Lab 05 - Application Security

Resources

- Buffer overflow explained [https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/]
- Shellcode explained [https://dhavalkapil.com/blogs/Shellcode-Injection/]

Setup

- Open a lab VM instance on openstack [https://cloud-controller.grid.pub.ro], image: ***ISC 2020***, flavor: ***m1.small***, availability zone: ***any***;
- Please record your screen using asciinema [https://www.mankier.com/1/asciinema] while working:

```
# install the package
$ sudo apt update -y && sudo apt install -y asciinema
# start recording (optionally, use --append to append to an existing one)
$ asciinema rec [--append] lab05_${LDAP_USERNAME}.cast

# echo your name in the terminal
$ echo "Andrei Popescu"
# work... then:
# stop recording
$ exit

# upload recording
$ ASCIINEMA_API_URL=https://asciinema.cs.pub.ro asciinema upload lab05_${LDAP_USERNAME}.cast
```

- Install the `libc6-dev-i386` library:

```
$ sudo apt install libc6-dev-i386
```

- Install the gdb peda plugin:

```
$ git clone https://github.com/longld/peda.git ~/peda
(...)
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
(...)
```

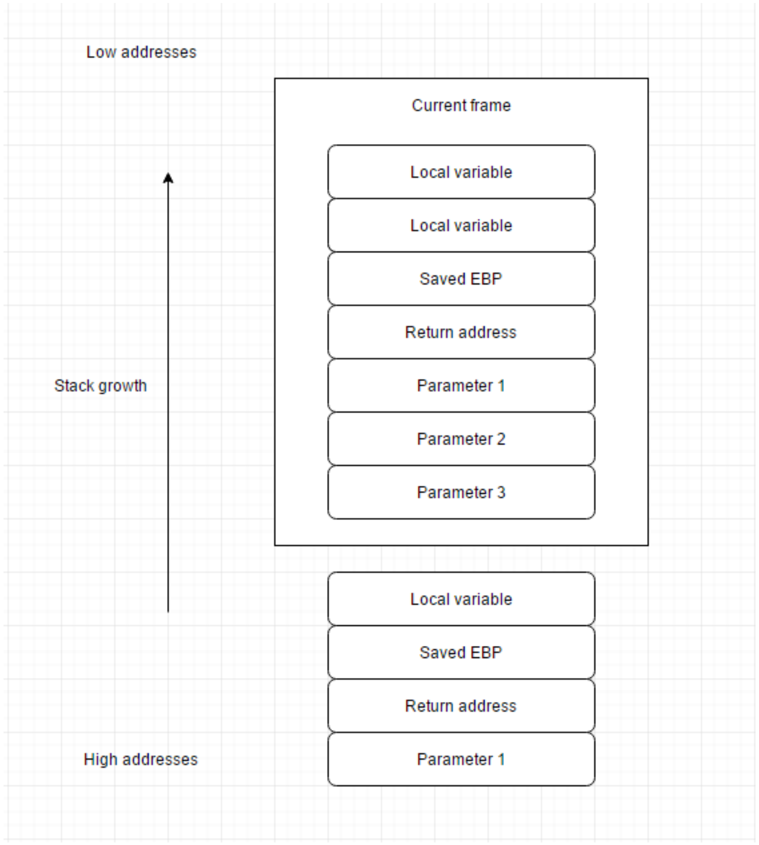
To check if everything is OK, run the command `gdb` with no arguments. The prompt should be similar to this:

```
$ gdb
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
....
gdb-peda$
```

Type `q` to exit gdb. We are using gdb peda instead of the classic gdb because it is much more user friendly. We hope you'll like it ;)

- Don't forget to submit the lab's feedback form [https://forms.office.com/r/GZzRJVqQuy]. Double check at form responses [https://ctipub-my.sharepoint.com/:x:/g/personal/mihai_chiroiu_upb_ro/Ee4pZApRKA5Iq9JgR2r652QB0FVL4J9EFtTBtva3jX-1Lw?e=ylQ7lu] ;)

Overview



This representation of the stack is valid for 32 bit programs. The calling convention is to save the parameters on the stack. To find out what's different for a 64 bit program check this website [https://www.systutorials.com/x86-64-calling-convention-by-gcc/]

A buffer overflow occurs when data written to a buffer overruns its boundary and overwrites adjacent memory locations, due to insufficient bounds checking.

