Due: Dec 11, 2017. 11:59pm.(Late days may be used.)

1 Understanding Heap-Spraying Attacks

This project is part of an NSF-funded effort to increase CyberSecurity-related education in CS and ECE core courses at Virginia Tech. As part of this effort, we ask that you complete a post-survey after the project. Information will be forthcoming.

This is the initial offering of this project. It is intended to allow you to explore CyberSecurity aspects related to integrity that go beyond what was done in the Attack lab. We hope you enjoy the project!

2 Introduction

In this lab, you will be deepening your understanding of cybersecurity threats directed at the integrity of critical systems. This lab builds on the techniques you have learned in the attack lab, but introduces a number of new elements that are part of real-life attacks. In this lab, you will

1. Construct an attack against a vulnerable web browser by crafting malicious JavaScript code.
2. Construct and deliver an exploit that executes a target program on the victim's machine.
3. Use the heap spraying technique to increase the likelihood of a successful attack in the face of uncertainty about memory addresses and memory layout.

3 Adversary Model

We assume that the adversary (which will be your team in this lab) has the following abilities:

1. Load a malicious piece of JavaScript code into the victim's browser and execute it. For instance, the attacker might have gained control of a 3rd party ad service which content providers use to embed ads into the web pages their users are visiting.
2. We assume that the browser contains a separate vulnerability (such as a buffer overflow vulnerability like the one you exploited in the attack lab) that allows the attacker to transfer control. However, unlike in the attack lab, the nature of the vul-

nerability does not allow you to transfer control to a specific address, but rather to an approximate address within certain bounds.

3. We assume that the attacker has found a way to disable executable space protection for the range of memory addresses in which some JavaScript objects are allocated. Executable space protection is a very efficient way of protecting against code injection attacks, but there have been attacks in which attackers were able to overcome this line of defense.

4. In addition to executable space protection, we assume that the system has taken other countermeasures, specifically that the attacker is prevented from allocating large, contiguous objects on the heap. Real-life virtual machines (particularly their Just-In-Time compiler) exploit a variety of measures that make it more difficult for attackers to control the layout and size of objects in memory.

4 Heap Spraying

In the attack lab, you had considered how to construct an attack using gadgets - snippets of code, often starting in the middle of an instruction - that can be chained together to create longer sequences of code an attacker wishes to execute. In many cases, it is difficult to find out for an attacker where to find such gadgets. Heap spraying is a technique by which an attacker uses existing memory allocation facilities to place suitably crafted gadgets into the victims memory. This technique can be used in systems - such as web browsers - where an attacker can trigger the execution of code on the victims machine, such as JavaScript code downloaded by a browser.

5 Mode of Attack

The JavaScript code you will write should allocate a number of objects of your choosing. We recommend that you use the Uint8Array type that was introduced in the latest version of JavaScript (ES2017). It provides functionality that is similar to an `unsigned char []` array in C. You may store arbitrary byte values in the instances you allocate. Internally, those values will be stored consecutively in memory.

For this project, you will need only a very small amount of JavaScript. Fundamentally, JavaScript uses a similar syntax to C/Java, despite being a dynamically typed language at its core. Block structures, operators, array accesses, conditionals and loops work similar to C/Java. That said, if you have any doubts, feel free to ask questions regarding JavaScript syntax and semantics and we will answer them!

We provide access to the vulnerabilities assumed in the adversary model through a C++ function which you can call from your JavaScript code. This function is called as follows:

```
triggerOverflow(obj)
```

The function receives a reference to a JavaScript object. It will then use an (assumed) existing vulnerability to transfer control to a randomly chosen address in the vicinity of the real virtual address at which the content of the object referred to by obj is located.

Your goal is to trick the vulnerable browser into executing a binary located on the victims machine. In a real life attack, this may be an executable such as a server program or shell that then allows access to the victims machine. In this lab, we ask you to execute the binary "`./success`" which you may assume is located in the current directory where your code runs.

