LINUX EXPLOIT DEVELOPMENT FOR BEGINNERS

STEP-BY-STEP GUIDE TO BUFFER OVERFLOWS IN KALI LINUX

MARCUS OREN

Linux Exploit Development for Beginners:

Step-By-Step Guide to Buffer Overflows in Kali Linux

By Marcus Oren

Contents

◆◆◆

Introduction

I wrote this series of book because I couldn't find anything else that was in bite-size chunks that methodically walked me through how to analyze compiled binaries, recognize bugs in software, fuzz, do dynamic analysis and write the exploit for various types of vulnerabilities in Linux applications. Don't get me wrong, lots of websites and blogs exist, and some companies will sell you several days of training for thousands of dollars, and others companies will sell you a pdf, some videos and access to a virtual range to get you started and then you're left to fend for yourself. I didn't like these options, and most were too expensive. I kept searching and searching for something that was affordable yet still provided the same level of information as the alternatives. Nothing really existed so, I said screw it, I'll write my own.

I will show you how to install some tools in your research environment, how to write shitty code (this is a very important skill to learn…lol), how to compile and break the shitty code you wrote. Next, I will show you how to write simple exploits to abuse the vulnerabilities you learned about. I found this way of doing things helped me to understand the exploit development process. Keep in mind everything starts off simple in order to allow you to lay the foundation upon which you will continue to build over time and through experimentation and invariably many failures. Keep this point in mind. This isn't a quick process and you won't be able to get a job doing vulnerability research after 3 or 4 weeks or maybe even 3 or 4 months. You will need to keep practicing, keep failing, keep learning, and keep doing. The people that just want to read about this will never be good at it, you must put your hands on the keyboard time and time again.

The world needs more people that can find vulnerabilities in software. There are not enough because it is not easy. In fact, it is tough especially if you don't have any formal education in software engineering or programming in general. But don't worry, many if not all the best security researchers and bug hunters don't have a college degree or if they do it's in something totally unrelated to finding bugs and writing exploits. It can be learned, and it all starts with analyzing compiled binaries and looking at source code to see how they work within the context of the Operating System (OS) on which they run. So, as you can imagine this also implies the need to eventually understand the OS architecture and intricacies of memory.

Chapter 1: Tool Installation and Configuration

◆◆◆

I've included this section from my first book in the series (**Linux Exploit Development for Beginners: Step-By-Step Guide to Binary Analysis in Kali Linux**) in case you haven't read it. Being able to set up your environment is important and can be a pain sometimes so, in this section I will tell you about the tools we will use and the commands you will need to run to install and configure them.

Many tools come preinstalled on Kali Linux, which I highly suggest you use in the beginning for this very reason. If you want to use some other version of Unix or Linux, feel free but all the explanations in this will be based on 64-bit Kali Linux Virtual Machines.

I will be using the latest rolling version of 64-bit Kali Linux for Virtualbox. We will need some tools not found in Kali and some extra libraries that will allow us to compile 32-bit applications as well as 64-bit applications.

The debugger we will use is called gdb, and on top of gdb we will add gef which makes gdb way more powerful and easier to use. AFL is an awesome fuzzer that we won't use in this book, but you will need it for follow-on things we will do later so, might as well install it now.

◆◆◆

Let's begin:

- **apt-get install gdb**
- **git clone https://github.com/hugsy/gef.git**
- **echo 'source /root/gef/gef.py' >> ~/.gdbinit**
- **wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz**
- **tar -xvf afl-latest.tgz**
- **cd afl-2.52b/**
- **make**
- **make install**
- **echo 0 > /proc/sys/kernel/randomize_va_space**
- **apt install libc6-dev-i386**
- **init 6**

◆◆◆

I'd like to take a minute to discuss gdb and gef. gdb has been around a long time and is somewhat difficult to use until you practice a lot. Even then it's not the easiest for exploit dev purposes. But, that's where gef comes in. It's awesome. Instead of trying to explain it, I'll provide a link to the documentation and a brief excerpt from the documentation below:

"GEF (pronounced ʤɛf - "Jeff") is a kick-ass set of commands for X86, ARM, MIPS, PowerPC and SPARC to make GDB cool again for exploit dev. It is aimed to be used mostly by exploit developers and reverse-engineers, to provide additional features to GDB using the Python API to assist during the process of dynamic analysis and exploit development. It has full support for both Python2 and Python3 indifferently (as more and more distros start pushing gdb compiled with Python3 support)."