GDB tutorial

GDB peda

Loading a program

In order to start debugging using GDB, you need to specify the program to be inspected. There are two options for doing this:

- When launching GDB:

```
student@host$ gdb prog
```

- After launching GDB:

```
student@host$ gdb
(...)
gdb-peda$ file prog
Reading symbols from prog...done.
```

Once the debugging symbols from the executable were loaded, you can start executing your program using the `run` command.

```
gdb-peda$ run
```

You do not need the specify the full command, GDB can fill in the rest of a word in a command for you, if there is only one possibility. E.g.: `r`, `ru` and `run` are equivalent; `c`, `co`, `continue` are equivalent.

In order to specify arguments for the debugged program, you can either:

- Specify them prior to starting the program:

```
gdb-peda$ set args arg1 arg2
```

- Specify them when starting the program:

```
gdb-peda$ run arg1 arg2
```

You do not need to specify the arguments each time: `run` with no arguments uses the same arguments used by the previous `run`, or those set by the `set args` command.

Breakpoints

Breakpoints represent places in your program where the execution should be stopped. They are added using the break command and its variants [http://www.delorie.com/gnu/docs/gdb/gdb_29.html]. Here are the most common usages:

- `break function` - Set a breakpoint at entry to function function. When using source languages that permit overloading of symbols, such as C++, function may refer to more than one possible place to break. See section Breakpoint menus, for a discussion of that situation.
- `break linenum` - Set a breakpoint at line linenum in the current source file. The current source file is the last file whose source text was printed. The breakpoint will stop your program just before it executes any of the code on that line.
- `break filename:linenum` - Set a breakpoint at line linenum in source file filename.
- `break filename:function` - Set a breakpoint at entry to function function found in file filename. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.
- `break *address` - Set a breakpoint at address address. You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

You can see an overview of the current breakpoints using the `info break` command.

```
gdb-peda$ info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804856d in main at buggy.c:33
2       breakpoint     keep y   0x080484d1 in print_message at buggy.c:12
3       breakpoint     keep y   0x080484d1 in print_message at buggy.c:12
```

Short for `info break` is `i b`.

In order to remove breakpoints [http://www.delorie.com/gnu/docs/gdb/gdb_32.html], you can use the `clear` or the `delete` (`d`) command. With the `clear` command you can delete breakpoints according to where they are in your program. With the `delete` command you can delete individual breakpoints by specifying their breakpoint numbers.

```
gdb-peda$ delete 2
gdb-peda$ clear buggy.c:33
Deleted breakpoint 1
```

Once you want to resume execution, you can use the `continue` command.

```
gdb-peda$ continue
Continuing.
[Inferior 1 (process 5809) exited normally]
```

Start

The `start` command is very similar to `run`, but instead of running the program until it ends (or until it crashes), it sets a breakpoint at the beginning of the main function.

```
[----------------------------------registers-----------------------------------]
EAX: 0x56556fd4 --> 0x1edc
EBX: 0x0
ECX: 0xffffd5c0 --> 0x1
EDX: 0xffffd5e4 --> 0x0
ESI: 0xffffd5c0 --> 0x1
EDI: 0x0
EBP: 0xffffd5a8 --> 0x0
ESP: 0xffffd580 --> 0xf7fbe3fc --> 0xf7fbf200 --> 0x0
EIP: 0x565555d0 (<main+31>:     mov    DWORD PTR [ebp-0x1f],0x6c6c6548)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565555c4 <main+19>:        call   0x56555616 <__x86.get_pc_thunk.ax>
   0x565555c9 <main+24>:        add    eax,0x1a0b
   0x565555ce <main+29>:        mov    esi,ecx
=> 0x565555d0 <main+31>:        mov    DWORD PTR [ebp-0x1f],0x6c6c6548
   0x565555d7 <main+38>:        mov    WORD PTR [ebp-0x1b],0x216f
   0x565555dd <main+44>:        mov    BYTE PTR [ebp-0x19],0x0
   0x565555e1 <main+48>:        sub    esp,0xc
   0x565555e4 <main+51>:        lea    edx,[ebp-0x1f]
[------------------------------------stack-------------------------------------]
0000| 0xffffd580 --> 0xf7fbe3fc --> 0xf7fbf200 --> 0x0
0004| 0xffffd584 --> 0x56556fd4 --> 0x1edc
0008| 0xffffd588 --> 0xffffd65c --> 0xffffd798 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=
0012| 0xffffd58c ("kVUV\001")
0016| 0xffffd590 --> 0x1
0020| 0xffffd594 --> 0xffffd654 --> 0xffffd782 ("/home/student/buffovf")
0024| 0xffffd598 --> 0xffffd65c --> 0xffffd798 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=
0028| 0xffffd59c --> 0xffffd5c0 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Temporary breakpoint 1, main (argc=0x1, argv=0xffffd654) at buffovf.c:16
16              char buf[] = "Hello!";
```

