CSE 565 Fall 2018
Homework 6
Due on November 13, 2018, at 9:30am

1. **Statistical databases.** Answer the questions below using the information stored in the table below. You can assume that a query can be formed using operators OR, AND, and NOT and can be of any of the following types: count(c), sum(c, a), max(c, a) and min(c, a), avg(c, a), where c is a query constraint and a is an attribute over which is computed the arithmetic function for all records that matched the query. The use of the attribute "Name" is not permitted in a query.

Name	Department	Position	Sex	Salary
Brown	CSE	Faculty	M	111
Eddy	Math	Staff	M	53
Dodge	Math	Faculty	F	102
Flint	EE	Student	F	28
Adams	CSE	Faculty	F	88
Toner	EE	Faculty	F	90
Evans	Math	Faculty	M	93
Smith	EE	Faculty	M	125
Hunt	CSE	Faculty	F	107
Jones	EE	Staff	M	66
Hayes	CSE	Faculty	M	95
Dieter	EE	Student	M	28

   (a) Assume no query size restriction and that a questioner knows that Toner is a female EE faculty. Show a sequence of a minimum number of queries that the questioner could use to reliably determine Toner's salary.

   (b) Suppose that there is a lower query size limit of 2 (i.e., queries that match a single record are rejected), but there is no upper limit. Show a sequence of a minimum number of queries that could be used to determine Toner's salary.

   (c) Now suppose that there is a lower query size limit of 2 (and no upper limit), but any two queries can overlap by at most 1 record (i.e., a query that overlaps a previous query by more than 1 record is rejected). The goal is also to determine Toner's salary through a sequence of queries. (If necessary, you can assume that the overlap size restriction applies to all queries of the same type only, i.e., querying the number of EE faculty and the sum of EE faculty salaries is allowed.)

2. **Return-to-libc buffer overflow attack.**

   The learning objective of this assignment is to gain deeper understanding of buffer overflow vulnerabilities, existing protection mechanisms, and techniques that might bypass them.

   As discussed in class, buffer overflow attacks pose a significant threat to computing systems, and modern operating systems provide defenses against buffer overflow attacks such as making the stack non-executable and address space randomization. The modern operating systems generally cannot be used to carry out simple buffer overflow attacks, and we will use a virtual machine which you can configure to disable some security features to work on building `return-to-libc` exploits. We'll use SEEDUbuntu12.04 image from http://www.cis.syr.edu/~wedu/seed/lab_env.html. The user manual lists account information and the web page also provides guidelines for running this VM using VirtualBox.

For this assignment, you will need to understand what information and in what order is placed on the stack. Also, you will need to read and understand the exploit given in the article "Bypassing non-executable-stack during exploitation using return-to-libc," which can be found at http://www.buffalo.edu/~mblanton/cse565/return-to-libc.pdf (referred to as [RTL] throughout this document). Information given in the article will be useful in constructing the exploit.

As a first step, consider the following program with a buffer overflow vulnerability.

```
/* vulnerable.c */
#include <stdio.h>

int main(int argc, char *argv[]) {
  char buffer[7];
  FILE *input = open("badfile", "r");

  fread(buffer, sizeof(char), XX, input);
  printf("Read from file: %s\n", buffer);

  return 0;
}
```

Your task will be to determine what value `XX` should take and create a file with the exploit (`badfile` above) to succeed in obtaining a shell from the above program.

To start, recall that return-to-libc attack overwrites local variables in a function, the function's return address, frame pointer, and one argument. We will be working with 32-bit platforms, so the latter three values will be 4 bytes long, while it is your task to determine the space occupied by the local variables in `vulnerable.c`.

Determining local variables space. For this part, you will need to perform the same experiments as described in [RTL] to determine the space occupied by local variables. By increasing the number of bytes you copy into the (fixed-size) buffer, you will cause the program to crash. By reading information in the core file using `gdb`, you will be able to tell when the function's return address is being overwritten (just like described in [RTL]).

Using the exact vulnerable program as given above is not necessary to correctly determine the space allocated to local variables. If it is going to make your task easier, you can create another program that reads input into the buffer from the standard input or from one of its arguments. However, make sure that your modified program allocates exactly the same amount of memory for local variables (i.e., one variable of 7 bytes long).

Note: by default, core files might not be created when your program crashes. To enable memory dumps into core files, use `ulimit` command (with bash).

Determining address of `libc` functions. The shared libraries may be loaded to a randomized address as well, so the techniques described in [RTL] may be necessary to find libc functions. To find out the address of any `libc` function, you can use any executable (e.g., an arbitrary program named `a.out`). The following commands will allow you to retrieve the addresses of functions `system()` and `exit()` (output from running the commands is omitted):

```
$ gdb a.out

(gdb) b main
(gdb) r
(gdb) p system
(gdb) p exit
```

The address will be printed in hexadecimal values, e.g., the address of the system function can be given as `0x9b4550`. For the exploit, you will need to overwrite the function's return address with the address of `system()` function, and old frame pointer with the address of `exit()` function (in this case, the program won't crash when you exit the shell).

