Understanding Software Vulnerabilities: C, Debugging Assembly, and Buffer Overflows

License

Contents

General notes about the labs

Often the lab instructions are intentionally open ended, and you will have to figure some things out for yourselves. This module is designed to be challenging, as well as fun!

However, we aim to provide a well planned and fluent experience. If you notice any mistakes in the lab instructions or you feel some important information is missing, please feel free to add a comment to the document by highlighting the text and click the comment icon (), and I (Cliffe) will try to address any issues. Note that your comments are public.

The labs are written to be informative and, in order to aid clarity, instructions that you should actually execute are generally written in this colour. Note that all lab content is assessable for the module, but the colour coding may help you skip to the "next thing to *do*", but make sure you dedicate time to read and understand everything. Coloured instructions in *italics* indicates you need to change the instructions based on your environment: for example, using your own IP address.

You should maintain a **lab logbook / document**, which should include your answers to the questions posed throughout the labs (in this colour).

Preparation

As with all of the labs in this module, start by loading the latest version of the LinuxZ template from the IMS system. If you have access to this lab sheet, you can read ahead while you wait for the image to load.

> To load the image: press F12 during startup (on the boot screen) to access the IMS system, then login to IMS using your university password. Load the template image: LinuxZ (load the latest version).

Once your LinuxZ image has loaded, log in using the username and password allocated to you by your tutor.

The root password -- **which should NOT be used to log in graphically** -- is "tiaspbiqe2r" (**t**his **i**s **a** **s**ecure **p**assword **b**ut **i**s **q**uite **e**asy **2** **r**emember). Again, never log in to the desktop environment using the root account -- that is bad practice, and should always be avoided.

These tasks can be completed on the LinuxZ system. Most of this lab could be completed on any openSUSE or using most other Linux distributions (although some details may change slightly).

You may need to install 32bit development packages. On openSUSE: "`sudo zypper in gcc-32bit`".

Introduction to programming errors

It is very hard to write secure code. Small programming mistakes can result in software vulnerabilities with serious consequences. The two main categories of software flaws are those caused by:

- design problems: such as, a programmer not thinking through the kind of authentication required
- implementation problems: such as, a programmer accidentally introducing a bug by using an insecure library method or trying to store too much data into a variable

Common programming errors that lead to security flaws include:

- Memory errors and bounds checking, such as buffer overflows
- Using input without sanitising it: this can lead to flaws such as command injection
- Race conditions
- Misconfigured/used access controls and other security mechanisms
- Misuse of pointers and strings

Programming in C and Debugging Assembly

C is an important programming language. C is one of the most widely used programming languages of all time. C was designed for programming Unix, and is used for the Linux kernel, amongst many other high profile software projects. C code maps well to machine code instructions, yet is easier to maintain than assembly code. Assembly code is essentially made up of low-level instructions that get assembled into machine code instructions, which are executed by a CPU.

However, since C is a low-level language, which exposes the complexity of underlying systems (such as memory management), and since it does not have all of the security features included in newer languages, programs written in C (and C++) tend to be prone to many security flaws. This makes C a good choice when studying software security, and to get an understanding of software vulnerabilities.

Hello, world!

Lets start by writing, compiling, and running a very basic C program.

Open a terminal console.

> One way to do this is to start Konsole from KDEMenu → Applications → System → Terminal → Konsole.

Move to a new directory for our code:

```
mkdir ~/code
```

```
cd ~/code
```

Create and edit a new file “hello.c”:

```
vi hello.c
```

Reminder: Vi is ‘modal’: it has an insert mode, where you can type text into the file, and normal mode, where what you type is interpreted as commands. Press the “i” key to enter “insert mode”. Type the below C code:

```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
}
```

Exit back to “normal mode” by pressing the Esc key. Now to exit and save the file press the “:” key, followed by “wq” (write quit), and press Enter.

> Tip: alternatively if you want to copy and paste, rather than using vi, run “cat > hello.c” and paste into the terminal, then press Ctrl-D to end the file.

Compile your code:

```
gcc hello.c -g -m32 -o helloworld
```

> The arguments here are the file to compile (hello.c), the “-g” flag instructs the compiler to include debugging information, which makes examining the compiled program easier. The “-m32” flag tells gdb to generate a 32 bit executable, which will make it more likely that your system will generate similar assembly to these examples. Finally, “-o” is used to specify what filename to

output to, in this case to an executable named “helloworld”.