◆◆◆

If you want to know more, info can be found at https://gef.readthedocs.io/en/master/

Chapter 2: Buffer Overflows in C

◆◆◆

Buffer overflows generally exist in low level languages like C. That's not to say other languages don't have overflow vulnerabilities but for the most part, C and C++ are the primary culprits. The reason for revolves around the concept of program control and user data being mixed and because the language allows for a variety of mistakes to be made.

Writing code is not easy and when programs are large and complex it is expected that errors will be made. Sometimes compilers, in their attempts to optimize code, can introduce vulnerabilities. Developer created and compiler generated issues present opportunities for abuse. Languages like C allow for the access and manipulation of memory. This presents many opportunities for errors resulting in unintended program behavior and exploitation.

The classic stack-based buffer overflow is most often present when an unintended write operation past the end of a buffer into other areas of memory is possible. In order to exploit this vulnerability, additional steps must be taken to determine if it is possible to write some code of our choosing and to ultimately get that code to execute.

As you can imagine nothing is as simple as just writing past the end of a buffer. Modern day operating systems and compilers have a litany of built-in security mechanisms or exploit mitigation controls that you will have to learn to identify and defeat.

In the beginning we will turn off all the other security mechanisms in order to learn about the underlying vulnerability. Further down the road we can remove the training wheels and address exploit mitigation control defeats.

I prefer to avoid as much "theory" as possible, however, I feel it is beneficial to show some examples of shitty code wherein overflow vulnerabilities exist. This will give the reader more practice analyzing code (which is a useful skill in high demand).

Code analysis, bug hunting, secure code review, are all terms you will hear and **roughly** convey the same sort of meaning. These three terms connote the idea of looking at code to find problems that might lead to unintended program behavior or the exploitation of the program.

I will draw several examples from Carnegie Mellon's Software Engineering Institute's Cert C Coding Standard, 2016 Edition. Free copies available to download from their site.

◆◆◆

Our first example includes an uninitialized local variables and a failure to check input size. If a function reads the memory associated with an uninitialized variable, it will read an indeterminate value, and in this case, the sprintf() function will then copy that indeterminate value from the arbitrary memory location until it encounters a string terminator or NULL character.

Reading indeterminate data from memory means that it could be anywhere from 1 byte to unknown amount of bytes until sprint() reaches a NULL terminator. Depending on how large the value is, a buffer overflow could happen.

Below you will see that the buffer is only 24 bytes.

Uninitialized Local Variable – Buff Overflow Vulnerability

```
#include <stdio.h>
/* Get username and password from user, return -1 on error */
extern int do_auth(void);
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
    const char *error_log;  /*Uninitialized Local Variable*/
    char buffer[BUFFERSIZE];
sprintf(buffer, "Error: %s", error_log);
    printf("%s\n", buffer);
}
int main(void) {
    if (do_auth() == -1) {
          report_error("Unable to login");
          }
          return 0;
}
```

Presume for a minute that the program above was fixed and the error_log variable was initialized to the value of the input referenced by msg (void report_error(const char *msg)).

The problem now is that the string input referenced by msg in the report_error() function is not checked to make sure it doesn't exceed the buffer size. So, if its size-including spaces and the NULL terminator- exceeds the size of the buffer then an overflow condition is present. For example, say a msg input of 30 bytes is sent to the report_error() function. The value of msg is then assigned to the error_log variable, which is then used as input later in the program. However, the buffer size is only 24 bytes which means the input would overflow the buffer.

◆◆◆

Another example involves the use of the gets() function which was removed from the C11 standard. This function is dangerous because it does not check how much input is sent from stdin. Older programs or developers who are unaware of this issue may use gets(). gets() reads characters from stdin and writes them to an array or buffer until an end-of-file (EOF) or a newline character is encountered. It will then throw away the newline character and add a null character to the end of the string. In the code below we have a buffer size of 1024 bytes and a call to gets() which will not check how many characters are read from standard in. If the amount of characters read in exceed the size of the buffer, an overflow will happen.

Use of gets() – Buffer Overflow Vulnerability

```
#include <stdio.h>
#define BUFFER_SIZE 1024
void func(void) {
    char buf[BUFFER_SIZE];
    if (gets(buf) == NULL) {
          /* Handle error */
    }
}
```

The string print function. sprint() copies a string to a buffer. In the following example we see this in action, but the program does not check the length of the input string which is called *name* in this example. The string could come from another part of the program, a user, network traffic or an environment variable, so we have no way of knowing how many characters it is.

