moodle.univ-smb.fr

INFO001_INFO: Le TP sur le Buffer Overflow

12–15 minutes

Accès à la machine virtuelle

Se connecter en ssh à la machine virtuelle : ssh seed@IP_VM , mot de passe : dees

L'adresse IP de la VM vous sera fournie par l'enseignant.

The vulnerable program

We have placed a C program `wisdom-alt.c` in the `projects/1` directory in the virtual machine. Type `cd projects/1` to change into this directory. If you type `ls` you will see that also in this directory is a compiled version of the program, called `wisdom-alt`. This executable was produced by invoking `gcc -fno-stack-protector -ggdb -m32 wisdom-alt.c -o wisdom-alt` (in case you accidentally delete it and need to reproduce it).

Running the program

The program reads data from the `stdin` (i.e., the keyboard) and writes to `stdout` (i.e., the terminal). You can run the program by typing `./wisdom-alt` on the command prompt. When we do this, we see the following greeting:

```
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

At this point, it is waiting for the user to type something in. Typing the number 1 allows you to "receive wisdom" and typing 2 allows you to "add wisdom". Extending the interaction, suppose we type 1 (and a carriage return).

```
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
```

```
1. Receive wisdom
2. Add wisdom
Selection >1
no wisdom
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Notice that it outputs no wisdom and then repeats the greeting. Now if we type 2 we can try to add some wisdom; here's what happens:

```
Selection >2
Enter some wisdom
```

Now the program is waiting for the user to type something in. Suppose we type in `sleep is important` and press return. Then we will get the standard greeting again. If we type 1 at that point we will get the following:

```
Selection >2
Enter some wisdom
sleep is important
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
sleep is important
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

We can continue to add wisdom, by typing 2. For example, if we did this sequence again, with the entry `exercise is useful`, we would get:

```
Selection >2
Enter some wisdom
exercise is useful
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
sleep is important
exercise is useful
Hello there
```

```
1. Receive wisdom
2. Add wisdom
Selection >
```

We can keep doing this as long as we like. We can terminate interacting with the program by typing control-D.

Crash!

This program is vulnerable to a buffer overflow. It is easy to see there is a problem, by typing in something other than 1 or 2. For example, type in 156.

```
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >156

Segmentation fault
```

In fact, the program has (at least) *two* vulnerabilities; the above is demonstrating one of them, but there is one other. Your job in this lab is to find and exploit both vulnerabilities. The lab will guide you through steps to do so, and you will answer questions in the on-line quiz to show that you took each of these steps.

Exploiting the program

We are now going to show you some tools you'll need to exploit this program.

Entering binary data

To exploit the program, you will need to enter non-printable characters, i.e., *binary data*. To input binary data to the program, use the following command line instead:

```
./runbin.sh
```

Then we can type in binary-format strings (e.g., with hex escaping). For example:

```
seed@seed-desktop:~/projects/1$ ./runbin.sh
Hello there
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
```

```
\x41\x41
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
AA
```

In the above, `\x41\x41` represents two bytes, defined in hexadecial format. 41 in hex is 65 in decimal, which in ASCII is the character A. As a result, when we ask for wisdom, the program prints AA. Entering something like `\x07` would be a byte 7. This is not a printable character, but is the "bell". So when it "prints," you would actually hear a sound (if sound were enabled on this VM).

To exploit the program, you will have to enter sequences of binary bytes that contain addresses, which are 4-byte (i.e., 32-bit) words on the VM. The x86 architecture is "little-endian", meaning that the bytes in a word are stored from least significant to most significant. That means that the hexadecimal address 0xabcdef00 would be entered as individual bytes in reverse order, i.e., \x00\xef\xcd\xab. Here is a refresher on endianness, if you need it.

Note: `runbin.sh` is a shell script that is just a wrapper around the following code:

```
while read -r line; do echo -e $line; done |
./wisdom-alt
```

This is what is converting the hex digits into binary before passing them to the `wisdom-alt` program. When carrying out the lab, please use the `runbin.sh` program, and not the above code directly, or your answers may be slightly off, as discussed at the end.

Using GDB

To exploit the program, you will have to learn some information about how it is laid out in memory. You can find out this information using the gdb program debugger. You can *attach* gdb to your running program, and then use it to print information about the state of that program, and step through executions of that program.

To attach gdb to `wisdom-alt`, you should first invoke `./runbin.sh`, and then, in a separate terminal, from the `projects/1` directory invoke the following line:

```
gdb -p `pgrep wisdom-alt`
```

The `-p` option to gdb tells it to attach to a running program with the process ID (PID) given to the option. The command `pgrep wisdom-alt` searches the process table to find the PID of the `wisdom-alt` program; this PID is then fed as the argument to `-p`. *Be warned*: If you have multiple `wisdom-alt` programs running, you may not attach to the one you expect! Make sure they are all killed (perhaps by killing and restarting the terminals you started them in) if you run into trouble. Also, be sure you use backquotes around `pgrep wisdom-alt` and not forward quotes.

Once you have connected to the process, you can start using gdb commands to start examining its state and controlling it. For example:

```
    seed@seed-desktop:~/projects/1$ gdb -p `pgrep
wisdom-alt`

GNU gdb 6.8-debian
Copyright (C) 2008 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 "i486-linux-gnu".
Attaching to process 29727
Reading symbols from /home/seed/projects
/1/wisdom-alt...done.
Reading symbols from /lib/tls/i686/cmov
/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xb7fe1430 in __kernel_vsyscall ()
(gdb)
```

This shows starting gdb and attaching it to a running `wisdom-alt` process. Then the gdb command prompt comes up. At this point, the execution of that program is paused, and we can start entering commands. For example:

```
(gdb) break wisdom-alt.c:100
Breakpoint 1 at 0x80487ea: file wisdom-alt.c,
line 100.
(gdb) cont
```

```
Continuing.
```

Here we enter a command to set a breakpoint at line 100 of `wisdom-alt.c`. Then we enter command `cont` (which is short for `continue`) to tell the program to resume its execution. In the other terminal, running `wisdom-alt` we enter 2 and press return. This causes execution to reach line 100, so the breakpoint fires, and the gdb command prompt comes up again, pausing the program in the process.