Confirm this has generated the compiled program “helloworld”:

```
ls -la
```

Run the program:

```
./helloworld
```

Note that in *modern* C, comments can be written as either

```
/* comment */
```

or

```
// comment
```

Edit the file to add comments describing each line:

```
// this first line of code below imports code from elsewhere,
// which is what gives us use of the printf() function
#include <stdio.h>
// this is the start of the main function,
// which is the code that starts when the program is run
int main() {
    // this line calls the printf function, and passes it
    // the string of characters “hello, world!”
    // which it prints to the standard out (console)
    // \n inserts a new line
    // ; ends a line of code
    printf("Hello, world!\n");
// this ends the main function
}
```

Compile and run your new code, confirming that it has not changed the behaviour of the program.

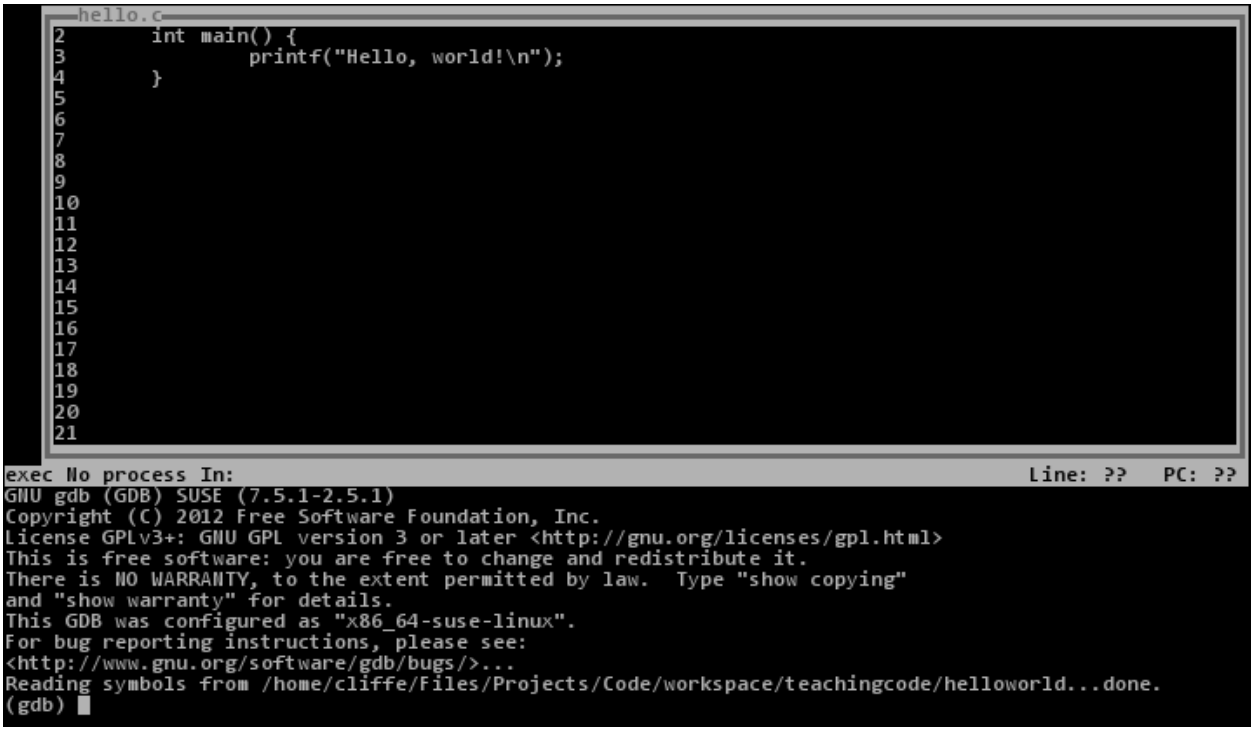
Debugging code using GDB

A debugger is a program that can be used to test other programs, and to “debug” the program by inspecting the state of the program, while stepping through code.