In this case sprint() copies the string name to the buffer filename which is 128 bytes. It then creates a file name from the string. The assumption that the size of name will never exceed 128 bytes is bad. Had the developer used precision in the form of sprintf(filename, **"%.120s.txt"**, name); then sprint() would have only used the first 120 characters of the *name* value.

Use of sprintf() – Buffer Overflow Vulnerability

```
#include <stdio.h>
void func(const char *name) {
    char filename[128];
    sprintf(filename, "%s.txt", name);
}
```

◆◆◆

In this final example our buffer is declared with "**char bof_buf[15];** " and is only 15 bytes. The next function call in the program is "**strcpy(bof_buf, argv[1]);**" where the strcpy() function copies the user supplied input into the bof_buf char buffer.Inadequate space allocation for copying command line arguments can result in vulnerabilities because the attacker can send whatever he wants at the command line and in this case there is no length check on the user input.

The function strcpy() does not check the length of what it copies into the buffer and therefore presents an opportunity for us to send more than 15 bytes and write outside the bounds of the buffer.

Once we can write outside the bounds of the original char buffer, we might be able to write a memory address and modify the flow of the program to point to some of our code that we want the program to run.

bof.c – C Program Vulnerable to Buffer Overflow

```
/* bof.c - Simple Buffer Overflow Program */
#include <string.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
        char bof_buf[100];
    strcpy(bof_buf, argv[1]);

/* Print contents of the buffer to standard out */
printf("You provided the following input:%s\n", bof_buf);

        return 0;
}
```

You won't always have access to source code in the applications you're working with but in the beginning, it helps you understand what's happening. We are going to compile the code with debugging symbols (-g) as an added benefit to assist our understanding of the application when we begin to look at it in a debugger.

Additionally, we will compile it with no exploit mitigation controls. More on these in another book when we learn to defeat them. Feel free to look up what the arguments below do when passed to gcc.

◆◆◆

Type out the program in a text editor and then compile the program with the following command. Then run the program as shown with various command line inputs and watch what happens.

gcc -g -m32 -mpreferred-stack-boundary=2 -fno-stack-protector -z execstack -no-pie -o bof bof.c

Run the program as shown sending various command line inputs and watch what happens.

◆◆◆

Running the Compiled bof.c Program

```
*****************************
    root@kali:~# ./bof
    Instructions: ./bof <Provide some input>
    root@kali:~# ./bof Hello
    You provided the following input:Hello
    root@kali:~# ./bof 0123456789ABCD
    You provided the following input:0123456789ABCD
*****************************
```

The program seems to work as intended when it gets user input that does not exceed the buffer size.

Let's make sure we correctly compiled this program before we start trying to find the buffer overflow vulnerability in the next Chapter by sending it more data than the buffer can handle.

We need to start a gdb session on our compiled binary "bof".

Then run the **checksec** command to see that we compiled it correctly and determine what exploit mitigation controls are or are not in play. As mentioned earlier we don't want any special mitigations so we should just see X's when we run the command. Don't worry about the RelRO part for now. We see the other categories are not enabled which is what we wanted.

Run Checksec

```
*****************************
    root@kali:~# gdb -q bof
    GEF for linux ready, type `gef' to start, `gef config' to configure
    79 commands loaded for GDB 8.3.1 using Python engine 3.7
    [*] 1 command could not be loaded, run `gef missing` to know why.
    [+] Configuration from '/root/.gef.rc' restored
    Voltron loaded.
    Reading symbols from bof...
    gef➤  checksec
    [+] checksec for '/root/bof'
    Canary               : ✗
    NX                   : ✗
    PIE                  : ✗
    Fortify              : ✗
    RelRO                : Partial
*****************************
```

◆◆◆

Chapter 2: Vocabulary

◆◆◆

Uninitialized local variable – a variable inside of a function that was not assigned a specific value by the developer and as such will contain whatever is in memory at the location the variable value points to. This could be benign, or it could be information from another program or even sensitive data.

Standard In (stdin) – Stream data as input from the command line, a file or something else.

Standard Out (stdout) – Stream to which a program writes its output, this can be the terminal that started the program.

Standard Error (stderr) – Stream to which programs send error messages, this can be the terminal that started the program.