Putting the shell string in the memory. The last step that remains is to place the string `"/bin/sh"` in the memory and determine its address. Normally, this can be achieved by using environment variables. For example, the environment variable `SHELL` might point to `/bin/bash` and can be used in this case. To minimize interference with programs that might use that environment variable, we will introduce a new shell variable called `MYSHELL`. You might instantiate it with the command:

```
$ export MYSHELL="/bin/sh"
```

(the syntax can vary depending on the shell you use). We will use the address of this variable as an argument to `system()` call. The location of the variable in the memory can be found easily using a program such as:

```
int main() {
  char *shell = getenv("MYSHELL");
  if (shell)
    printf("%x\n", shell);
  return  0;
}
```

If no address randomization is used by the operating system, this program will print the same address on different executions. However, when you run the vulnerable program `vulnerable`, the address of the environment variable might not be exactly the same as the one that you get by running the above program. Such address can even change when you rename your program or change the value of the variable, but will remain very close in different programs. To make it easier for you to experiment, you can disable address space randomization by executing command `sysctl -w kernel.randomize_va_space=0` as root.

Forming the exploit string. Your exploit can be stored in a file `"badfile"` either using Perl as suggested in [RTL] or by creating a small C program. The space allocated to local variables can be filled with an arbitrary pattern. For example, suppose your exploit consists of 24 bytes for local variables, `system` address `0x9b4550`, `exit` address `0x9a9b70`, and address of `"/bin/sh"` `0xffffdd11`. In the first case, we can use:

```
$ perl -e 'print "a" x 24 . "\x50\x45\x9b\x00\x70\x9b\x9a\x00\x11\xdd\xff\xff"' > badfile
```

In the second case, you can store addresses in a buffer in a C program using the following fragment:

```
char buf[36];
FILE *output;
...
*(int *)&buf[24] = 0x9b4550;
*(int *)&buf[28] = 0x9a9b70;
*(int *)&buf[32] = 0xffffdd11;
fwrite(buf, sizeof(buf), 1, output);
```

The attack is successful if after creating the exploit and executing `vulnerable` a shell prompt is obtained.

Because Ubuntu 12.04 uses stack protection (StackGuard), you will need to disable it in order for buffer overflows to work. This can be achieved by compiling your program with the `-fno-stack-protector flag` as in

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

There is also an option to explicitly making the stack to be executable or not (with options `-z execstack` and `-z noexecstack`), but for this assignment we want non-executable stack.

Dealing with address randomization. Because the VM uses address randomization, the address at which the string `"/bin/sh"` is stored, as well as the locations of the `system` and `exit` functions, will vary even on different executions of the same program. To be able to carry out the attack, you are allowed to modify the program `vulnerable.c` in any way that makes the attack succeed. Examples of modifications to the program include populating a large amount of memory with copies of the string `"/bin/sh"` so that knowing the exact location of the string is not necessary or obtaining its address and modifying the content of `badfile` while `vulnerable` is running. If you use additional variables, make sure that you compensate for them in your exploit. As before, the attack is successful if you obtain a shell prompt from `vulnerable`.

Update. Because of the specifics of the OS distribution that we are using, it is difficult to create a successful exploit when address randomization is enabled even if you extract the necessary addresses from `vulnerable` while it is running. Therefore, for the purpose of this assignment, you are required to have a working exploit when address randomization is disabled and an exploit that partially gets around address randomization when it is enabled. The latter needs to successfully determine the address of `"/bin/sh"` (taken from the environment variable) and place it in the exploit. Then if you can't reliably obtain the address of `system` for your exploit, the exploit will no longer work, but you need to demonstrate that all other components of your exploit are correct (and the exploit would work after placing the address of `system` in the exploit).

Submission of the assignment. Submissions of the code for this assignment is to be performed in electronic form using the `submit_cse565` command from a student system. Create a directory called `rtl`, place all of your programs in that directory, and submit the directory. Other information is to be turned in printed in class. This needs to include your report providing

- description of the steps performed to complete this assignment;
- description of the environment (e.g., any changes to the default configuration);
- values determined during the course of this assignment (i.e., amount of space allocated to local variables in the context of your program, addresses of `system()`, `exit()`, and `"/bin/sh"`);
- description of the approach for getting around address randomization;
- printout of the screen with the execution of the attack; it is not necessary to include a picture of the screen, copying information from the terminal or transcript into your report is sufficient.