6 Constructing the Exploit Payload

Similar to the attacklab, you will need to figure out assembly code sequences to place inside the gadgets you allocate. You will want to include a "nop slide," a sequence of nop instructions followed by the actual exploit code. If control is transferred to any address within the nop slide, the exploit code at the bottom of the slide will eventually be triggered. Secondly, you will need to research how to write assembly code that performs a system call that triggers the execution of the success binary.

System calls are a way for a program to obtain access to services from the Operating System Kernel (which is the supervisory core control program that controls all activity on the machine). You will learn a lot more about system calls in CS3214. Examples of such services include input/output to terminals, files, or the network, as well as interaction with other processes running on the same machine. Starting a new process or executing a new program is also done via system calls. System calls look like C functions when they are invoked, but under the hood they trigger a transfer to the OS kernel that performs the desired task. You will need to construct the necessary assembly code to perform a system call to start executing the "`./success`" program.

As an example, consider the following assembly code sample.s, which does the equivalent of `write(1, "Hello World\n", 12); exit(0);`

```
# Look in /usr/include/asm/unistd_64.h for these constants
SYS_write = 1
SYS_exit = 60

 .globl start # entry symbol must be global

start:
 mov $SYS_write, %rax # system call number
 mov $1, %rdi # arg0
 lea hello(%rip), %rsi # arg1
 mov $hello_len, %rdx # arg2
 syscall # eq. of write(1, "Hello, World\n", sizeof(...));

 mov $SYS_exit, %rax # system call number
 mov $0, %rdi # arg0
 syscall # eq. of exit(0)
 # should not be reached if exit() is successful

hello:
 .asciz "Hello, World\n"
hello_len = . - hello
```

You can build and run this code as follows:

```
$ gcc -c sample.s -o sample.o
$ ld -e start sample.o -o sample
$ ./sample
Hello, World
```

The objdump command will show you the assembly code:

```
$ objdump -d ./sample

./sample: file format elf64-x86-64

Disassembly of section .text:

0000000000400078 <start>:
 400078:   48 c7 c0 01 00 00 00    mov $0x1,%rax
 40007f:   48 c7 c7 01 00 00 00    mov $0x1,%rdi
 400086:   48 8d 35 19 00 00 00    lea 0x19(%rip),%rsi # 4000a6 <hello>
 40008d:   48 c7 c2 0e 00 00 00    mov $0xe,%rdx
 400094:   0f 05    syscall
 400096:   48 c7 c0 3c 00 00 00    mov $0x3c,%rax
 40009d:   48 c7 c7 00 00 00 00    mov $0x0,%rdi
 4000a4:   0f 05    syscall
```

Note the use of RIP-relative addressing in the `lea` (load effective address) instruction. The use of this instruction allows for the code to be written in a way that makes it independent of the address in memory where it is stored. The only assumption is that the string “Hello, World” starts at offet 0x19 from 0x40008d. Appendix 10 I contains full documentation about the x86_64 Linux system call conventions.

Your exploit should provide code invoke the system call

```
    execve("./success", NULL, NULL);
```

7 Delivering the Exploit

Your JavaScript code will be included into the following webpage as script `heapspray.js`:

```
<!DOCTYPE html>
<html>
 <head>
 <title>A malicious page</title>
 <script type="text/javascript" src="heapspray.js"></script>
 </head>
 <body>(Elided)</body>
</html>
```

For testing, the autograder's browser will load this page repeatedly. We provide two autograder versions:

- `cs2506test` This version will run your code 5 times. You will get full information (including console output) for all runs to help you debug JavaScript syntax or runtime errors. You may use console.log to output information to the console.
- `cs2506final` This version will run your code 100 times and provide only tabulated results how often your exploit was successful in executing the ./success program. This will take longer, so you should try it only once you have consistently achieved success using the cs2506test target.