Buffer overflow – a condition where data is written past the end of a buffer into contiguous memory

Segmentation fault - Computer’s way of telling us that the application tried to access restricted memory or a memory address that did not actually exist. If you’re a programmer this is bad if you’re trying to write exploits, this is good because it means you have somehow caused the application to access memory it was not supposed to.

printf() - The functions in the printf() family produce output according to a format as described below. The function printf() and writes output to stdout, the standard output stream; fprintf() writes output to the given output stream; sprintf(), snprintf(), write to the character string *str*.

gets() - Reads characters from the standard input (stdin) and stores them as a C string into *str* (pointer to an array of chars where the string was copied) until a newline character or the end-of-file is reached.

Pointer - A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location where that other variable is stored.

Canary – A value placed on the stack between a buffer and other important information to catch buffer overflow attacks. If an attacker overflows a buffer, the overwrite will corrupt the canary, which creates an overflow alert.

NX – Method to make certain parts of memory as No Execute. If “NX” is on, the stack will be non-executable.

PIE – Position independent executable is added protection on top of ASLR to add more entropy to memory addresses of things in a program. This makes it harder to find a specific memory address to use in an exploit.

Chapter 2: Resources

https://www.tutorialspoint.com/cprogramming/c_scope_rules.htm (local vars)

https://en.wikipedia.org/wiki/Standard_streams (stdin, stdout, stderr)

https://en.wikipedia.org/wiki/Buffer_overflow (BOF)

https://kb.iu.edu/d/aqsj (segmentation fault)

https://linux.die.net/man/3/printf (printf family of functions man page)

https://www.geeksforgeeks.org/fgets-gets-c-language/ (gets() and fgets() functions)

https://www.tutorialspoint.com/cprogramming/c_pointers.htm (Pointers in C)

https://en.wikipedia.org/wiki/Buffer_overflow_protection (Canaries)

https://en.wikipedia.org/wiki/NX_bit

https://eklitzke.org/position-independent-executables (PIE)

Chapter 3:Buffer Overflow & Dynamic Analysis

In Chapter 2, we learned about buffer overflows, we wrote and compiled some vulnerable code and we sent some input to our program to see how it acted. Now that we have a working binary let's begin to debug it and try to find out more about how it is vulnerable by doing some dynamic analysis.

Let's begin by debugging the program with gdb and gef. I will not include all the output from every command because it's not needed. I will, however, attempt to provide output that is relevant to what we are trying to learn.

Your output after running these commands will have additional things that I don't include in the book. But what I do include should be in your output somewhere. Make sure you scroll down or up before getting frustrated and assuming something went wrong.

We will begin by debugging the application with gdb and then disassembling the main function. By this I mean we will see what the debugger tells us the machine code and instruction mnemonics in assembly look like from a compiled application that we are debugging.

This is helpful because it allows us to get into the guts of the program and really figure it out. We first run gdb and tell it to start debugging the program **bof**, then we run the program and give it some input. The program should act normally, print out the message and exit normally.

◆◆◆

Debug bof with GDB and GEF

```
*****************************
root@kali:~# gdb -q bof
GEF for linux ready, type `gef' to start, `gef config' to configure
79 commands loaded for GDB 8.3.1 using Python engine 3.7
[*] 1 command could not be loaded, run `gef missing` to know why.
[+] Configuration from '/root/.gef.rc' restored
Voltron loaded.
Reading symbols from elf1...
gef➤  run Marcus
Starting program: /root/bof Marcus
You provided the following input:Marcus
[Inferior 1 (process 1753) exited normally]
*****************************
```

Now we can really start to dig in. We begin by disassembling the main function so we can see the assembly instructions. At first this will seem overwhelming but stick with it. We only need to look at a few things in the beginning and we will build on those as time goes by. Can you identify the call to the printf() function and the call to the strcpy() function? If you refer back to the original code, you will see these function calls.