Start gdb with the “GDB Text User Interface” enabled. The GDB TUI is an interactive console interface that shows various views at once. The C source code, resulting assembly instructions, registers, and GDB commands can be displayed in separate text-based windows.

```
gdb -tui ./helloworld
```

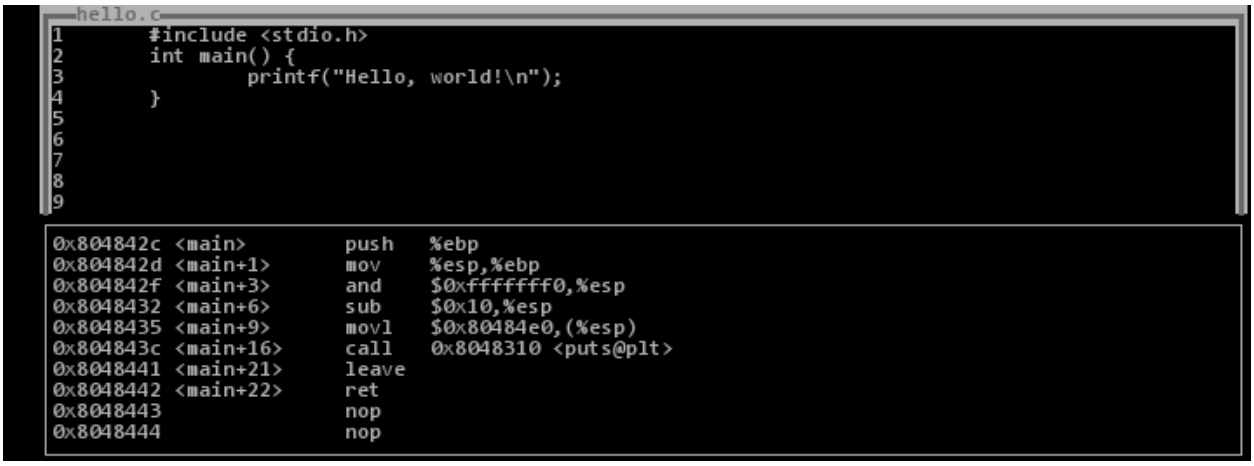
The resulting view includes the original C code, and below that the interactive console, which you type commands into.



Start by changing views, to also show assembly instructions:

```
(gdb) layout split
```

The C code and assembly instructions are now both shown at once:



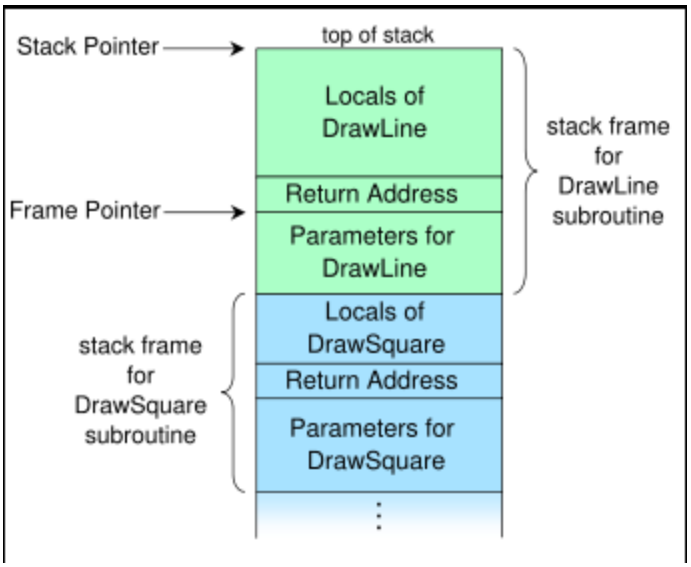
The assembly code is a very low-level representation of the program. Assembly code is close to a one-to-one representation of the actual machine code that is executed by the CPU when the program runs.

Here we can see *exactly* what actions the program will attempt. Although many programmers don't bother to understand the assembly that their C code produces, **these details are important** for understanding certain important kinds of software vulnerabilities, such as buffer overflows.

In this very small program, the *main* function only involves a few instructions.

The first few instructions in the main function make up the *function prologue*. The function prologue sets up the stack for this function.

The call stack (or simply "the stack") is an area of memory used to keep track of program execution (such as, remembering which code to return to after each function ends), it stores local variables for functions, and every time a function is called, a new stack *frame* is added to the stack for that function.



