3. Exercises

ASI36

2020

1 Tweety Pie (`twpie.c`)

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

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

#define MAXBUF 16
#define SEC_SZ 6

char secret[MAXBUF] = {'0'};
int checked = 0;

typedef struct {
     char s[MAXBUF];
     int (*f)(char *);
} checker;

int basic_check(char *guess)
{
     char locguess[16];
     printf("Running basic check\n");
     strcpy(locguess, guess);
     checked = 1;
     return strcmp(secret, locguess); // returns 0 on equality
}

int easy_check(char *guess)
{
     char locguess[7];
     printf("Running easy check\n");
     strcpy(locguess, guess);
     checked = 1;
     return !((strlen(locguess) == strlen(secret))
               && (locguess[0] = secret[0])
          );
}

int apply_checker(checker cck)
{
     return cck.f(cck.s);
}

int indirect_check(char *guess)
{
     checker cck;

     printf("Running indirect check\n");
     cck.f = &basic_check;
     strcpy(cck.s, guess);

     return cck.f(cck.s);
}

```

```c
int f(int n, char* guess)
{
     int (*check)(char *);

     check = &basic_check;
     if (n == 42) check = &easy_check;
     if (n == 0xffffffff) check = &indirect_check;

     return check(guess);
}

void win()
{
     printf("Success!\n");
     return;
}

void fail()
{
     printf("Failure!\n");
     return;
}

int main(int argc, char *argv[])
{
     char x = 0;
     int res;

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

     srand(time(NULL));
     for (int i = 0; i < SEC_SZ; i++)
          secret[i] = (char) (0x41 + rand() % 26);

     printf("My secret is %s. You would be lucky to guess it :-)\n", secret);

     x = atoi(argv[1]);
     res = f(x, argv[2]);

     if (!res && checked) win();
     else fail() ;
     return 0;
}
```

1. Exploit this program without stack canaries.
2. 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.
3. Turn on `-fstack-protector-all`
   - What are the differences between the binary now and for 2 ? (check the `*_check` functions)
   - Does your exploit still work ? If not, propose another exploit that will still trigger the winning message.
4. 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 return-oriented programming. The program below is compiled with **ASLR** and **DEP** but **without stack protection**.

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

char string[100];

void exec_string()
{
    system(string);
}

void add_bin(int magic)
{
    if (magic == 0xdeadbeef)
    {
        strcat(string, "/bin");
    }
}

void add_sh(int magic1, int magic2)
{
    if (magic2 == 0x8badf00d && (magic1 ^ magic2 == 0x754c2ea0))
    {
        strcat(string, "/sh");
    }
}

void vulnerable_function(char* string)
{
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char** argv)
{
    string[0] = 0;
    if (argc < 2)
    {
        printf("Usage: roppable.bin arg\n");
        exit(1);
    }
    vulnerable_function(argv[1]);
    printf("Too bad ... try again\n");
    return 0;
}
```

1. What happens if you execute:

   ```
   ./roppable.bin $(python2 -c 'print "A"*100')
   ```

   Same question with

   ```
   ./roppable.bin $(python2 -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!