Let's take a look at the previous output that gdb peda prints. You can see it is seprated into 3 sections: registers, code and stack. With the original gdb you would have to manually print registers, disassemble code and inspect the stack. Thanks, God, for peda!!

Step

There might be situations when you only want to execute one line of source code, or one machine instruction from your program. This action is called a step [https://sourceware.org/gdb/onlinedocs/gdb/Continuing-and-Stepping.html] and can be categorized as follows:

- Step into - `step` or `s`: Continue running your program until control reaches a different source line, then stop it and return control to GDB. If the line you are stepping over represents a function call, this command will step inside it.
- Step over - `next` or `n`: Continue to the next source line in the current stack frame. This is similar to step, but function calls that appear within the line of code are executed without stopping.

There are also equivalent functions fo the machine instructions: stepi, nexti.

If you stepped into a function and you want to continue the execution until the function returns, you can use the `finish` command.

Printing variables and memory

No need to manually print registers anymore, but you still might need to print the content of a variable:

```
gdb-peda$ print input
$6 = 0
```

The `print` command allows you to specify the format of the output like this (you can find a full list of possible format specifiers here [https://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_9.html#SEC55]):

```
gdb-peda$ p/x $esp
$3 = 0xffffcf00
gdb-peda$ x/2x 0xffffcf00
0xffffcf00:     0xffffcf4c      0xf7fcf0f0
gdb-peda$ p/d $esp
$4 = 4294954752

gdb-peda$ p/s buf
$9 = "OK. Bye!\n"
gdb-peds$ p/x buf
$10 = {0x4f, 0x4b, 0x2e, 0x20, 0x42, 0x79, 0x65, 0x21, 0xa, 0x0}
```

Reading and modifying memory

You can use the command `x` (for "examine") to examine memory in any of several formats, independently of your program's data types.

```
x/nfu addr
```

`n`, `f`, and `u` are all optional parameters that specify how much memory to display and how to format it; `addr` is an expression giving the address where you want to start displaying memory.

- **n** - the repeat count: The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units u) to display.
- **f** - the display format: The display format is one of the formats used by print
- **u** - the unit size: The unit size is any of `b` (bytes), `h` (halfwords), `w` (words)

E.g.: Print 10 words in hexadecimal format, starting from the address of the current stack pointer.

```
gdb-peda$ x/10xw $esp
0xffffcf00:     0xffffcf58      0x4b4fdf10      0x7942202e      0x000a2165
0xffffcf10:     0xffffcf32      0xf7ffd918      0xffffcf58      0x080485b8
0xffffcf20:     0x00000000      0x080486b8
```

In order to change the value of a variable or of a specific memory area, you can use the `set` command:

```
gdb-peda$ set g=4
gdb-peda$ set {int}0x83040 = 4
```

Stack info

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

`backtrace`: Print a backtrace of the entire stack: one line per frame for all frames in the stack.

E.g.:

```
gdb-peda$ bt
#0  print_message (input=0) at buggy.c:16
#1  0x080485b8 in main () at buggy.c:38
```

It is also possible to move up or down the stack using the following commands:

- `up n`: Move n frames up the stack. For positive numbers n, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. n defaults to one.
- `down n`: Move n frames down the stack. For positive numbers n, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. n defaults to one.