The call stack (image CC BY-SA by R. S. Shaw: http://en.wikipedia.org/wiki/File:Call_stack_layout.svg)

So, looking at the assembly for our hello world program, take some time to understand the purpose of each instruction in the function prologue:

```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x10,%esp
```

The function prologue starts by **push**ing the base pointer register onto the stack (saving it for later). Note the the stack is a FILO (first in last out) data structure (and a "pop" instruction returns data from the top of the stack). Like a stack of dinner plates, the first one you put in is the last you get out.

Next, the **mov** instruction overwrites the base pointer register (ebp) with the value from the stack pointer register (esp). This sets the bottom of the current stack frame, as stored in ebp.

> *EBP: base pointer - bottom of the current stack frame*

The **and** instruction here, is not too important, it aligns the stack pointer for optimisation. The **sub** instruction subtracts the hex value 0x10 from the stack pointer (esp), effectively reserving space for use by local variables and temporary storage. Note that hex 0x10 = 16 bytes. This space includes the area that the string "Hello, world\n" will be temporarily stored. The result is that the stack pointer has been moved to a lower memory address (if things weren't interesting enough, the stack grows downward on most platforms! - So the address at the top of the stack (esp) is a

lower number than that the bottom (ebp)).

ESP: stack pointer - top of the current stack frame

Scroll down on the C code, to the printf line, the resulting assembly code will also line up to this point.

The next few instructions provide the actual functionality of our program, by calling printf to display our message:

```
movl   $0x80484e0,(%esp)
call   0x8048310 <puts@plt>
```

The **movl** instruction copies the value $0x80484e0 into the memory location pointed to (rather than the register itself, due to the brackets) by the stack pointer register (esp); thus placing a pointer to our string onto the top of the stack. This in effect passes the address of the string “Hello, world!”, which is stored in temporary area, to a known location for the next instruction, which **call**s the printf function.

The last few instructions make up the function epilogue, which simply ends this program:

```
leave
ret
```

The focus (or “fs”) command can be used to switch to focusing on a particular view for scrolling keystrokes. You can switch focus to “src” for source, “cmd” for commands, “regs” for registers, or “asm” for the assembly view (or “next”/”prev”). Switch focus to the command window:

```
(gdb) focus cmd
```

From within the debugger you can run the program, using the command “run” (or “r”):

```
(gdb) run
```

Note that the program runs, and finishes before we have a chance to investigate.

Set a breakpoint, so the program pauses as soon as the main function starts:

```
(gdb) break main
```

Restart the program:

```
(gdb) run
```

The program will pause, and the views of code will indicate the the function prologue has completed and the code is sitting at line 3, the printf statement and movl instruction.

The next instruction is set to store some data in the memory location pointed to by the esp register. View the value in the esp register:

```
(gdb) print $esp
```

What memory address does that point to? (the x command displays a value from memory):

```
(gdb) x memory_address_from_esp
```

Now, instruct gdb to execute the next assembly instruction:

```
(gdb) stepi
```

Note that you can step through C lines using “step”, but since our whole program is one line, that would bring us to the end; so we use “stepi”, which steps one assembly instruction at a time.

View the value in that memory address again and confirm that it points to the new value:

```
(gdb) x memory_address_from_esp
```

And to confirm that this is a pointer to our greeting string:

```
(gdb) x/s 0x80484e0
```

The “/s” shows the memory as a string.

The printf function is provided by the glibc libraries, which are dynamically loaded when the program started (or in this case when GDB loaded the program). This involves quite a bit of code.

Enable the registers view:

```
(gdb) fs asm
```

```
(gdb) layout regs
```

```
(gdb) focus cmd
```

(Layout regs places the register view over the window you are not focused on, so this ensures you end up with the assembly and registers view).

Step through the printf code by running:

```
(gdb) stepi
```

Run stepi many times, and note the esp moves each time a function is called, and that each call has a function prologue. The ebp stays the same.

Another register you will notice changing a lot is **eip**, the instruction pointer. The instruction pointer refers to the location of the *next* instruction that the CPU will run. We will return to this very important detail later.

