

3. Exercises

ASI36

2021

1 Tweety Pie (twpie.c)

The goal of this exercise is to evaluate the stack protection mechanisms of GCC.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <time.h>
5
6 #define MAXBUF 16
7 #define SEC_SZ 6
8
9 char secret[MAXBUF] = {'0'};
10 int checked = 0;
11
12 typedef struct {
13     char s[MAXBUF];
14     int (*f)(char *);
15 } checker;
16
17 int basic_check(char *guess)
18 {
19     char logguess[16];
20     printf("Running basic check\n");
21     strcpy(logguess, guess);
22     checked = 1;
23     return strcmp(secret, logguess); // returns 0 on equality
24 }
25
26 int easy_check(char *guess)
27 {
28     char logguess[7];
29     printf("Running easy check\n");
30     strcpy(logguess, guess);
31     checked = 1;
32     return !((strlen(logguess) == strlen(secret))
33             && (logguess[0] = secret[0])
34             );
35 }
36
37 int apply_checker(checker cck)
38 {
39     return cck.f(cck.s);
40 }
41
42 int indirect_check(char *guess)
43 {
44     checker cck;
45
46     printf("Running indirect check\n");
47     cck.f = &basic_check;
48     strcpy(cck.s, guess);
49
50     return cck.f(cck.s);
51 }
52
```

```

53 int f(int n, char* guess)
54 {
55     int (*check)(char *);
56
57     check = &basic_check;
58     if (n == 42) check = &easy_check;
59     if (n == 0xffffffff) check = &indirect_check;
60
61     return check(guess);
62 }
63
64 void win()
65 {
66     printf("Success!\n");
67     return;
68 }
69
70 void fail()
71 {
72     printf("Failure!\n");
73     return;
74 }
75
76 int main(int argc, char *argv[])
77 {
78     char x = 0;
79     int res;
80
81     if (argc != 3)
82     {
83         printf("Usage: p num str\n");
84         exit(1);
85     }
86
87     srand(time(NULL));
88     for (int i = 0; i < SEC_SZ; i++)
89         secret[i] = (char) (0x41 + rand() % 26);
90
91     printf("My secret is %s. You would be lucky to guess it :-)\n", secret);
92
93     x = atoi(argv[1]);
94     res = f(x, argv[2]);
95
96     if (!res && checked) win();
97     else fail();
98     return 0;
99 }

```

1. Exploit this program without stack canaries.
 1. Turn-on basic stack canaries `-fstack-protector`.
 - What are the differences between the binary now and for 1 ? (check the `*_check` functions)
 - Does your exploit still work ? If not, propose another exploit that will still trigger the winning message.
 2. Turn on `-fstack-protector-all`
 - What are the differences between the binary now and for 1 ? (check the `*_check` functions)
 - Does your exploit still work ? If not, propose another exploit that will still trigger the winning message.
 3. Turn on non-executable stack (i.e. remove the `-z execstack` flag).
 - Does your exploit still work ? If not, propose another exploit that will still trigger the winning message.

2 ROP (roppable.c)

This exercise will show how to use basic return-oriented programming. The program below should be compiled with **ASLR** and **DEP** but **without stack protection**.

```
1 #include <string.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 char string[100];
6
7 void exec_string()
8 {
9     system(string);
10 }
11
12 void add_bin(int magic)
13 {
14     if (magic == 0xdeadbeef)
15     {
16         strcat(string, "/bin");
17     }
18 }
19
20 void add_sh(int magic1, int magic2)
21 {
22     if (magic2 == 0x8badf00d && (magic1 ^ magic2 == 0x754c2ea0))
23     {
24         strcat(string, "/sh");
25     }
26 }
27
28 void vulnerable_function(char* string)
29 {
30     char buffer[100];
31     strcpy(buffer, string);
32 }
33
34 int main(int argc, char** argv)
35 {
36     string[0] = 0;
37     if (argc < 2)
38     {
39         printf("Usage: roppable.bin arg\n");
40         exit(1);
41     }
42     vulnerable_function(argv[1]);
43     printf("Too bad ... try again\n");
44     return 0;
45 }
```

1. What happens if you execute:

```
1| ./roppable.bin $(python -c 'print("A"*100)')
```

Same question with

```
1| ./roppable.bin $(python -c 'print("A"*110)')
```

2. The goal is to make the program execute `"/bin/sh"` and thus give you a shell.
 - Draw the stack needed to execute the sequence `add_bin; add_sh; exec_string`
Some lightweight bitwise operations may be needed, e.g., to retrieve the value for `magic1`

- Find the needed gadgets in the binary code
- Make your exploit!