Another useful command for printing information related to the current stack frame is `info frame`. This command prints a verbose description of the selected stack frame, including:

- the address of the frame
- the address of the next frame down (called by this frame)
- the address of the next frame up (caller of this frame)
- the language in which the source code corresponding to this frame is written
- the address of the frame's arguments
- the address of the frame's local variables
- the program counter saved in it (the address of execution in the caller frame)
- which registers were saved in the frame

Exercises

00. Our test program

Compile the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void wanted(int a) {
        if (a == 0xcafebabe) {
                puts("well done, you're cool!");
        } else {
                puts("at least you tried");
        }
}

void copy() {
        char name[12];

        printf("what's ur last name?\n");
        gets(name);
        printf("bye\n");
}

int main(int argc, char **argv) {
        if (argc == 1) {
                puts("Usage: %s <name>\n");
                return 1;
         }
        char buf[] = "hey";

        printf("%s, %s\n", buf, argv[1]);
        copy();

        exit(0);
}
```

like this:

```
gcc buffovf.c -o buffovf -fno-stack-protector -m32 -g
```

You may need to install `libc6-dev-i386` for 64-bit systems.

01. Run, break, step

Run the program using GDB, setting the argument "AAA". Set a breakpoint at the beginning of the `main` function. Continue execution until you hit the breakpoint. Try to reach the beginning of the `copy` function without setting another breakpoint.

Hint: use step over and step into.

After solving this exercise, don't close gdb.

02. Printing stuff

Remove the existing breakpoint and set a new one at the beginning of the `copy` function. Run again the program and continue execution until you hit the breakpoint. Print the value and the address of `name`. Print the value of it after `gets(name)` is executed.

03. ASLR

Start the execution again (do not exit GDB) and print the address of `buf` from the main function. What do you notice? Check in another tab if ASLR is enabled on your PC. What happens and how can you fix it?

Hint: http://visualgdb.com/gdbreference/commands/set_disable-randomization [http://visualgdb.com/gdbreference/commands/set_disable-randomization]

04. Address investigation

Restart gdb and run until the beginning of the `copy` function using `next` and `step` accordingly. Display stack info (`bt`, `info frame`). At what address is `name` located? At what address is the saved return address located? How many bytes of input do you need in order to overwrite the return address?

05. Buffer overflow

We want to overflow the buffer `name` from the copy() function. Run the program and provide an input so that the program crashes.

You can use `gdb$ run <program args> < <(python -c 'print("A" * 30)')` for stdin redirection directly within GDB! ;)

06. Call the "wanted" function

We want to create an attack which invokes the `wanted` function. What is the address of this function? Adjust the input so that the return address is overwritten with the address of the `wanted` function.

Use *objdump -d -M intel buffovf* to list all the addresses from the binary. Look for the address of the `wanted` function.

You can see that when using `objdump` the addresses look weird (short):

```
000005fd <wanted>:
 5fd:   55                      push   ebp
 5fe:   89 e5                   mov    ebp,esp
 600:   53                      push   ebx
```

They aren't actually real addresses, they are offsets counting the number of bytes from the beginning of the file. This happens because the program was compiled as PIC (position independent code). More details can be found here [https://codywu2010.wordpress.com/2014/11/29/about-elf-pie-pic-and-else/]. Recompile the program without PIC and PIE using `-fno-pic -no-pie` options for gcc.

```
080484b6 <wanted>:
 80484b6:       55                      push   ebp
 80484b7:       89 e5                   mov    ebp,esp
 80484b9:       83 ec 08                sub    esp,0x8
```

Use *python -c 'print "A" * NR + "\xGH\xEF\xCD\xAB"'* to generate the payload for calling the function with address 0xABCDEFGH. You have to find the value of NR.

07. Calling ''wanted'' function with the correct arguments

The `wanted` function takes an argument. Adjust the previous payload so that when calling `wanted`, the message `well done, you're cool!` is displayed.

08. Graceful exit

We can see that even if we call `wanted` with the correct arguments, the program still crashes. Let's remove any trace that we've been there. Adjust the previous payload so that the program exits without a segmentation fault.

Can you call another function after `wanted`? What would be a great function to call? Where can you get its address from? After finding out the function, look for its address using objdump, you might find something there.

09. Feedback

Please take a minute to fill in the feedback form [https://forms.gle/5Lu1mFa63zptk2ox9] for this lab.

isc/labs/05.txt · Last modified: 2021/04/05 08:44 by florin.stancu