiar with how:

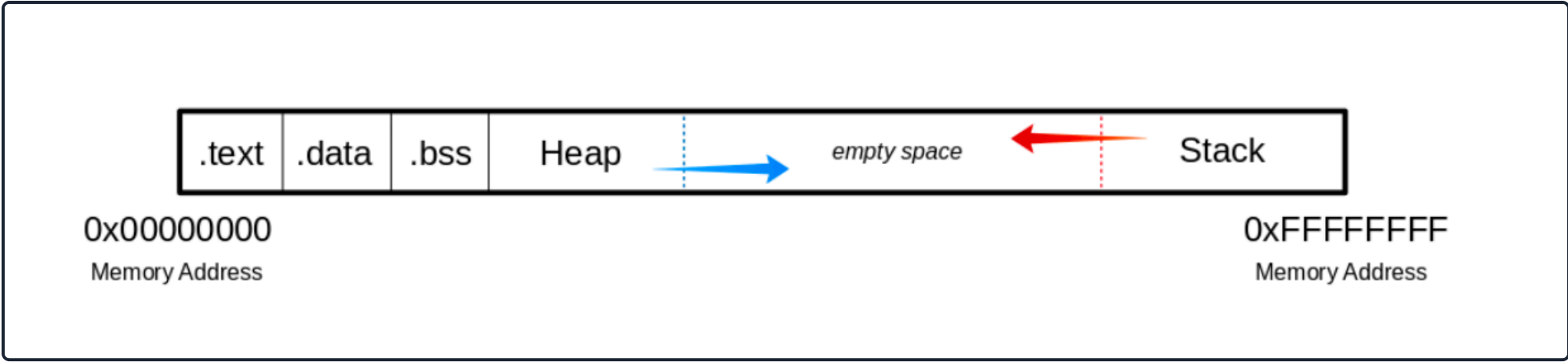
- the memory is divided and used
- the debugger displays and names the individual instructions
- the debugger can be used to detect such vulnerabilities
- we can manipulate the memory

Another critical point is that the exploits usually only work for a specific version of the software and operating system. Therefore, we have to rebuild and reconfigure the target system to bring it to the same state. After that, the program we are investigating is installed and analyzed. Most of the time, we will only have one attempt to exploit the program if we miss the opportunity to restart it with elevated privileges.

The Memory

When the program is called, the sections are mapped to the segments in the process, and the segments are loaded into memory as described by the `ELF` file.

Buffer



.text

The `.text` section contains the actual assembler instructions of the program. This area can be read-only to prevent the process from accidentally modifying its instructions. Any attempt to write to this area will inevitably result in a segmentation fault.

.data

The `.data` section contains global and static variables that are explicitly initialized by the program.

.bss

Several compilers and linkers use the `.bss` section as part of the data segment, which contains statically allocated variables represented exclusively by 0 bits.

The Heap

`Heap memory` is allocated from this area. This area starts at the end of the ".bss" segment and grows to the higher memory addresses.

The Stack

`Stack memory` is a `Last-In-First-Out` data structure in which the return addresses, parameters, and, depending on the compiler options, frame pointers are stored. `C/C++` local variables are stored here, and you can even copy code to the stack. The `Stack` is a defined area in `RAM`. The linker reserves this area and usually places the stack in RAM's lower area above the global and static variables. The contents are accessed via the `stack pointer`, set to the upper end of the stack during initialization. During execution, the allocated part of the stack grows down to the lower memory addresses.

Modern memory protections (`DEP/ASLR`) would prevent the damaged caused by buffer overflows. DEP (Data Execution Prevention), marked regions of memory "Read-Only". The read-only memory regions is where some user-input is stored (Example: The Stack), so the idea behind DEP was to prevent users from uploading shellcode to memory and then setting the instruction pointer to the shellcode. Hackers started utilizing ROP (Return Oriented Programming) to get around this, as it allowed them to upload the shellcode to an executable space and use existing calls to execute it. With ROP, the attacker needs to know the memory addresses where things are stored, so the defense against it was to implement ASLR (Address Space Layout Randomization) which randomizes where everything is stored making ROP more difficult.

Users can get around ASLR by leaking memory addresses, but this makes exploits less reliable and sometimes impossible. For example the ["Freefloat FTP Server"](#) is trivial to exploit on Windows XP (before DEP/ASLR). However, if the application is ran on a modern Windows operating system, the buffer overflow exists but it is currently non-trivial to exploit due to DEP/ASLR (as there's no known way to leak memory addresses.)

Vulnerable Program

We are now writing a simple C-program called `bow.c` with a vulnerable function called `strcpy()`.

