2. Exercises

ASI36

2019

1 Using the tools : objdump, gdb, ida, ...

The following tools can be of help:

- **objdump**, to display various in information about object files;
- **nm**, to list symbols from object files;
- **strace**, to trace system calls and signals;
- **gdb**, to debug/follow the execution of your binaries;
- **ida**, for a graphical view of the CFG.

  See their respective man pages (e.g., `man objdump`) for details.

  A very thorough summary of `gdb`'s command can be found at

  `http://www.yolinux.com/TUTORIALS/GDB-Commands.html`

  Various cheat sheets are available online, such as this one:

  `https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf`

  A freeware version of IDA is available from the following page

  `https://www.hex-rays.com/products/ida/support/download_freeware.shtml`

1.1 `gdb` Extensions

It may be useful to install an extended configuration of `gdb`. Tools like `GEF`, `pwndbg`, `gdb-dashboard` can enhance your experience with `gdb`. Here is where you can download them

Tools	URL
`GEF`	`https://github.com/hugsy/gef`
`gdb-dashboard`	`https://github.com/cyrus-and/gdb-dashboard`
`pwndbg`	`https://github.com/pwndbg/pwndbg`

1.2 Disabling basic protections

These practical entry-level exercises need be done without usual practical protections (`W^X`, canaries, ASLR), which are the object of the next lecture.

That is why all binaries are compiled with the following options from `gcc`.

```
1 gcc -fno-stack-protector -z execstack
```

1.2.1 A note about ASLR

There is a good chance that your kernel uses ASLR. Check it out with the following command.

```
cat /proc/sys/kernel/randomize_va_space
```

If it returns something that is not 0, then it ASLR is enabled.
To disable it temporarily, spawn a shell like so:

```
setarch `uname -m` -R /bin/bash
```

All child processes launched from this shell will have ASLR disabled.

2 Optimization (`optims.c`)

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

int g(int a, int b) {
     if (a < 0 || b <= 0) {
          printf("Bad arguments\n");
          return -1;
     }

     if (a + b < 0) {
          printf("Overflow error\n");
          return -2;
     }
     printf("No detected errror\n");
     return a + b;
}

int main() {
     int r;
     // Add code here
     return 0;
}
```

1. Add the following calls:

   (a) `r = g(4, 8)`

   (b) `r = g(2, INT_MAX)`

   What are the possible results, depending on the compiler ?

   Try with:

   - `gcc`
   - `gcc -O2`
   - `gcc -O2 -fno-strict-overflow`

2. Propose a solution to make this function *secure*. You may use `https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow`

3 Basics (`basics.c`)

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

int main(int argc, char *argv[])
{
     char x = 0;
     char t[8] = {'0'};
     int i;

     if (argc != 3)
     {
          printf("Usage: p num1 num2\n");
          exit(1);
     }

     for (i = 0; i <= atoi(argv[2]); i++)
          t[i] = atoi(argv[1]);

     if (x != 0) printf("You win!\n");
     else printf("You lose\n");

     return 0;
}
```

1. Compile this program with `gcc` using the `fno-stack-protector` flag.

   This program may exhibit several behaviors. List them all and find test inputs for them. Explain what happens in every case, e.g., by drawing the execution stack.

   Look at the assembly code emitted for `main` in order to retrieve the offsets in the stack of the local variables.

2. Compile this program with `gcc` using the `fstack-protector` flag. What are the possible behaviors now ?

   Look at the assembly code and compare with the previous result question.

4 Take the heap (`h.c`)

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

char *p ;

int f (char a[])
{
     p = (char *) malloc (16 * sizeof(char));
     strcpy (p, "ls ");
     if (strlen(a) > 14)
     {
          printf("Filename too long !\n");
          free(p);
          return 0;
     }
     strcat(p, a);
     return 1 ;
}


int main(int argc, char *argv[])
{
     char *p3;
     if (!f(argv[1]))
     {
```

```
27          printf("Error:: Enter your log message (< 24 characters)\n");
28          p3 = (char *) malloc (24 * sizeof (char));
29          scanf("%s", p3);
30      }
31      system(p) ;
32      return 0 ;
33  }
```

This program takes a directory name as input and prints its content, like `ls`. If the argument is too long, an error message is printed and the user is asked to enter a string.

Test this program and explain why it is vulnerable.

Find how you can use this program to execute any shell command of your choice (`fortune`, `xeyes`, `figlet "foo"`, ...).

5 Format-string exploitation (`fmt.c`)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6       char text[1024];
7       static int test_val = -72, next_val = 0x11111111;
8
9       if(argc < 2) {
10          printf("Usage: %s <text to print>\n", argv[0]);
11          exit(0);
12      }
13      strcpy(text, argv[1]);
14
15      printf("The right way to print user-controlled input:\n");
16      printf("%s", text);
17
18      printf("\nThe wrong way to print user-controlled input:\n");
19      printf(text);
20
21      printf("\n[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val, test_val);
22      printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val, next_val);
23
24      if (next_val == 0xddccbbaa) printf ("You win!\n");
25      else printf ("You lose!\n");
26
27      exit(0);
28  }
```

1. Make the following code print "You win!" on the terminal.

   In order to do that, try to understand what the following does, assuming `next_val` is located at `0x5655702c`.

   ```
   ./fmt.bin $(printf "\x2c\x70\x55\x56")%x%x%8x%n
   ./fmt.bin $(printf "\x2c\x70\x55\x56")%x%x%100x%n
   ./fmt.bin $(printf "\x2c\x70\x55\x56JUNK\x2c\x70\x55\x56JUNK\x2c\x70\x55\x56")%x%x%100x%n%150x%n%228x%n
   ```

2. Find about `direct parameter access` and do the same exercise with it. Beware of wrap-arounds.

3. The last question uses short writes (as described below) to achieve the same results.

   A short is usually a two-byte word. Format parameters have a special way of dealing with them. See the length modifier section of the man page of `printf`.

As direct parameters can still be used, propose an enhanced solution using short writes with direct parameter access. Beware of possible wrap-arounds.

Read about short writes.

6 Exploiting reverse engineering (`bof`)

For the last exercise, you **only** have the binary. The program `bof` has a vulnerability that you need to exploit. You do not have access to the source code of `bof.c`.

In case of failure, it prints "I tawt I taw a putty tat!".

In case of success, for any of the 3 questions below, something else is printed. Guess what it prints, before answering any of said questions.

1. Make the program execute the `uwin` function.
2. Make the program execute the `superwin` function.
3. Make the program execute the `superwin` function **twice**.