Exploit Analysis

2010Mathematics Subject Classification.68M25.

1998CR Categories and Descriptors.D.4.6 [Sofware]: Operating Systems –Security and Protection

Abstract

Nowadays, speed is the most important thing that a program can have, since everyone is working exactly as much as they need, but also try to encapsulate as many activities as possible in a shorter time. Thus, more software applications became faster, being helped both by hardware components and by the programmer and technology. However, while everyone is using a computer, people are increasingly vulnerable to attacks of all kinds, most of them leading to personal information leaks or corruption of already existing files. This way, one must look forward to an application that is as safe as possible but also as fast as possible.

1. Overview

In this section, we provide an overview of an exploit, how to create it and how to protect an application for it.

1.1. What is an exploit?

An exploit is a piece of software, a chunk of data, or a sequence of command that takes advantage of a bug or vulnerability in an application or a system to cause unintended or unanticipated behavior to occur. The name comes from the English verb to exploit, meaning “to use something to one’s own advantage”. This means that the target of an attack suffers from a design flaw that allows people to create the means to access it and use it in his interest. (“What is an exploit? - Bitdefender”) Among the most well-known web-based security vulnerabilities are: SQL injection attacks, cross-site scripting, cross-site request forgery and broken authentication code or security misconfigurations. In general, exploits can be classified in two main categories: known and unknown (or zero-day vulnerabilities). The zero-day vulnerabilities are by far the most dangerous, as the occur when a software contains a critical security vulnerability of which the vendor is unaware. (“Roblox Free Exploit | QWIX Exploit 2021 - Download Cheat”) The vulnerability only becomes known when a hacker is detected exploiting the vulnerability, hence the term zero-day exploit. “Once such an exploit occurs, systems running the software are left vulnerable to an attack until the vendor releases a patch to correct the vulnerability and the patch is applied to the software.” (“What is an exploit? - Bitdefender”)

1.2. Buffer Overflows

The call stack is a data structure that contains the information about the active program. It is also known as the execution stack. This stack contains a stack frame for each function that has been called, and every stack frame contains their local variables, return parameter and the call arguments.

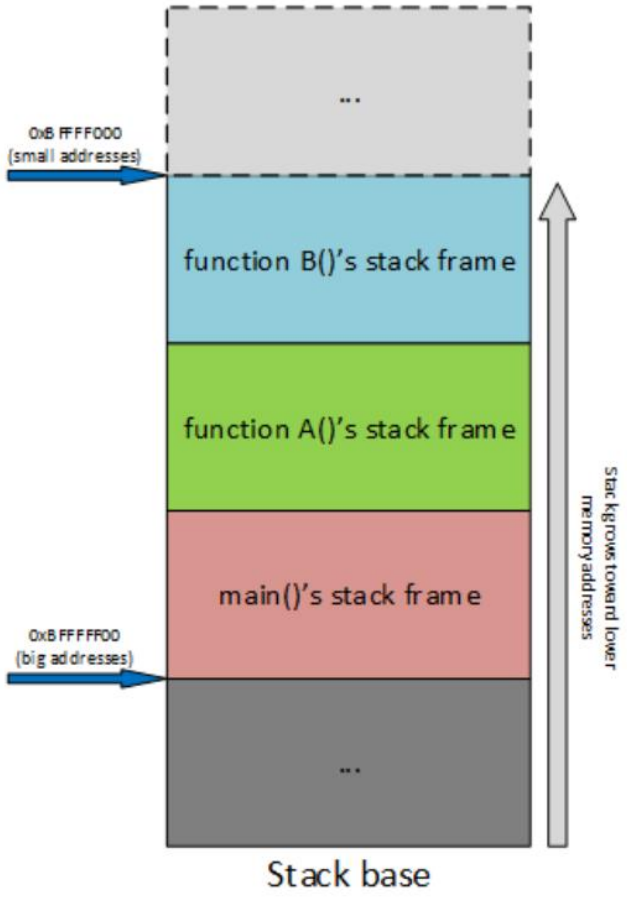

Image 1: Execution Stack of a Program

The buffer overflow attack is trying to get out of this stack frame to change the return address of a method so it can alter the future state of the program. Thus, by modifying the return address of the current function B (Image 1), one can go to another function, to another area in the memory to execute something else (say open a shell command line)

2. Testing

Testing is done on several types of vulnerabilities: Starting from buffer overflow vulnerabilities and going to more complex situations, such as SQL Injection. This approach wants to see how much time does protecting a software application consume.

2.1. Buffer Overflows

The testing for these types of exploits is going to be done with reading a high quantity of data, illustrating that for the same text that produces the altering of a function the results are going to be different. For example, taking this piece of code will present the effectiveness of protecting your application.

