CURTIN UNIVERSITY
Faculty of Science and Engineering
Department of Computing
Cyber Crime and Security Enhanced Programming

Assignment, Semester 2, 2017

Due date/time: Monday, 30 October, 2017 at 8AM.

1. Overview

Your task is to produce a small create/read/update/delete (CRUD) application for maintaining an employee records database. The 'database' exists as a simple CSV file that is loaded by the program and editable by the user before saving it back (it is obviously not going to be a multi-user system!). However, the point of the assignment is that the system must have exploitable vulnerabilities which you must document and demonstrate how to successfully exploit.

2. System Specifications

First though, we will discuss the specifications of the basic system. The system must be written in C/C++ as a console application that can compile and run under Linux. Note that the following specifications deliberately make little mention of input validation, error checking or error handling. This is because such checking would inhibit your options for exploitable vulnerabilities! However, where requirements are given, you must address them properly.

On startup the system must display the following main menu options:

1. Load Database
2. Display Employee Record
3. Add Employee
4. Edit Employee
5. Delete Employee
6. Save Database

2.1 LOAD DATABASE:

This option must request the path to a CSV file containing the current employee records. You may assume a maximum of 1,000 employees (so that you can use an array rather than a linked list – this is for your convenience, not for allowing a vulnerability). The CSV file has the following format per row:

- Employee ID – integer that counts from 1 and upwards
- Salutation – string up to 3 chars. **One of Mr, Ms, Mrs, Sir, Mdm**
- Employee first name – string up to 20 chars long. **Must not be blank**
- Employee last name – string up to 30 chars long. **Must not be blank**
- Position – string up to 15 chars long. **Must not be blank**
- Salary – Integer value that is **> 0** (use int: don't worry about decimal values)
- Deleted Flag – Integer. Not an editable field: 0 indicates employee is current, 1 indicates employee has been deleted (eg: due to termination of employment). In other words, deleted employees are not removed from the database, they are simply marked as 'deleted'.

Once the user has typed in the file path, the program must load the data and return the user to the main menu.

2.2 DISPLAY EMPLOYEE:

This option will first prompt the user for the employee ID to display. The program should then display the chosen employee record information (including the deleted flag) before returning the user to the main menu.

2.3 ADD EMPLOYEE:

This option must auto-generate a new employee ID, display the ID and then prompt the user to enter each of remaining the employee fields (salutation, first name, etc). The aforementioned valid input types (must not be blank names, > 0 salary, etc) must apply. The deleted flag must be initialized to 0.

The user is finally presented with the option to Save or Cancel the adding of the new employee. If Cancel is chosen, the employee record is *not* added to the list of employees. Otherwise with Save the new record is added to the end of the list. The user is then returned to the main menu.

2.4 Edit Employee:

This option will first prompt the user for the employee ID to edit. The program should then prompt the user to type in changes to the existing values of each field. The user is not allowed to edit the deleted flag.

The user is finally presented with the option to Save or Cancel the changes to employee record. If Cancel is chosen, the employee record is *not* changed (ie: the original values are left untouched). Otherwise with Save the changes are saved back to the relevant record in the list. The user is then returned to the main menu.

2.5 Delete Employee:

This option will first prompt the user for the employee ID to delete. The program should then display the employee name (first name followed by last name) and ask for a password to allow confirmation of the delete since only authorized users may delete employee records

(this password will probably need to be hardcoded into the program as we are not based on a true database – you may choose any password you like but it must be documented in the report you submit).

If the user confirms with the correct password, the record is marked as deleted (records are never truly deleted) and the user is returned to the main menu.

2.6 Save Database:

This option must request the path to a CSV file to save the database to. The program must write out the list of employees as they are in memory (ie: including all the edits and additions) in the same format as the loaded CSV file so that the new database file can be re-read by the program later.

2.7 Extra:

You may add reasonable additional specifications (eg: logging, etc) to the above list for the purposes of creating aspects that you can build in vulnerabilities.

3. Required Exploits

3.1 VULNERABILITIES REQUIRED

As mentioned, the point of the task is not merely to write the application (which at third year level should not be very hard to do). Instead, your assignment is to write the program in such a way as to include a set of exploitable vulnerabilities. These must be *no more and no less than* the following list:

- Exactly three (3) stack-based buffer overflows that can cause a crash or code injection vulnerability
- Exactly one (1) heap-based buffer overflow that can cause a crash or code injection vulnerability
- Exactly one (1) heap-based buffer overflow that allows corruption of another record (eg: negative salary, blank name), but *without* causing a crash
- Exactly three (3) memory leaks
- Exactly one (1) access-after-free of memory
- Exactly one (1) double-free

The point of this exercise is to get you thinking about what makes code vulnerable. I suggest that you begin to write the application as if it is to be bug-free, but then deliberately choose certain design and coding paths that will lead to vulnerabilities (eg: decide not to perform input validation on a string). You must *plan* your vulnerabilities – the idea is that when you think through where vulnerabilities can happen, you will begin to become a more vigilant programmer who looks out especially when coding those situations in the future.

A single line of source code cannot be used for more than one vulnerability. In other words, you cannot create multiple vulnerabilities out of a single flaw. Instead, you must ensure each flaw is wholly separate from the others. If it turns out to be very difficult to suppress secondary flaws without losing the intended flaw (eg: access-after-free might produce a double-free that can't be resolved without fixing the access-after-free) **then explain this in the report, and count it as a *single* flaw** (eg: access-after-free)**.**

Do not indicate where vulnerabilities are in the source code. You have a report for that.