Disassemble Main Function of bof

```
*****************************

 gef➤  disas main
 Dump of assembler code for function main:
0x08049172 <+0>:push   ebp
0x08049173 <+1>:mov    ebp,esp
```

```
0x08049175 <+3>:push   ebx
0x08049176 <+4>:sub    esp,0x64
0x08049179 <+7>:call   0x80490b0 <__x86.get_pc_thunk.bx>
0x0804917e <+12>:add    ebx,0x2e82
0x08049184 <+18>:mov    eax,DWORD PTR [ebp+0xc]
0x08049187 <+21>:add    eax,0x4
0x0804918a <+24>:mov    eax,DWORD PTR [eax]
0x0804918c <+26>:push   eax
0x0804918d <+27>:lea    eax,[ebp-0x68]
0x08049190 <+30>:push   eax
0x08049191 <+31>:call   0x8049040 <strcpy@plt>
0x08049196 <+36>:add    esp,0x8
0x08049199 <+39>:lea    eax,[ebp-0x68]
0x0804919c <+42>:push   eax
0x0804919d <+43>:lea    eax,[ebx-0x1ff8]
0x080491a3 <+49>:push   eax
0x080491a4 <+50>:call   0x8049030 <printf@plt>
0x080491a9 <+55>:add    esp,0x8
0x080491ac <+58>:mov    eax,0x0
0x080491b1 <+63>:mov    ebx,DWORD PTR [ebp-0x4]
0x080491b4 <+66>:leave
0x080491b5 <+67>:ret
End of assembler dump.
*****************************
```

Since we are working with a simple program and we have access to the source code we know there are two functions that may be of interest.

We want to find out how the program handles out input and where it goes. When we do this, we can look at a snapshot of what's going on in memory and learn a lot of information. Instead of using a name for input we are going to use something easier to find when we are scanning for information.

We are going to pretend for a moment that we don't know how big the buffer is and provide 150 (arbitrary number) random letters to the program as input. We can use the pattern create tool in gef to generate a pattern of n bytes of our choosing. Then we run the program with the pattern we created as input.

◆◆◆

Pattern Create / Overflow

```
*****************************
gef➤  pattern create 150
[+] Generating a pattern of 150 bytes
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaa
[+] Saved as '$_gef1'
gef➤                                                                              run
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaav
Starting                          program:                               /root/bof
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaa
You                provided                  the                          following
input:aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaa

Program received signal SIGSEGV, Segmentation fault.
0x62616163 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────── registers ─────
$eax   : 0x0
```

```
$ebx   : 0x6261617a ("zaab"?)
$ecx   : 0x7fffff48
$edx   : 0xf7fb3010  →  0x00000000
$esp   : 0xffffd2e0  →  "daabeaabfaabgaabhaabiaabjaabkaablaabma"
$ebp   : 0x62616162 ("baab"?)
$esi   : 0xf7fb1000  →  0x001d6d6c
$edi   : 0xf7fb1000  →  0x001d6d6c
$eip   : 0x62616163 ("caab"?)
$eflags: [zero carry parity ADJUST SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
──── stack ────
0xffffd2e0│+0x0000: "daabeaabfaabgaabhaabiaabjaabkaablaabma" ← $esp
0xffffd2e4│+0x0004: "eaabfaabgaabhaabiaabjaabkaablaabma"
0xffffd2e8│+0x0008: "faabgaabhaabiaabjaabkaablaabma"
0xffffd2ec│+0x000c: "gaabhaabiaabjaabkaablaabma"
0xffffd2f0│+0x0010: "haabiaabjaabkaablaabma"
0xffffd2f4│+0x0014: "iaabjaabkaablaabma"
0xffffd2f8│+0x0018: "jaabkaablaabma"
0xffffd2fc│+0x001c: "kaablaabma"
─── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x62616163
─────── threads ────
[#0] Id 1, Name: "bof", stopped 0x62616163 in ?? (), reason: SIGSEGV
```

We now see errors messages saying the program **[!] Cannot access memory at address 0x62616163.** That's because that is not a real memory address it's part of the input we sent the program.

We need to take the address **0x62616163** and feed it to pattern offset in order to determine how much of an offset we need to overflow the buffer and to control the instruction pointer or EIP. In this case it returns the message **"[+] Found at offset 108 (little-endian search) likely"** as seen below.

We can use this information to create another pattern of size 108 bytes but add an additional four characters at the end so we can easily identify any overwrites that occur. Let's use a capital A; in hex this is represented as \x41\. We will see our input of capital A loaded into memory.