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  int num; // [rsp+Ch] [rbp-4h]

  init(argc, argv, envp);
  puts("Welcome, what is your name?");
  read(0, NAME, 0x20uLL);
  RANDBUF = "/dev/urandom";
  while ( 1 )
  {
    while ( 1 )
    {
      print_menu();
      num = get_num();
      if ( num != 1337 )
        break;
      game();
    }
    if ( num > 1337 )
    {
LABEL_10:
      puts("Invalid choice.");
    }
    else if ( num == 1 )
    {
      set_username();
    }
    else
    {
      if ( num != 2 )
        goto LABEL_10;
      print_username();
    }
  }
}
```

```
int print_menu()
{
  puts("1. Set Username");
  puts("2. Print Username");
  return printf(format);
}

int get_num()
{
  char buf[12]; // [rsp+Ch] [rbp-14h] BYREF
  unsigned __int64 v2; // [rsp+18h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  read(0, buf, 0xBuLL);
  return atoi(buf);
}

unsigned __int64 game()
{
  FILE *stream; // [rsp+8h] [rbp-18h]
  int ptr; // [rsp+14h] [rbp-Ch] BYREF
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  stream = fopen(RANDBUF, "rb");
  fread(&ptr, 1uLL, 4uLL, stream);
  printf("guess: ");
  if ( get_num() == ptr )
    system("/bin/sh");
  return v3 - __readfsqword(0x28u);
}

size_t set_username()
{
  FILE *v0; // rbx
  size_t v1; // rax

  puts("What would you like to change your username to?");
  v0 = stdin;
  v1 = strlen(NAME);
  return fread(NAME, 1uLL, v1, v0);
}
```

```
size_t set_username()
{
  FILE *v0; // rbx
  size_t v1; // rax

  puts("What would you like to change your username to?");
  v0 = stdin;
  v1 = strlen(NAME);
  return fread(NAME, 1uLL, v1, v0);
}

int print_username()
{
  return puts(NAME);
}
```

Image 2: Game code

We notice that if we call the function set username twice, providing a different length each time will result in only the length of the first name to be read. If we inspect the name variable, we can see that it reads 20 bytes. Reading 20 bytes will result in not having a null terminator, meaning that the function print username will print more than the name.

```
[DEBUG] Received 0x1c bytes:
    b'Welcome, what is your name?\n'
[DEBUG] Sent 0x20 bytes:
    b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB'
[DEBUG] Received 0x24 bytes:
    b'1. Set Username\n'
    b'2. Print Username\n'
    b'> '
[DEBUG] Sent 0x2 bytes:
    b'2\n'
[DEBUG] Received 0x4b bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 42  |AAAA|AAAA|AAAA|AAAB|
    00000020  24 40 17 ef  71 55 0a 31  2e 20 53 65  74 20 55 73  |$@··|qU·1|. Se|t Us|
    00000030  65 72 6e 61  6d 65 0a 32  2e 20 50 72  69 6e 74 20  |erna|me·2|. Pr|int |
    00000040  55 73 65 72  6e 61 6d 65  0a 3e 20                 |User|name|·> |
    0000004b
b' AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB$@\x17\xefqU\n'
```

Image 3: Debugging result

Leaking more than the name variable, we can take the “randbuf” address, the one where the number is called from. Overwriting that address will make us change the place where the random number is read from. To protect the application, we just need to add conditions that the new username is no longer than the previous, or to add a limit. This will be a T(1) operation, having no effect on the efficiency of the program.

2.2. SQL Injections

These types of exploits can be tested when parsing of an sql query is not done. Taking for example a simple query

SELECT * FROM Users Where Name = $name and Password = $password

Setting the $name variable to 'root or 1 == 1' will show us all the users. Thus destroying the whole security of an application. In order to protect the app from this kind of attacks, we need to use a Regular Expression. This is having an O(n) complexity, proven by using a Finite Automata. This will incresease the time complexity of the program considerably for large data sets or for long variable names. For login may not be a problem, but considering an example of searching in a subparagraph in a text, everytime we will search for something long, the user will have to wait for a longer time in order to get the result of the request.

3. Related Work

[1] Halfond, William G. J., and Alessandro Orso. "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks." Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 174–183.

This article presents an API created to automatically protect data from SQLIAs, and this implies a very high complexity as everything needs to be checked, having a worst time complexity of n^3, depending of the query type.

[2] Dalton, Michael, et al. "Real-World Buffer Overflow Protection for Userspace & Kernelspace." SS'08 Proceedings of the 17th Conference on Security Symposium, 2008, pp. 395–410.

This approach uses only the compiler defenses against buffer overflows, being even a better approach than adding conditions in code. Thus, this won't affect the efficiency of the program, letting the program doing its job as fast as possible and as safe as possible