Homework 2 Recitation

Çağrı Utku Akpak

November, 2021

Outline

1. Introduction
2. Specifications
3. Attack
   - Code Injection
   - Return Oriented Programming

Introduction

- 4 attacks against two targets.

Introduction

- 4 attacks against two targets.
- Buffer overflow is the security vulnerability.

Introduction

- 4 attacks against two targets.
- Buffer overflow is the security vulnerability.

Introduction

Objectives

- Learn to write safer programs.

Introduction
Objectives

- Learn to write safer programs.
- Gain understanding of stack and parameter passing mechanisms.

Introduction

Objectives

- Learn to write safer programs.
- Gain understanding of stack and parameter passing mechanisms.
- Gain understanding of how x64 instructions are encoded.

Introduction

Objectives

- Learn to write safer programs.
- Gain understanding of stack and parameter passing mechanisms.
- Gain understanding of how x64 instructions are encoded.
- Gain more experience with debugging tools OBJDUMP and GDB.

Introduction

Objectives

- Learn to write safer programs.
- Gain understanding of stack and parameter passing mechanisms.
- Gain understanding of how x64 instructions are encoded.
- Gain more experience with debugging tools OBJDUMP and GDB.

Getting Files

Server

`http://cakpak2.ceng.metu.edu.tr:15513/`

Getting Files

File Contents

- `README.txt`: Describe contents
- `ctarget`: Executable for *code-injection* attacks.
- `rtarget`: Executable for *return-oriented-programming* attacks.
- `cookie.txt`: An 8-digit unique identifier in hexadecimal.
- `farm.c`: Source code of the gadget farm for *return-oriented-programming* attacks.
- `hex2raw`: Program to generate attack strings.

Getting Files

Important Points

- You must work on similar machines. Only INEK machines can submit solutions to server.
- You are not allowed to circumvent validation (with touch1 exception). Any address you incorporate with `ret` instruction should be to one of the following destinations:
  - The addresses for functions `touch2`, or `touch3`.
  - The address of your injected code
  - The address of one of your gadgets from the gadget farm.
- You may only construct gadgets from file `rtarget` with addresses ranging between those for functions `start_farm` and `end_farm`.

Target Programs

- Two executables named CTARGET and RTARGET.
- Both target read from stardard input with `getbuf` function defined below:

GETBUF Function

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

Target Programs
GETBUF

- `Gets` works similarly to `gets`. Simply reads from `stdin` until it encounters `EOF`.
- Destination is an array `buf`, declared as having `BUFFER_SIZE` bytes.

Target Programs
GETBUF

- `Gets` works similarly to `gets`. Simply reads from `stdin` until it encounters `EOF`.
- Destination is an array `buf`, declared as having `BUFFER_SIZE` bytes.
  - They do not have a way to determine if the array is large enough to store the input.
  - This means that it is possible to overwrite the bounds allocated at destination.

Target Programs
GETBUF

- If the string is short it will return normally:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
[enter CTRL+D after newline to terminate]
No exploit.  Getbuf returned 0x1
Normal return
```

Target Programs
GETBUF

- If the string is short it will return normally:

```
unix> ./ctarget
Cookie: 0x1a7dd803
Type string: Keep it short!
[enter CTRL+D after newline to terminate]
No exploit.  Getbuf returned 0x1
Normal return
```

- Typically an error occurs if you type a long string:

```
unix> ./ctarget
Cookie: 0x1a7dd803
[enter CTRL+D after newline to terminate]
Type string: This is not a very interesting string, but it
has the property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

Target Programs
GETBUF

- Both targets works in the same way.
- Errors resulted from the program state corruption.
- You need to feed special strings to CTARGET and RTARGET to achieve certain results. They are called *exploit* strings.

Target Programs

Arguments

Command line arguments for CTARGET and RTARGET:

- `-h`: Print list of possible command line arguments
- `-q`: Don't send results to the grading server. Offline working option.
- `-i FILE`: Supply input from a file, rather than from standard input

You can also use gdb to make sure your program work as intended:

Example

```
> gdb ./ctarget
(gdb) r -q
(gdb) r -i ctarget.l1.raw
(gdb) r -q -i ctarget.l1.raw
```

Target Programs

Important Points

- Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program HEX2RAW will enable you to generate these *raw* strings.
- HEX2RAW expects two-digit hex values separated by one or more white spaces. So if you want to create a byte with a hex value of 0, you need to write it as `00`. To create the word `0xdeadbeef` you should pass "`ef be ad de`" to HEX2RAW (note the reversal required for little-endian byte ordering).

Target Programs

Important Points

- When you have correctly solved one of the levels, your target program will automatically send a notification to the grading server.