Bow.c

Code: `c`

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

int bowfunc(char *string) {

    char buffer[1024];
    strcpy(buffer, string);
    return 1;
}

int main(int argc, char *argv[]) {

    bowfunc(argv[1]);
    printf("Done.\n");
    return 1;
}
```

Modern operating systems have built-in protections against such vulnerabilities, like Address Space Layout Randomization (ASLR). For the purpose of learning the basics of buffer overflow exploitation, we are going to disable this memory protection features:

Disable ASLR

Stack-Based Buffer Overflow

```
student@nix-bow:~$ sudo su
root@nix-bow:/home/student# echo 0 > /proc/sys/kernel/randomize_va_space
root@nix-bow:/home/student# cat /proc/sys/kernel/randomize_va_space

0
```

Next, we compile the C code into a 32bit ELF binary.

Compilation

Stack-Based Buffer Overflow

```
student@nix-bow:~$ sudo apt install gcc-multilib
student@nix-bow:~$ gcc bow.c -o bow32 -fno-stack-protector -z execstack -m32
student@nix-bow:~$ file bow32 | tr ",," "\n"

bow: ELF 32-bit LSB shared object
Intel 80386
version 1 (SYSV)
dynamically linked
interpreter /lib/ld-linux.so.2
for GNU/Linux 3.2.0
BuildID[sha1]=93dda6b77131deecaadf9d207fdd2e70f47e1071
not stripped
```

Vulnerable C Functions

There are several vulnerable functions in the C programming language that do not independently protect the memory. Here are some of the functions:

- `strcpy`

- `gets`
- `sprintf`
- `scanf`
- `strcat`
- ...

GDB Introductions

GDB, or the GNU Debugger, is the standard debugger of Linux systems developed by the GNU Project. It has been ported to many systems and supports the programming languages C, C++, Objective-C, FORTRAN, Java, and many more.

GDB provides us with the usual traceability features like breakpoints or stack trace output and allows us to intervene in the execution of programs. It also allows us, for example, to manipulate the variables of the application or to call functions independently of the normal execution of the program.

We use **GNU Debugger (GDB)** to view the created binary on the assembler level. Once we have executed the binary with **GDB**, we can disassemble the program's main function.

GDB - AT&T Syntax

Stack-Based Buffer Overflow

```
student@nix-bow:~$ gdb -q bow32

Reading symbols from bow...(no debugging symbols found)...done.
(gdb) disassemble main

Dump of assembler code for function main:
0x00000582 <+0>:    lea    0x4(%esp),%ecx
0x00000586 <+4>:    and    $0xffffffff0,%esp
0x00000589 <+7>:    pushl  -0x4(%ecx)
0x0000058c <+10>:   push  %ebp
0x0000058d <+11>:   mov    %esp,%ebp
0x0000058f <+13>:   push  %ebx
0x00000590 <+14>:   push  %ecx
0x00000591 <+15>:   call  0x450 <__x86.get_pc_thunk.bx>
0x00000596 <+20>:   add    $0x1a3e,%ebx
0x0000059c <+26>:   mov    %ecx,%eax
0x0000059e <+28>:   mov    0x4(%eax),%eax
0x000005a1 <+31>:   add    $0x4,%eax
0x000005a4 <+34>:   mov    (%eax),%eax
0x000005a6 <+36>:   sub    $0xc,%esp
0x000005a9 <+39>:   push  %eax
0x000005aa <+40>:   call  0x54d <bowfunc>
0x000005af <+45>:   add    $0x10,%esp
0x000005b2 <+48>:   sub    $0xc,%esp
0x000005b5 <+51>:   lea    -0x1974(%ebx),%eax
0x000005bb <+57>:   push  %eax
0x000005bc <+58>:   call  0x3e0 <puts@plt>
0x000005c1 <+63>:   add    $0x10,%esp
0x000005c4 <+66>:   mov    $0x1,%eax
0x000005c9 <+71>:   lea    -0x8(%ebp),%esp
0x000005cc <+74>:   pop    %ecx
0x000005cd <+75>:   pop    %ebx
0x000005ce <+76>:   pop    %ebp
0x000005cf <+77>:   lea    -0x4(%ecx),%esp
0x000005d2 <+80>:   ret

End of assembler dump.
```

In the first column, the hexadecimal numbers represent the **memory addresses**. The numbers with the plus sign (+) show the **address jumps** in memory in bytes, used for the respective instruction. Next, we can see the **assembler instructions (mnemonics)** with registers and their **operation suffixes**. The current syntax is **AT&T**, which we can recognize by the **%** and **\$** characters.

Memory Address	Address Jumps	Assembler Instruction	Operation Suffixes
0x00000582	<+0>:	lea	0x4(%esp),%ecx
0x00000586	<+4>:	and	\$0xffffffff0,%esp
...

The **Intel** syntax makes the disassembled representation easier to read, and we can change the syntax by entering the following commands in GDB:

GDB - Change the Syntax to Intel

Stack-Based Buffer Overflow

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main

Dump of assembler code for function main:
0x00000582 <+0>:    lea    ecx,[esp+0x4]
0x00000586 <+4>:    and    esp,0xffffffff0
0x00000589 <+7>:    push  DWORD PTR [ecx-0x4]
0x0000058c <+10>:   push  ebp
0x0000058d <+11>:   mov   ebp,esp
0x0000058f <+13>:   push  ebx
0x00000590 <+14>:   push  ecx
0x00000591 <+15>:   call  0x450 <__x86.get_pc_thunk.bx>
0x00000596 <+20>:   add   ebx,0x1a3e
0x0000059c <+26>:   mov   eax,ecx
0x0000059e <+28>:   mov   eax,DWORD PTR [eax+0x4]
<SNIP>
```

We don't have to change the display mode manually continually. We can also set this as the default syntax with the following command.