EIP: address of next instruction

When you have finished analysing the assembly of our hello world program, you can exit the debugger.

```
(gdb) quit
```

Variables and C

A variable is a buffer (some memory) for storing data, such as input from a user. In C, variables should be defined at the start of a function, before any other code.

Create and edit a new file “variables.c”:

```
vi variables.c
```

Enter the below C code:

```
#include <stdio.h>
int main() {
    int num1, num2, sum;
    num1 = 10;
    num2 = 15;
    sum = num1 + num2;
    printf("The sum of %d and %d is: %d\n", num1, num2, sum);
}
```

The first few lines define some variables that are Integers (int), that is, they only store whole numbers, and then assign them some values (=). The two numbers are added together (+), storing the result in the "sum" variable, and finally the result is printed.

Note that the use of printf is more complex in this example: the first argument ("*The sum of %d and %d is: %d*\n") is actually a *format string*, which is a template used to create the text to output. The "%d" token tells printf to take the next argument and format it as an Integer (%d: integer, %f: double, %s: string). The first %d is replaced with num1, and the next num2, and so on.

For more information about printf, you may wish to refer to:

```
man 3p printf
```

or http://en.wikipedia.org/wiki/Printf_format_string

Compile your code:

```
gcc variables.c -g -m32 -o vars
```

Confirm this has generated the compiled program "vars":

```
ls -la
```

Run the program:

```
./vars
```

Reading input and testing conditions

Create and edit a new file "input.c":

```
vi input.c
```

Enter the below C code:

```
#include <stdio.h>
int main() {
    int num1, num2, sum;
    printf("Please enter a number:\n>");
    scanf( "%d", &num1 );
    num2 = 15;
    sum = num1 + num2;
    printf("The sum of %d and %d is: %d\n", num1, num2, sum);
    if(sum > 100) {
```

```
        printf("The sum is greater than 100\n");
    } else {
        printf("The sum is less than or equal to 100\n");
    }
}
```

Compile and run your code.

The main differences here are:

- The inclusion of scanf, which reads user input into a variable, based on the format string. In this case, scanf reads an *integer* into num1, since the format string specifies "%d".
- The if statement, which branches the code, depending on whether the "sum > 100" condition is met. Other test conditions that could be used include "==" does it equal, "<" less than, "<=" less than or equal to, "!=" not equal to.

Modify your code to have the user input ***three*** numbers, and calculate the **product** (multiply). Tell the user whether the product is less than 50.

Test your code:

Does your code work with the numbers: 1, 2, and 3?

1, 2, and 0?

Does it work when the result is a real number, with decimal places? If not, why not?

What happens if the user enters a very very big or small number?

What about if the user enters a letter?

If you haven't done so already, update your code to store the input and product in a "double" (real number), rather than using "int" (integers).

Compile and test your code.

Security flaws: type safety, bounds checking, and buffer overflows

As you have seen from the testing you have just completed, it is easy to make simple mistakes that result in a program misbehaving. Sometimes simple coding errors can result in more serious problems, such as introducing security vulnerabilities.

Create and edit a new file "testerr.c":

```
vi testerr.c
```

Enter the below C code:

```
#include <stdio.h>
int main() {
    char execute [15] = "ls";
    char name [10];
    printf("What is your name?\n");
    scanf("%[^\n]s", &name);
    printf("Hello %s, executing the command %s\n"
           "The files in the dir are:\n", name, execute);
    sleep(2);
    system(execute);
}
```

> Tip: an alternative is to use the line "gets((char*)&name);" in place of the scanf line. Both versions will have similar security problems.

The line "`scanf("%[^\n]s", &name);`" reads any input, accepting anything except a newline. "`[^\n]`" is a regular expression, which represents any character except \n (a newline). As specified, `scanf()` stores into the variable "name".

Compile your code:

```
gcc testerr.c -g -m32 -o testerr
```

Run the program:

```
./testerr
```

Can you already spot the security issue in this code?

Test this program, by running it a few times and try entering different inputs.

Does it work normally if you enter your own name?

What happens if you enter an input that is very long?