Example

```
unix> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0xXXXXXXXX, 0xXXXXXXXX)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

- You can view the scoreboard by pointing your Web browser at

  `http://cakpak2.ceng.metu.edu.tr:15513/scoreboard`

- Unlike the Bomb Lab, there is no penalty for making mistakes in this lab.

Target Programs

Point Distribution

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	`touch1`	10
2	CTARGET	2	CI	`touch2`	25
3	CTARGET	3	CI	`touch3`	25
4	RTARGET	2	ROP	`touch2`	35

CI: Code injection

ROP: Return-oriented programming

Figure: Summary of attack lab phases

Main Points I

- Your exploit strings will attack CTARGET in the part.
- Stack positions will be consistent from one run to the next and so that data on the stack can be treated as executable code.
- These features make the program vulnerable to attacks where the exploit strings contain the byte encodings of executable code.
- Function `getbuf` is called within CTARGET by a function `test` having the following C code:

```c
void test()
{
    int val;
    val = getbuf();
    printf("No exploit.  Getbuf returned 0x%x\n", val);
}
```

Main Points II

- When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within function `test` (at line 5 of this function). You need to change this behaviour.

Level 1 I

- For Phase 1, you will not inject new code. Your exploit string will redirect the program to execute an existing procedure. Its C representation is given below:

```
void touch1()
{
    srand(10);
    if ( rand()%25 == 20) { //This will always return 20.
        printf("Misfire: You called touch1() but you must not execute this par
        fail(1);

    }
        else { //This part will never be executed if touch1 is called directly
        vlevel = 1; /* Part of validation protocol */
        printf("Touch1!: You called touch1()\n");
        validate(1);
    }
    exit(0);
}
```

Level 1 II

- Your task is to get CTARGET to execute the code for `touch1` when `getbuf` executes its return statement, rather than returning to `test`. However you should make sure that else part of the if-statement is executed.

Level 1

Some Advice

- Exploit string for this level can be determined by examining a disassembled version of CTARGET. Use `objdump -d` to get this dissembled version.
- The idea is to position a byte representation of the address where `vlevel=1` is executed so that the `ret` instruction at the end of the code for `getbuf` will transfer control to it.
- Be careful about byte ordering.
- You can use GDB to step the program through the last few instructions of `getbuf`.
- The placement of `buf` within the stack frame for `getbuf` changes for each student and it is determined by `BUFFER_SIZE`, as well as the allocation strategy used by GCC. You need to examine dissembled version to determine its position.

Level 2

- Your task is to get CTARGET to execute the code for `touch2` rather than returning to `test`. Its C representation given below:

```
void touch2(unsigned val1, unsigned val2)
{
    vlevel = 2;       /* Part of validation protocol */
    if (val1 == cookie && val2 == (cookie >> 2) ) {
        printf("Touch2!: You called touch2(0x%.8x, 0x%.8x)\n", val1, val2);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x, 0x%.8x)\n", val1, val2);
        fail(2);
    }
    exit(0);
}
```

- First Argument: cookie
- Second Argument: cookie >> 2

Level 2

Some Advice

Some Advice:

- First argument to a function passed in register `%rdi` and the second argument is passed in register `%rsi`.
- Your injected code should set the registers to your cookie and cookie >> 2, then use a `ret` instruction to transfer control to the first instruction in `touch2`.
- Do not attempt to use `jmp` or `call` instructions in your exploit code.
- You need generate the byte-level representations of instruction sequences for injection.

Level 3 I

- Your task is to get CTARGET to execute the code for `touch3` rather than returning to `test`. Its C representation given below:

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    if ( sval == NULL )
        return 0;
    char cbuf[120];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 110;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}

/* Check if the cookie and its reverse are summed correctly in each element.
   The size of the summation array is 8. */
int checksum(unsigned cookie_param, unsigned* cookie_and_reverse_sum) {

    if ( cookie_and_reverse_sum == NULL )
```

Level 3 II

```
        return 0;
    char cbuf[70];
    /* Make position of check summation unpredictable */
    char *s = cbuf + random() % 60;
    sprintf(s, "%.8x", cookie_param);
    for ( int i=0 ; i<8 ; i++ ) { // Array size is 8
        if ( cookie_and_reverse_sum[i] != s[i] + s[7-i] )
            return 0;
    }
    return 1;
}

void touch3(char *cookie_string, unsigned* cookie_and_reverse_sum)
{
    vlevel = 3;        /* Part of validation protocol */
    if ( hexmatch(cookie, cookie_string) && checksum(cookie, cookie_and_revers
        printf("Touch3!: You called touch3 with correct parameters.\n");
        validate(3);
    } else {
        printf( "Misfire: You called touch3 with %s as your first "
                "argument and %p as your second argument.\n",
```