Each vulnerability **must** be able to be triggered by the user via user input and/or the CSV file data (which is also input). However, **your program must be able to allow the user *some* path to perform every legitimate task of the application successfully without crashing**. This means your error checking/input validation is going to be patchy, because that's where vulnerabilities exist. Moreover, some vulnerabilities always cause a crash, so the only way to set up this dual-role of 'can do the task without crashing', but still allow triggering the vulnerability is if the program behaves differently

depending on the sequence of the user's decisions. For example, if a user edits a record, saves it then edits the same record again, the second time may crash.

- Memory leaks don't usually cause crashes
- Access-after-free may or may not cause a crash

The vulnerabilities must be a natural part of the program that come about due to laziness of the programmer or poor design. **You are not allowed to use artificial methods to trigger vulnerabilities**. For example, adding Boolean flags whose only purpose is to determine when to trigger a vulnerability will lose you marks. However, a count variable that is used to track the number of edits in a logging array that eventually goes out of bounds *is* acceptable *because it has a legitimate purpose, it is just badly implemented*.

3.2 Avoiding Too Many Vulnerabilities

Seeing as there is a strict limit on the number of vulnerabilities, you will be required to write much of the code so that it cannot be exploited. This is most difficult when handling user input. For example, note that by default scanf is unsafe and can allow a buffer overflow. However, by specifying the field width parameter one can ensure the buffer will not be overrun:

```
char sFirstName[21];
scanf("%20f", sFirstName); // Ensure first name cannot overflow
```

An alternative is fgets(), which allows explicitly defining the maximum length allowed. However, note that fgets() will include any '\n' in the input string.

Finally, note that both scanf and fgets will leave extra characters beyond the size (eg: 20) on the input buffer, and these will be read in by the next scanf/fgets without waiting for the user to type more. Although this is not a vulnerability, it can be annoying so you can 'eat' any extra input beyond the required maximum via a function such as:

```
void eatStdin()
{
   int c;
   do {
      c = fgetc(stdin);
   while ((c != EOF) && (c != '\n'));
}
```

4. Deliverables

You are required to submit a working version of the source code for your program and a report. All submissions should be digital (ie: uploaded to Blackboard).

4.1 PROGRAM

Submit your source code that complies with the specifications of this assignment to Blackboard. Ensure it can compile and run under the Linux environment in the university labs.

No part of your program source code is to indicate where vulnerabilities are. In other words, there should not be comments mentioning them, nor commented-out code that indicates what should have been there, or anything else special that implies there is an issue at that location. Comments are still required, but not to point out the vulnerabilities (the report is where vulnerabilities are discussed).

4.2 REPORT

You must also submit a professionally-organised report that documents your vulnerabilities in the system. It must have at least the following sections:

- A brief overview of the system.
- A section detailing each vulnerability, including:
    - The type of vulnerability.
    - The possible exploits that may be done using the vulnerability (eg: memory leak, crash, code injection, etc – nothing too fancy).
    - Step-by-step instructions on how to trigger the vulnerability (including sample user inputs) and the expected outcome of the provided example exploit. Screenshots or copies of the text output can be used to illustrate the example.
    - Where in the code the vulnerability is located.
    - A description of the vulnerability: what causes it to exist and be exploitable (this may be simple, or may involve explaining what flaws in the design the system has that lead to the vulnerability).
    - A description of how the code could be corrected to remove the vulnerability.
- Instructions on how to compile the program.
- Any known defects in the program (aside from the described vulnerabilities of course!).

4.3 Marks Breakdown

- Code: Good commenting — 4 marks
- Code: Clarity — 3 marks
- Code: Consistency — 3 marks
- Vulnerabilities
  - Stack buffer overflows (3) — 20 marks
  - Heap overflows (2) — 25 marks
  - Memory leaks (3) — 15 marks
  - Memory access-after-free (1) — 10 marks
  - Memory double-free (1) — 10 marks
- Report presentation/clarity — 10 marks
- *Additional Vulnerabilities* — *Up to -10 per extra vulnerability*
- *Invalid ('artificial') Vulnerabilities* — *Up to -10 per 'fake' vulnerability*

Total: **100 marks**

Appendix A: Code Evaluation

I will be looking through your code and marking you on how easy it is to read and understand your code as well as how much of the system you actually managed to implement. The first three (commenting, clarity, consistency) are described in more detail below:

Commenting: Ideally I should be able to skim through the comments in your code to understand what the code is supposed to be doing. Good commenting has several aspects:

- Briefly describes the purpose of blocks of code, rather than restating every line of code.
- Explains not just *what*, but also briefly summarises *why* and *how* for the more difficult parts
- Accurately reflects the code
- Provides a comment header for each function

Clarity: Names should be meaningful. Moreover, avoid unnecessarily complicated code since it's unmaintainable and makes it hard for people to understand your code. I find that for complicated bits, if I write the comment first I am forced to do a mini-design and think my way through solving the problem, and the resulting code is far less messy. So think before you code! And if you later figure out a better/simpler way of doing something, refactor the code to do so (it'll make later coding a lot easier and faster).

Consistency: Variables, classes and methods should be named according to Dept. of Computing coding standard and you should stick with that standard. Also, similar structures in different parts of the code (eg: loops through an array) should follow similar styles – since you are writing all the code this shouldn't be too hard to ensure.