Finding the Offset
****

```
gef➤  pattern offset 0x62616163
[+] Searching '0x62616163'
[+] Found at offset 108 (little-endian search) likely

gef➤  pattern create 108

[+] Generating a pattern of 108 bytes
```

```
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaa

[+] Saved as '$_gef0'

gef➤ run
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaav

Starting program: /root/bof
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaa

You provided the following
input:aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaaua

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
───── registers ──────
$eax   : 0x0
$ebx   : 0x6261617a ("zaab"?)
$ecx   : 0x7ffffff6e
$edx   : 0xf7fb3010  →  0x00000000
$esp   : 0xffffd300  →  0x00000000
$ebp   : 0x62616162 ("baab"?)
$esi   : 0xf7fb1000  →  0x001d6d6c
$edi   : 0xf7fb1000  →  0x001d6d6c
$eip   : 0x41414141 ("AAAA"?)
$eflags: [zero carry PARITY ADJUST SIGN trap INTERRUPT direction overflow RESUME virtualx86
identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────── code:x86:32 ──────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x41414141
────── threads ──────
[#0] Id 1, Name: "bof", stopped 0x41414141 in ?? (), reason: SIGSEGV
*****************************
```

◆◆◆

The program crashes because we overflowed a buffer and overwrote some control data on the stack. That is a lot of information to digest but if you look closely you will see familiar things like some of the letters and the capital A data we provided as input to the program all over memory.

The important thing to identify and understand here is when we look at the contents of EBP and EIP they both contain at least some of our malicious input. This means we were able to overflow the buffer and write into parts of memory we shouldn't be allowed to write to.

```
$eax   : 0x0
$ebx   : 0x6261617a ("zaab"?)
$ecx   : 0x7ffffff6e
$edx   : 0xf7fb3010  →  0x00000000
$esp   : 0xffffd300  →  0x00000000
$ebp   : 0x62616162 ("baab"?)
$esi   : 0xf7fb1000  →  0x001d6d6c
$edi   : 0xf7fb1000  →  0x001d6d6c
$eip   : 0x41414141 ("AAAA"?)
```

This is the beginning of the process of finding a buffer overflow vulnerability and gaining control of program execution. In this case we have overwritten EIP with arbitrary data (AAAA), but we could overwrite it with a memory address to another location that holds code that we want to execute.

◆◆◆

Chapter 3: Vocabulary

◆◆◆

Stack – an area of memory that stores variables, and other information for a running process. Items are pushed onto the stack from a register and then popped off the stack into a register for storage. Every function in a program has its own stack space.

Memory Address – usually in Hex; for example **0x080491c8**

Top of the Stack (ESP) – In the section above the stack starts at address **$ebp** and grows towards lower memory in the direction of the "top" of the stack or **$esp.**

Bottom of the Stack (EBP) – this is the beginning of the stack frame, it's like the anchor of the specific stack frame you're looking at.

ESP – Extended Stack Pointer aka top of the stack

EBP – Extended Base Pointer aka bottom of the stack or the beginning of the stack

EAX – Extended Accumulator Register aka a special temporary place in memory to store data before pushing onto the stack. For example, storing a string as in **$eax : 0x0804a024** → **"%s, today we are going to analyze this 32-bit bina[...]"**

◆◆◆

Chapter 3: Resources

◆◆◆

https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709

https://stackoverflow.com/questions/15020621/what-is-between-esp-and-ebp

Conclusion

◆◆◆

In **Chapter 1** we installed some tools and some additional libraries to assist us in compiling and analyzing both 64-bit and 32-bit binaries.

◆◆◆

In **Chapter 2** we learned about a variety of ways buffer overflow vulnerabilities can happen. We looked at specific C functions and specific examples of code that is vulnerable. We reviewed code that contained uninitialized local variables, the gets() function and the sprint() function.

We wrote a small program in C and compiled it with gcc. We ran the program and provided some command line input to make sure it was a functional executable. We then started **gef** and ran **checksec** to make sure we compiled it correctly and without all the modern exploit mitigation controls for now. The vocabulary section of Chapter 2 further explained some of the exploit mitigation controls and other concepts for this chapter.

◆◆◆

In **Chapter 3** we learned some basic dynamic analysis with gdb/gef. We used pattern create to generate some random data to send to our program so we could look at it in memory.

After we ran the program with the data we learned that it crashed because the next instruction was not located at an actual memory address because we had overflowed the buffer and overwrite the address to the next instruction.

We then used pattern offset to learn how much data it qould take to overflow the buffer. We added 4 additional bytes as a proof of concept to overwrite EIP.

When we ran the program again with 108 bytes + 4 bytes to overwrite EIP we saw that EIP had been successfully overwritten with capital A or x41.

Please leave a review on Amazon if you have a minute. Let us know what you think and how we can improve.