CSC 435 – Computer Security

Lab 2 – Buffer Overflow

Solution

Part 1: General Questions

1. What is buffer overflow? On which parts of memory does it work? Explain how or when does it work on each?
   A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

2. What happen when you run an executable? Explain the steps of a process (main, function calls, variables …) and how they are saved in memory and pushed into stack/heap…
   It run in a process. Every process is assigned a stack. As the process executes the main function of the program, it is likely to encounter local variables and calls to functions. As it encounters each new local variable, it is pushed into the stack, and as it encounters a function call, it creates a new stack frame on the stack. This operational logic works recursively, in the sense that as local variables and nested function calls are encountered during the execution of a function, the local variables are pushed into the stack and the function calls encountered result in the creation of stack frames.

3. List the functions in C that are vulnerable to buffer overflow. What are their safe substitutes?
   Some unsafe functions with their safe substitutions.

UNSAFE	SAFER Alternative

`gets()`	`fgets()`
	`getchar()`
	`getline()`
	`gets_s()`

`strcpy()`	NOTE: `strncpy()` not a secure alternative -- it still prone to null-termination errors and other problems

`strcpy()`	`strncpy()`
	`strlcpy()`
	`strdup()`
	`strcpy_s()`

`strcat()`	`strncat()`
	`strlcat()`
	`strcat_s()`

`sprintf ()`	`snprintf()`
`vsprintf ()`	`vsnprintf()`

4. Name the different buffer overflow protection algorithms, and give a brief explanation of each.
   - Compile-Time Defenses:
     - Use a modern high-level language: Not vulnerable to buffer overflow attacks - Compiler enforces range checks and permissible operations on variables
     - Safe Coding Techniques: Programmers need to inspect the code and rewrite any unsafe coding. Example: OpenBSD project
     - Language Extensions/Safe Libraries: Example: Libsafe
     - Stack Protection: Add function entry and exit code to check stack for signs of corruption. Stackshield and Return Address Defender (RAD)
   - Run-Time Defenses:
     - Executable Address Space Protection: Use virtual memory support to make some regions of memory non-executable
     - Address Space Randomization: Manipulate location of key data structures - Randomize location of heap buffers - Random location of standard library functions
     - Guard Pages: Place guard pages between critical regions of memory - Further extension places guard pages Between stack frames and heap buffers

5. What does this string (“%x:%x:%s”) do? How can this be used for memory leaking?
   Prints first two words of stack memory. Treats next stack memory word as memory addr and prints everything until first ‘\0’. Could segment fault if goes to other program’s memory.

Part 2: Vulnerable C

Examine the thee C-codes shown below and answer the following questions for each:

1. Explain the program and how does it work?
2. Where are the variables stored?
3. What exactly is the type of attack the program is vulnerable to?
4. In which line of code is the vulnerable to buffer overflow?
5. Explain how to achieve buffer overflow?
6. If you want to use safe coding, how can you make this code immune to buffer overflow (just by adding some code)?

```
A.
   #include <stdio.h>
   int foo(){
      char ch; char buffer[5]; int i = 0;
      printf("Say something: ");
      while ((ch = getchar()) != ’\n’) buffer[i++] = ch;
      buffer[i] = ’\0’;
      printf("You said: %s\n", buffer);
      return 0;
   }
   int main() {
         foo();
      }
```

B.

```
      char *buf;
      int i, len;
      read(fd, &len, sizeof(len));
      buf = malloc(len);
      read(fd,buf,len);
```

C.

```
      #include <stdio.h>
      int main(int argc, char* argv[])
      {
         if (argc > 1)
               printf(argv[1]);
         return 0;
      }
```

A.

This program asks a user to enter a message. Whatever the user enters in a single line is accepted as the message and stored in the array buffer of chars. Worth mentioning that the call to getchar() reads one character at a time from this buffer.

The program will continue to function up to a point as you enter longer and longer messages in response to the prompt. But at some point, the string you enter will begin to overwrite the memory locations allocated to other variables on the stack and also, possibly the location where the return address of the calling function is stored. When this happens, the program will be aborted with a segmentation fault.

The basic idea that is used in several buffer overflow protection algorithms is a combination of rearrangement of the local variables on the stack and the insertion of a special variable, commonly called a canary, just below the stack locations reserved for the local variables.

B.

The overflow would occur in read (read(fd, &len, sizeof(len));)

Len might be negative; a solution would be to check if negative.

```
if (len < 0)
        {error ("negative length"); return; }
```

C.

This program is vulnerable to format string attacks, where calling the program with strings containing special characters can result in a buffer overflow attack.