8 Using Uint8Array

Below is an example of how to write 4 bytes into contiguous memory using the `Uint8Array` class.

```
const payload = [
 0x31, 0xf6, 0x48, 0xbb,
];

var p = new Uint8Array(4);
for (var i = 0; i < payload.length; i++)
 p[i] = payload[i];

console.log(p);
```

Unlike in the attack lab (but more similar to an actual attacker in certain scenarios), you will not be able to run the exploit locally. However, you can write a C program that allows you to test your exploits. For the example given earlier, you could use:

```
/**
 * Show how to copy assembly code to the heap and invoke it.
 */
#include <sys/mman.h>
#include <sys/user.h>   // for PAGE_SIZE, PAGE_MASK
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>

const char payload[] = {
 /*400078:*/   0x48, 0xc7, 0xc0, 0x01, 0x00, 0x00, 0x00, // mov    $0x1,%rax
 /*40007f:*/   0x48, 0xc7, 0xc7, 0x01, 0x00, 0x00, 0x00, // mov    $0x1,%rdi
 /*400086:*/   0x48, 0x8d, 0x35, 0x19, 0x00, 0x00, 0x00, // lea    0x19(%rip),%rsi
 /*40008d:*/   0x48, 0xc7, 0xc2, 0x0e, 0x00, 0x00, 0x00, // mov    $0xe,%rdx
 /*400094:*/   0x0f, 0x05,                               // syscall
 /*400096:*/   0x48, 0xc7, 0xc0, 0x3c, 0x00, 0x00, 0x00, // mov    $0x3c,%rax
 /*40009d:*/   0x48, 0xc7, 0xc7, 0x00, 0x00, 0x00, 0x00, // mov    $0x0,%rdi
 /*4000a4:*/   0x0f, 0x05,                               // syscall
 /*4000a6:*/   0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x20,
               0x57, 0x6f, 0x72, 0x6c, 0x64, 0x0a        // "Hello, World\n"
};

typedef void (*function_t)(void);   // a pointer to a void f(void) function.
int
main()
{
    // allocate heap memory for the payload.
    char * heap_memory = malloc(sizeof(payload));
    memcpy(heap_memory, payload, sizeof(payload));

    // turn no-execute protection off for this page.
    void * page_base = (void *)((uintptr_t) heap_memory & PAGE_MASK);
    int rc = mprotect(page_base, PAGE_SIZE, PROT_READ | PROT_EXEC | PROT_WRITE);
    assert (rc == 0);

    ((function_t) payload) ();
}
```

9 Taking Memory Layout Into Account

In a real-world attack, the attacker may not be able to know the exact addresses of where in memory the objects they allocated are placed. We model this uncertainty using the assumed `triggerOverflow` function as follows: If you call `triggerOverflow(obj)`, then control will be transferred to a random address that is in the vicinity of obj.

Your goal, therefore, must be to maximize the chance that jumping to a random address in the vicinity of where `obj` is allocated will lead to the execution of your payload. To that end, you should experiment with allocating multiple payload objects as well as with nop slides in front of your payload. Keep in mind the restriction we have imposed for this lab that `Uint8Array` object cannot be larger than 1 000 bytes.

Furthermore, JavaScript - like Java - is a garbage collected language. This means objects to which no references are kept might be garbage-collected.

```
// will allocate 100 objects, but keep only the last one alive!
// objects 1..99 might be garbage collected
for (var i = 0; i < 100; i++) {
    var obj = new Uint8Array();
}
```

To avoid this, you must store a reference to the object inside some other object, such as an array.

10 How to submit

Submission will be via the autograder as described in this Google Document.

Good Luck!

The x86_64 system call conventions are as follows:

1. Registers used to pass the system call arguments. See also syscall(2).

arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7
x86_64	rdi	rsi	rdx	r10	r8	r9	-

2. A system-call is done via the syscall instruction. It may clobber %rcx and %r11, as well as %rax, but other registers are preserved.

3. The number of the syscall has to be passed in register %rax. System calls are limited to six arguments, no argument is passed directly on the stack. Upon return from the syscall, register %rax contains the result of the system call. A value in the range between -4095 and -1 indicates an error. User programs will set the errno variable to this value, multiplied by -1, and return -1 from the actual call.