Buffer Overflow Exploits and Protection Mechanisms

Glen Cathey, Emre Karagoz, Denis Ulybyshev
Department of Computer Science

Motivation

- Do a survey on protection methods against Buffer Overflow vulnerabilities
- Develop a lab for students so that they can:
  - Practice with exploiting a Buffer Overflow vulnerability in a simple file compression utility, written in C language
  - Fix the vulnerability and learn about different protection mechanisms

Buffer Overflow Exploit

- Buffer Overflow is a cyber-security vulnerability where assumed immutable data are corrupted or modified via the overflow of a buffer with malicious user input.
- 82 students were provided a generic C source code file (see Fig. 1), which contains a buffer overflow vulnerability.
- Each student was provided with a separate executable compiled from the vulnerable source code similar to a generic one, but with the unique secretFunction().
- The students' goal was to jump into the secretFunction [1] by overwriting the return address of FileCompress() – see Fig. 1. This was to be done by providing the malicious input file name to compress and overflowing the character buffer up the stack, therefore overwriting both %ebp and the return address of the FileCompress() function – see Fig. 5.
- The idea to jump to a secretFunction() is taken from [1]

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

void secretFunction()
{
    printf("You are in the secretFunction(). Secret message.\n");
}

void FileCompress()
{
        char buffer[20];
        char exists[28] = "test -f ";
        char zip[25] = "gzip ";
        char cp[45] = "cp ";
        char mv[45] = "mv ";
        int status;


        printf("Enter file name to compress:\n");
        gets(buffer);

        strcat(exists, buffer);
        status = system(exists);
        if (status == 256) {
                printf("%s", "File not found, exiting");
                return;
        }

        strcat(cp, buffer);
        strcat(cp, " ");
        strcat(cp, buffer);
        strcat(cp, "1");
        system(cp);

        strcat(zip, buffer);
        status = system(zip);
        if (status == 256) {
                printf("%s", "Gzip failed, exiting");
                return;
        }
        else {
                printf("%s has been zipped. \n", buffer);
        }

        strcat(mv, buffer);
        strcat(mv, "1");
        strcat(mv, " ");
        strcat(mv, buffer);
        system(mv);

        return;
}

int main()
{
    FileCompress();

    return 0;
}
```

Figure 1 : C source code for a simple file compression utility

Protection Mechanisms

- Non-Executable Stack – Malicious code injected into the stack will not be executed, only code in the data section of the address space is executed. This is achieved via flags at code compilation.
- Address Space Layout Randomization (ASLR) – The layout of the process' address space is randomized to prevent the same malicious payload from always functioning.
- Canaries – Special marker is placed between the buffer and return address and checked for correctness before executing the return.
- Address Sanitizer (ASan) – Open-source algorithm from Google™, which manages memory allocation and deallocation to prevent buffer overflows [2].
- Valgrind™ – Runs all the instructions in code virtually, in order to analyze memory usage and prevent data corruption [3].

```
08049152 <secretFunction>:
 8049152:       55                      push   %ebp
 8049153:       89 e5                   mov    %esp,%ebp
 8049155:       83 ec 08                sub    $0x8,%esp
 8049158:       83 ec 0c                sub    $0xc,%esp
 804915b:       68 08 a0 04 08          push   $0x804a008
 8049160:       ff 15 f4 bf 04 08       call   *0x804bff4
 8049166:       83 c4 10                add    $0x10,%esp
 8049169:       83 ec 0c                sub    $0xc,%esp
 804916c:       68 1c a0 04 08          push   $0x804a01c
 8049171:       ff 15 f4 bf 04 08       call   *0x804bff4
 8049177:       83 c4 10                add    $0x10,%esp
 804917a:       90                      nop
 804917b:       c9                      leave
 804917c:       c3                      ret
```

Figure 2 : Function to be jumped into via buffer overflow exploit

```
80492a8:       c6 85 7d ff ff ff 00    movb   $0x0,-0x83(%ebp)
80492af:       83 ec 0c                sub    $0xc,%esp
80492b2:       68 45 a0 04 08          push   $0x804a045
80492b7:       ff 15 f4 bf 04 08       call   *0x804bff4
80492bd:       83 c4 10                add    $0x10,%esp
80492c0:       83 ec 0c                sub    $0xc,%esp
80492c3:       8d 45 e0                lea    -0x20(%ebp),%eax
80492c6:       50                      push   %eax
80492c7:       ff 15 ec bf 04 08       call   *0x804bfec
```

Figure 3 : Allocation of the buffer in FileCompress()

References

[1] Kapil, Dhaval. "Buffer Overflow Exploit." [Online], https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/. Last Accessed: 20 Apr. 2020

[2] "AddressSanitizer" [Online], https://clang.llvm.org/docs/AddressSanitizer.html. Last Accessed: 20 Apr. 2020

[3] "About Valgrind" [Online], https://valgrind.org/info/about.html. Last Accessed: 20 Apr. 2020

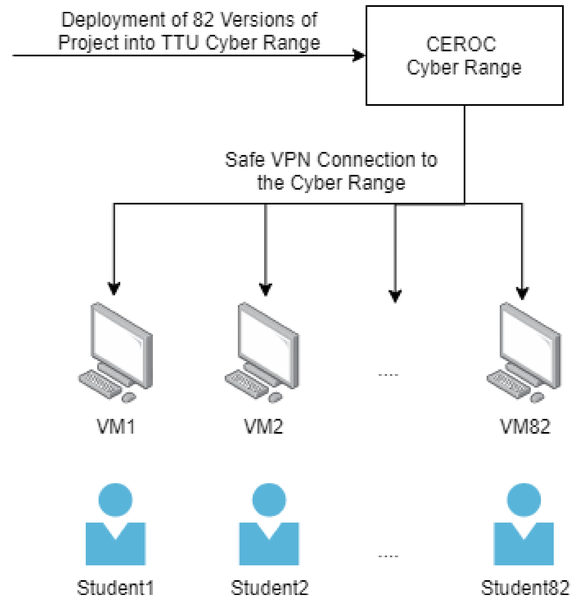

Figure 4 : CyberRange environment for the Buffer Overflow Lab

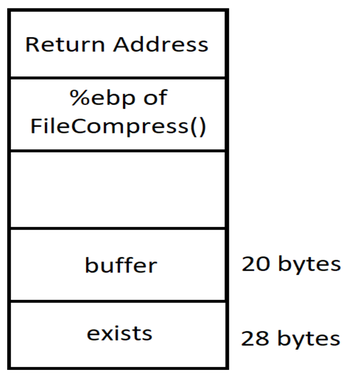

Figure 5 : Stack Memory Layout for FileCompress() Function

Acknowledgements

This Project is funded by QEP EDGE Curriculum grant from Tennessee Technological University