Lecture 4: Stack-Buffer Overflow

Summary

Stack-buffer overflows (and buffer overflows in general) are a fundamental class of security vulnerabilities in software. Generally caused by poor sanitization of user input, they may allow an attacker to run arbitrary code on a target machine. In this lecture, we focus on the mechanisms of a buffer-overflow attack; in the next lecture, we will cover some defenses against buffer overflow vulnerabilities.

Stack-Buffer Overflow

Reference: This material is not covered in P&P. A good reference is Stallings & Brown, *Computer Security: Principles and Practice,* Section 10.1.

Basic Overflow Refresher (see slides from Lecture 2)

Introduction to the in-out Package

Why is the in/out package “interesting” from a security perspective?

Owner and permissions

User Input

(Very) Basic fuzzing with Python

Find the vulnerability

Which function in the in/out package causes the buffer overflow?

Which C library function should the programmer *not* have used and why? What is an alternative that would have been better?

What does the stack look like (roughly) in the vulnerable function?

Exploitation Challenges

Knowledge of stack frame location

Where *exactly* is the functions return address?

Where *exactly* can we locate malicious code?

String processing

Malicious code must survive string processing

Exploit Components

There are three components of a basic stack-buffer overflow attack. All three components are part of a single string that will be passed to the vulnerable program as user input.

Shellcode (see sample)

What is the purpose of shellcode?

What are the two major constraints facing a shellcode writer?

Return Address

Why might an attacker include multiple copies of the return address?

NOP Sled

What is a NOP?

What is the purpose of the NOP sled (block of NOPs before the shellcode)?

Example: signin exploit walk-through

Exercises

1. For each of the following unsafe C library functions, find a safe alternative:

Unsafe Function	Safe Alternative
gets(char *str)	
sprintf(char *str, char *format, …)	

Unsafe Function	Safe Alternative
strcat(char *dest, char *src)	
strcpy(char *dest, char *src)	
vsprintf(char *str, char *fmt, va_list ap)	

2. Consider the shellcode used in class (it is available on the website — see Lecture 4). It is assumed that the call to `exec()` will be successful and not return. Suppose, however, that there is an error, and the call does return. The shellcode should exit gracefully by calling `exit(0)`. Extend the shellcode with the assembler instructions required to implement the call to `exit(0)`.

3. This is an example of a different type of buffer overflow vulnerability. How can one get the program to grant "root privileges" without knowing the correct password? (code example from www.thegeekstuff.com)

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

int main(void)
{
  int pass = 0;
  char buff[15];

  printf("Enter the password: ");
  gets(buff);

  if( strcmp(buff, "thegeekstuff") ) {
      printf ("\nWrong Password\n");
  } else {
    printf ("\nCorrect Password\n");
    pass = 1;
  }

  if(pass) {
    /* Now Give root or admin rights to user*/
    printf ("\nRoot privileges given to the user\n");
  }

  return 0;
}
```