Change GDB Syntax

Stack-Based Buffer Overflow

```
student@nix-bow:~$ echo 'set disassembly-flavor intel' > ~/.gdbinit
```

If we now rerun GDB and disassemble the main function, we see the Intel syntax.

GDB - Intel Syntax

Stack-Based Buffer Overflow

```
student@nix-bow:~$ gdb ./bow32 -q

Reading symbols from bow...(no debugging symbols found)...done.
(gdb) disassemble main

Dump of assembler code for function main:
    0x00000582 <+0>:    lea     ecx,[esp+0x4]
    0x00000586 <+4>:    and     esp,0xffffffff0
    0x00000589 <+7>:    push   DWORD PTR [ecx-0x4]
    0x0000058c <+10>:   push   ebp
    0x0000058d <+11>:   mov     ebp,esp
    0x0000058f <+13>:   push   ebx
    0x00000590 <+14>:   push   ecx
    0x00000591 <+15>:   call   0x450 <__x86.get_pc_thunk.bx>
    0x00000596 <+20>:   add     ebx,0x1a3e
    0x0000059c <+26>:   mov     eax,ecx
    0x0000059e <+28>:   mov     eax,DWORD PTR [eax+0x4]
    0x000005a1 <+31>:   add     eax,0x4
    0x000005a4 <+34>:   mov     eax,DWORD PTR [eax]
    0x000005a6 <+36>:   sub     esp,0xc
    0x000005a9 <+39>:   push   eax
    0x000005aa <+40>:   call   0x54d <bowfunc>
    0x000005af <+45>:   add     esp,0x10
    0x000005b2 <+48>:   sub     esp,0xc
    0x000005b5 <+51>:   lea     eax,[ebx-0x1974]
    0x000005bb <+57>:   push   eax
    0x000005bc <+58>:   call   0x3e0 <puts@plt>
    0x000005c1 <+63>:   add     esp,0x10
    0x000005c4 <+66>:   mov     eax,0x1
    0x000005c9 <+71>:   lea     esp,[ebp-0x8]
    0x000005cc <+74>:   pop     ecx
    0x000005cd <+75>:   pop     ebx
    0x000005ce <+76>:   pop     ebp
    0x000005cf <+77>:   lea     esp,[ecx-0x4]
    0x000005d2 <+80>:   ret

End of assembler dump.
```

The difference between the **AT&T** and **Intel** syntax is not only in the presentation of the instructions with their symbols but also in the order and direction in which the instructions are executed and read.

Let us take the following instruction as an example:

Stack-Based Buffer Overflow			
0x0000058d	<+11>:	mov	ebp, esp

With the Intel syntax, we have the following order for the instruction from the example:

Intel Syntax

Instruction	Destination	Source
mov	ebp	esp

AT&T Syntax

Instruction	Source	Destination
mov	%esp	%ebp

VPN Servers

⚠ Warning: Each time you "Switch", your connection keys are regenerated and you must re-download your VPN connection file.

All VM instances associated with the old VPN Server will be terminated when switching to a new VPN server.

Existing PwnBox instances will automatically switch to the new VPN server.

US Academy 6

 Recommended

Medium Load

PROTOCOL

UDP 1337 ☐ TCP 443

DOWNLOAD VPN CONNECTION FILE



Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

IN

75ms

 Terminate Pwnbox to switch location

Start Instance

∞ / 1 spawns left

Waiting to start...


☐

Enable step-by-step solutions for all questions 



Questions

Answer the question(s) below to complete this Section and earn cubes!

 Download VPN Connection File

Target(s): [Click here to spawn the target system!](#)

 SSH to with user "htb-student" and password "HTB_@cademy_stdnt!"

+ 1 

At which address in the "main" function is the "bowfunc" function gets called?

Submit your answer here...

+10 Streak pts

Submit

Previous

Next

Go to Questions

Table of Contents

Introduction

- Buffer Overflows Overview
- Exploit Development Introduction
- CPU Architecture

Fundamentals

- Stack-Based Buffer Overflow
- CPU Registers

Exploit

- Take Control of EIP
- Determine the Length for Shellcode
- Identification of Bad Characters
- Generating Shellcode
- Identification of the Return Address

Proof-Of-Concept

- Public Exploit Modification
- Prevention Techniques and Mechanisms

Skills Assessment

- Skills Assessment - Buffer Overflow

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left