```
Breakpoint 1, main () at wisdom-alt.c:100
100             int s = atoi(buf);
(gdb) next
101             fptr tmp = ptrs[s];
(gdb) print s
$1 = 2
(gdb) print &r
$2 = (int *) 0xbffff530
(gdb) cont
Continuing.
```

Then, we control the program by stepping using "next", which executes the current line of code, proceeding to the next. Then we print the contents of variable `s` with "print", and it displays the value we entered in the other terminal. Then we continue execution by entering "cont". In the other terminal we see the prompt to enter some wisdom.

When you are done working with gdb (perhaps when you've terminated the other program), just type `quit` to exit.

The basic GDB commands you will want to use are those we have already demonstrated: setting break points, stepping through execution, and printing values. If you are not familiar with GDB already, a quick GDB reference is available here, and a more in depth GDB tutorial is available here. You would find it helpful to be familiar with the "print", "break", and "step" commands. A full GDB user's manual is also available.

Questions

You are now ready to start your process of developing an exploit.

The first step is to identify where the vulnerabilities are. To do that you will have to look through the code of `wisdom-alt.c`. You can do this by using an editor on Linux virtual machine, like `vi` or `emacs`, both of which are installed. Alternatively you can look through the file on your own machine outside of the VM, in an editor of your choice --- the file is available here.

After looking over the code to see how it works, answer the following four questions.

1. There is a stack-based overflow in the program. What is the name of the stack-allocated variable that contains the overflowed buffer?
2. Consider the buffer you just identified: Running what line of code will overflow the buffer?
3. There is another vulnerability, *not dependent at all on the first*, involving a *non*-stack-allocated buffer that can be indexed outside its bounds (which, broadly construed, is a kind of buffer overflow). What variable contains this buffer?
4. Consider the buffer you just identified: Running what line of code overflows the buffer?

Now use GDB to examine the running the program and answer the following questions. These questions are basically going to walk you through constructing an exploit of the non-stack-based overflow vulnerability you just identified. We will do less "hand holding" when asking about exploiting the stack-allocated buffer.

Once you have answers for all of the questions here, you can enter them on the Coursera site for full credit.

Important: When carrying out the lab, you must follow the instructions given above **exactly** for running the program (using `runbin.sh`) and using GDB (attaching to `wisdom-alt` *in a separate terminal*, and *not* running gdb `./wisdom-alt`) or else the answers you get may not match the ones we are expecting. In particular, the addresses of stack variables may be different. These addresses might also be different if you have altered any environment variables in the Ubuntu terminals.

On to the first exploit:

5. What is the address of `buf` (the local variable in the main function)? Enter the answer in either hexadecimal format (a `0`x followed by 8 "digits" 0-9 or a-f, like `0xbfff0014`) or decimal format. Note here that we want the address of `buf`, not its contents.
6. What is the address of `ptrs` (the global variable) ? As with the previous question, use hex or decimal format.
7. What is the address of `write_secret` (the function) ? Use hex or decimal.
8. What is the address of p (the local variable in the `main` function) ? Use hex, or decimal format.

9. What input do you provide to the program so that `ptrs[s]` reads (and then tries to execute) the contents of local variable p instead of a function pointer stored in the buffer pointed to by `ptrs`? You can determine the answer by performing a little arithmetic on the addresses you have already gathered above -- be careful that you take into account the size of a pointer when doing pointer arithmetic. If successful, you will end up executing the `pat_on_back` function. Enter your answer as an unsigned integer.
10. What do you enter so that `ptrs[s]` reads (and then tries to execute) starting from the 65th byte in `buf`, i.e., the location at `buf[64]`? Enter your answer as an unsigned integer.
11. What do you replace `\xEE\xEE\xEE\xEE` with in the following input to the program (which due to the overflow will be filling in the 65th-68th bytes of `buf`) so that the `ptrs[s]` operation executes the `write_secret` function, thus dumping the secret? (Hint: Be sure to take endianness into account.) `771675175\x00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA \xEE\xEE\xEE\xEE`

    Now let's consider the other vulnerability:
12. Suppose you wanted to overflow the `wis` variable to perform a stack smashing attack. You could do this by entering 2 to call `put_wisdom`, and then enter enough bytes to overwrite the return address of that function, replacing it with the address of `write_secret`. How many bytes do you need to enter prior to the address of `write_secret`?

    To work out the answer here, you might find it useful to use the GDB `backtrace` command, which prints out the current stack, and the `x` command, which prints a "hex dump" of the bytes at a given address. For example, by typing `x/48xw $esp` you would print out 48 words (the `w`) in hexadecimal format (the `x`) starting at the address stored in register `$esp`.

    Arrêt de la machine virtuelle

    sudo halt -p