Try entering:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

```
[cliffe@cliffe-pc err]$  ./testerr
What is your name?
Cliffe
hello Cliffe, executing the command ls
The files in the dir are:
testerr  testerr.c
[cliffe@cliffe-pc err]$  ./testerr
What is your name?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA, executing
 the command AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The files in the dir are:
sh: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA: command not found
Segmentation fault
```

Testing and breaking the program

As shown above, when you enter an input that is long, this program starts to behave incorrectly. It is even apparent from the above that the “execute” variable seems to have changed! Woah… This looks like trouble!

> Tip: if you do not see this behaviour, follow the previous tip regarding using gets rather than scanf, and recompile.

Run the program again, and try entering 10 “A”s, followed by “touch iwashere;ls” (as shown below).

```
[cliffe@cliffe-pc err]$  ./testerr
What is your name?
AAAAAAAAAAtouch iwashere;ls
hello AAAAAAAAAAtouch iwashere;ls, executing the command touch iwashere;ls
The files in the dir are:
iwashere  testerr  testerr.c
```

Subverting the behaviour of the program

Confirm you have tricked the program into creating a new file called “iwashere”:

```
ls -la
```

You may need to adjust the number of As, so that your command is correctly written to the “execute” variable.

At first glance the code looks innocent enough, but as a result of a potential programming mistake the user can gain control of another variable, and alter the way the program behaves.

So what is happening?

The two variables get positioned together on the stack, one next to the other. It is up to the compiler to decide which order they appear on the stack, but during my tests, they were positioned as follows:

name	0	1	2	3	4	5	6	7	8	9	**execute**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
												l	s	\0															

Note that in C, a string is simply an **array** of characters, so `name[0]` refers to the first letter, `name[1]` the second and so on. A C string is terminated with a '\0' character (null). So for example, as shown above, the value we set for the `execute` variable is followed by a null character.

If the user does as expected and just enters a short name, our variables contain values such as:

name	0	1	2	3	4	5	6	7	8	9	**execute**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	C	l	i	f	f	e	\0					l	s	\0															

However, our code did not instruct scanf how many characters to read from the user, so `scanf()` will obligingly read as many characters as the user enters, and writes them into the `name` buffer. Our example of entering 10 "A"s, followed by "touch iwashere;ls" results in the simplest form of a buffer overflow. A ***buffer overflow*** is when a buffer overflows into other memory. In this case one buffer overflows into an adjacent variable.

The result is:

name	0	1	2	3	4	5	6	7	8	9	**execute**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	A	A	A	A	A	A	A	A	A	A		t	o	u	c	h		i	w	a	s	h	e	r	e	;	l	s	\0

And now from the `printf()` call on line 7:

- printing `name` gives us "AAAAAAAAAAtouch iwashere;ls", since `execute[17]` is the first null character since the start of `name`
- printing `execute` gives us "touch iwashere;ls"

The ability to overwrite data that the programmer does not think the user can control can result in massive security consequences. In this example, the user can subvert the behaviour of the program to execute any command (within some length constraints).

Confirm this using GDB:

```
echo "Bob" > test-input1

echo "AAAAAAAAAAtouch iwashere;ls" > test-input2

echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" > test-input3

gdb -tui ./testerr

(gdb) layout split

(gdb) fs cmd

(gdb) break main
```

Debug the program, feeding in out test input:

```
(gdb) run < test-input2
```

Step into the code (to the next line of C code), setting the local variables:

```
(gdb) step
```

Print the value and then the memory address of the `execute` variable:

```
(gdb) print execute

(gdb) print &execute
```

Display the value stored at that memory address (the GDB x command), and output in the form of a string (/s):

```
(gdb) x/s 0xffffccae
```

(Where *0xffffccae* is from the output from the above “print &execute” command)

Use a scientific (numerical system mode) calculator to confirm that:

(hex) *0xffffccae* - *0xffffcca4* = 10 (base 10).

Hint: in another terminal window, run "`kcalc &`"

Therefore, you can conclude the two buffers are in adjacent locations of memory, and 10 bytes are reserved for the `name` buffer, as expected.

