ICT287 Computer Security

Tutorial Three: Software Security

In this lab you will learn all about the principles behind one of the most common and critical software security problems: buffer overflows.

You will use Kali Linux in this tutorial to write vulnerable C code to discover how this can be exploited. The tutorial assumes that you are familiar with the basics of C.

The weekly readings (linked on LMS) will be a great help, especially the old (but still relevant) article by Aleph One.

Hello World

First off, check that the C compiler (gcc) is working fine with a basic hello world. This particular one creates a couple of arrays that we will use later.

```
#include <stdio.h>
#include <string.h>


int main(void)

{

   char buf[10];

   char buf2[10];

   printf("Hello World\n");

   return 0;

}
```

Save this code with a useful name like hello_world.c and compile it with gcc. When you run the resulting file (with ./hello_world), the results should be self-explanatory.

Memory addresses

Using this base code, now print out the contents of the (uninitialized) **buf** and **buf2** using more **printf** statements in the main section. Also print out the base addresses of the two arrays (Hint: For a char array the address is the same as the address of the first element. For clarity we can use a pointer to the first byte like **printf("%p\n",&buf[0]);**)

Task 1: Compare the base addresses of buf and buf2. Use a calculator to work out the difference in bytes between them. Are they allocated in one continuous block? If so what would happen if we overflowed either buf or buf2?

A basic overflow

Now let's use some code straight from the Aleph One reading. Compilers have evolved a bit since that code was written, and as we'll see, we don't get the same results as he did.

```
#include <string.h>

void function(char *str) {

   char buffer[16];

   strcpy(buffer,str);

}

void main() {

   char large_string[256];

   int i;

   for( i = 0; i < 255; i++)

      large_string[i] = 'A';

   function(large_string);

}
```

Before running the code, think about what we expect to happen here, where is the overflow going to occur? What happens when you run the program?

Depending on your configuration of gcc, stack protection may be turned on and you are seeing stack canaries in action. These were discussed in the lecture.

You can turn off some of the protection that gcc offers to get the expected result.

```
$ gcc -fno-stack-protector example.c -o example
```

If stack protection is turned off by default on your system, turn it on, recompile the program and see what happens when you run it.

```
$ gcc -fstack-protector example.c -o example
```

In this example the stack smashing went way past the end of buffer. You can decrease the size of large_string to something around 25 and see if the stack protector is more helpful in this case.

A more dangerous buffer overflow

The following code is vulnerable to a buffer overflow attack. Where does the problem originate? If you are in doubt, refer to the reading on vulnerable C functions.

Think about the size of the buffer that can be overflowed, as you'll need this later.

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

void main()

{

   char *name;

   char *command;

   name=(char*)malloc(10);

   command=(char*)malloc(128);

   printf("Enter your name:");
```

```
    gets(name);

    printf("Hello %s\n",name);

    system(command);

}
```

Task 2: Compile and run this code to observe how the program works. Make sure you understand what each line of code is doing.

To make this do something useful, we want to overflow the vulnerable buffer and make the system do something new. If we refer to the above code, we can see that **name** and **command** arrays are likely to be sitting alongside one another. If we overflow **name**, the extra characters will end up in a different array, this one called **command**. Referring to the above code, we can see that the program will execute anything in this buffer.

Task 3: Use the same technique you used at the start of the lab, to discover the base addresses of name and command.

What is the difference (in bytes) between these two addresses? This is how much you need to input in order to overflow the **name** buffer.

Task 4: Craft the input to the program in such a way that you exploit a buffer overflow vulnerability to dump the system /etc/passwd or /etc/shadow file to screen.
Try running different system programs, like spawning a new shell and so forth.

Challenge task

In reality we may often work with compiled binaries and not be able to view the source code. A disassembler is a tool that will allow us to look inside the workings of a binary and observe the contents of variables and buffers as the program runs. The GNU debugger (gdb) is one such tool that is included in most Linux distributions.

For a challenge – refer to the Aleph one reading. The code example3.c from this reading shows code that will change the return address from a function to execute arbitrary code.

```
#include <stdio.h>

void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
   int x;

   x = 0;
   function(1,2,3);
   x = 1;
   printf("%d\n",x);
}
```

Unfortunately, this code is unlikely to work on modern architectures even if we use the `-fno-stack-protector` argument to `gcc`.

As a challenge – modify the values in "function" to change the flow of execution, and output "0" to the screen instead of "1".

You will need to change these lines:

```
   ret = buffer1 + 12;
   (*ret) += 8;
```

As for all challenges you will need to show your tutor how you solved the challenge to demonstrate that your solution can be generalised to new binaries. You should also provide the output of the modified tool to demonstrate that it worked.

Hint: to change the first line you need to use the debugger (gdb) to find out the address of buffer1 and the value of the stored EIP/RIP register during runtime. Make sure you compile the program with gcc -g.