Level 3 III

```
                    cookie_string, (void *)cookie_and_reverse_sum);
        printf("Contents of your second argument follow below:\n");
        fflush(stdout);
        for ( int i=0; i<8; i++ ) {
            printf("%.8x ", cookie_and_reverse_sum[i]);
            fflush(stdout);
        }
        printf("\n");
        fail(3);
    }
    exit(0);
}
```

- First Argument: String representation of your cookie
- Second Argument: Element-wise summation of your cookie and its reverse using `ASCII` values

Level 3

Some Advice

Some Advice:

- The cookie string should consist of the eight hexadecimal digits (ordered from most to least significant) without a leading "`0x`."
- Do not forget to put a 0 at the end of your string.
- Second argument should have a 8 unsigned int characters consecutively. Unsigned integers are 4 bytes long.
- Your injected code should set register `%rdi` to the address of this cookie string and `%rsi` to the address of the summation array.
- The functions `hexmatch`, `checksum` and `strncmp` push data onto the stack, overwriting portions of memory that held the buffer used by `getbuf`. You need be careful where to place your arrays.

Generating Byte Codes I

- You will use GCC as an assembler and OBJDUMP as a disassembler to generate byte codes. For example, suppose you write a file `example.s` containing the following assembly code:

```
# Example of hand-generated assembly code
pushq   $0xabcdef          # Push value onto stack
addq    $17,%rax           # Add 17 to %rax
movl    %eax,%edx          # Copy lower 32 bits to %edx
```

- You can now assemble and disassemble this file:

```
unix> gcc -c example.s
unix> objdump -d example.o > example.d
```

- The generated file `example.d` contains the following:

Generating Byte Codes II

```
example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 68 ef cd ab 00          pushq  $0xabcdef
   5: 48 83 c0 11             add    $0x11,%rax
   9: 89 c2                   mov    %eax,%edx
```

- From this file, you can get the byte sequence for the code:

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

Generating Byte Codes III

- You can also add C-style comments to your string before feeding them to HEX2RAW.

```
68 ef cd ab 00     /* pushq  $0xabcdef  */
48 83 c0 11        /* add    $0x11,%rax */
89 c2              /* mov    %eax,%edx  */
```

Using HEX2RAW I

- HEX2RAW takes as input a *hex-formatted* string. In this format, each byte value is represented by two hex digits.
- Hex characters should be separated by whitespace.

Example

"012345" $\Rightarrow$ 30 31 32 33 34 35 00

- You can also put C-style comments into exploit string.

```
48 c7 c1 f0 11 40 00 /* mov    $0x40011f0,%rcx */
```

Using HEX2RAW I

Examples

There are several ways you can use HEX2RAW:

1. You can set up a series of pipes to pass the string through HEX2RAW.

```
unix> cat exploit.txt | ./hex2raw | ./ctarget
```

2. You can store the raw string in a file and use I/O redirection:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget < exploit-raw.txt
```

3. This approach can also be used when running from within GDB:

```
unix> gdb ctarget
(gdb) run < exploit-raw.txt
```

Using HEX2RAW II

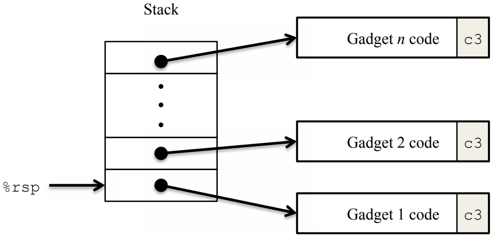
Examples

4. You can store the raw string in a file and provide the file name as a command-line argument:

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./ctarget -i exploit-raw.txt
```

Return Oriented Programming I

- RTARGET uses two techniques to prevent code-injection.
  - Randomizes stack so that its position cannot be determined.
  - Makes the stack non-executable.
- Solution is to use existing code other than injecting new code.
- The strategy of ROP is to identify byte sequences followed by a return instruction. These are called gadgets and they can be chained using return instructions.



Return Oriented Programming II

Figure: Setting up sequence of gadgets for execution. Byte value `0xc3` encodes the `ret` instruction.

Return Oriented Programming III

Examples

- One version of RTARGET contains following code:

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

- When we look at the dissambled machine code we encounter:

```
0000000000400f15 <setval_210>:
  400f15:       c7 07 d4 48 89 c7   #movl   $0xc78948d4,(%rdi)
  400f1b:       c3                  #retq
```

where `48 89 c7` encodes the instruction `movq %rax, %rdi` followed by a `ret` instruction.