Continuing from above, step into the code (the next two lines of C code) that reads the input:

```
(gdb) step
(gdb) step
```

After the input has been processed, re-display the contents of the `execute` variable:

```
(gdb) print execute
(gdb) x/s 0xffffccae
```

(Where *0xffffccae* is from the output from the previous "print &execute")

```
(gdb) print name
(gdb) x/s 0xffffcca4
```

Stack smashing buffer overflows

In the previous example the data overwritten was another variable, changing the behaviour of the program. It is also possible to overwrite the return pointer that is stored on the stack as part of the function prologue, so that we overwrite the EIP register and therefore change the next code that is executed, effectively making the program jump to executing a different set of instructions. This type of attack is known as a ***stack smashing*** buffer overflow and is one of the most common and critical kinds of software vulnerabilities.

Continuing from above:

```
(gdb) layout regs
(gdb) disable breakpoints
```

```
(gdb) run < test-input3
```

When prompted whether to restart the program, select yes.

```
Program received signal SIGSEGV, Segmentation fault.

No function contains program counter for selected frame.
```

Our very long input has crashed the program. Often a stack smashing buffer overflow can result in crashing the program (causing a segmentation fault). However, sometime a far worse result can occur.

Read the current value of the EIP register:

```
(gdb) print $eip

$10 = (void (*)()) 0x41414141
```

As we can see, we have overwritten EIP with 0x41414141, which is a good sign that the program is vulnerable to further exploitation, since 41 is the hexadecimal ASCII representation of the letter 'A' (65 base10), which is what our input was made up of. So we did manage to write over the stack containing the EIP to restore!

As we will cover in a later lab, the ability to modify the EIP register often provides an attacker with the keys to the kingdom! As an attacker, we would aim to overwrite it with an address that points to code that we want to run.

Since 0x41414141 does not contain valid code, the program will crash with a segmentation fault.

```
(gdb) x/s 0xffffccae
```

Fixing the code

Programming languages such as C are not (strongly) type-safe; that is, it is possible to write any type of data such as integers or characters into any area of memory. Also these programming languages do not provide any automatic bounds checking to ensure that data is only written into the memory that has been allocated for specific variables. These issues have led to higher level programing languages, such as Java, including some type safety and bounds checking to avoid some of these security problems. However, there are many other design and programming mistakes that can lead to software vulnerabilities, and C and C++ are important, often highly efficient, and still widely used languages.

The vulnerability in this specific example code is caused by the incorrect use of `scanf()` or `gets()` (if you used the alternative code provided above).

The `gets()` function is ***never*** safe to use, it does not perform any bounds checking.

It is possible to safely use scanf(), by specifying the length of the buffer in the format string. For example "%31[^\n]" would read up to 31 characters.

Make a copy of the code:

```
cp testerr.c testerr_fixed_scanf.c
```

Edit the code:

```
vi testerr_fixed.c
```

Edit the new copy, to **fix the security problem by adding a length to the `scanf()` call**.

When you believe you have fixed the issue save your changes to the file (Esc, ":wq").

Compile your code:

```
gcc testerr_fixed_scanf.c -g -m32 -o testerr_fixed_scanf
```

Test your changes, with a valid, and an invalid input (try a long string).

If there are problems, edit and recompile the code until the program is secure.

> Hint: If your code is *still* overwriting the `execute` variable (and `execute` ends up empty), remember that you need to allow for an extra character for the terminating null (\0) character.

It is *recommended to use `fgets()`*, which accepts the length of text to read as a parameter.

From the man page ("`man 3 fgets`"):

> char *fgets(char *s, int size, FILE *stream);
>
> "fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by **s**. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer."

Make a copy of the code:

```
cp testerr.c testerr_fixed_fgets.c
```

Edit the code:

```
vi testerr_fixed_fgets.c
```

Edit the new copy, to **fix the security problem by adding a length to the `scanf()` call**.

When you believe you have fixed the issue save your changes to the file (Esc, ":wq").

Compile your code:

```
gcc testerr_fixed_fgets.c -g -m32 -o testerr_fixed_fgets
```

Test your changes, with a valid, and an invalid input (try a long string).

If there are problems, edit and recompile the code until the program is secure.

Conclusion

At this point you have:

- Learned how to understand C code, write simple C programs, and compile C into executable programs using GCC
- Learned some foundations of Assembly language, and interpreted machine instructions
- Used GDB to debug C code, stepping through code and viewing the assembly code and registers
- Caused some buffer overflows
- Written some C code to fix the issues in our example vulnerable software, using scanf and fgets securely

Well done!