Lab1 Report

唐亚周 519021910804

1 Basic

1. 使用gcc编译源代码并执行观察输入输出。此处我额外 `#include <string.h>` 来消除 warnings。

```
➜  ~/S/3-Junior-2/I/SysSec-l/L/basic gcc -g flag.c -o flag
➜  ~/S/3-Junior-2/I/SysSec-l/L/basic ./flag
please input the flag:
flag
wrong!
```

2. 使用IDA Pro分析可执行文件并使用IDA Pro的hex ray插件(f5)进行反编译

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    int i; // [rsp+24h] [rbp-2Ch]
4    int j; // [rsp+28h] [rbp-28h]
5    char s1[8]; // [rsp+2Dh] [rbp-23h] BYREF
6    int v7; // [rsp+35h] [rbp-1Bh]
7    char s[15]; // [rsp+39h] [rbp-17h] BYREF
8    unsigned __int64 v9; // [rsp+48h] [rbp-8h]
9
10   v9 = __readfsqword(0x28u);
11   *(_QWORD *)s1 = 0x1D1C0B1B1E160D04LL;
12   v7 = 1381910;
13   memset(s, 0, sizeof(s));
14   puts("please input the flag:");
15   __isoc99_scanf("%11s", s);
16   for ( i = 0; i <= 10; ++i )
17     s1[i] ^= s[i];
18   for ( j = 0; j <= 10; ++j )
19     s1[j] = (s1[j] - 86) % 26 + 97;
20   if ( !strcmp(s1, "qwertyuiopa") )
21     puts("flag is correct!");
22   else
23     puts("wrong!");
24   return 0;
25 }
```

3. 分析程序中分析的算法并试解出正确flag。

   反汇编得到的代码中，源代码 `encrypted_text` 由 `s1` 和 `v7` 组合而成。注意，虽然这里 `s1` 的初始长度只有 8，但后面循环中访问了 0~10 位，说明溢出到 `v7` 所在的地址区域了。

```
1  *(_QWORD *)s1 = 0x1D1C0B1B1E160D04LL;
2  v7 = 1381910;
```

   考虑到我的电脑是 x86 架构，应为小端序，因此可以计算得到源代码 `encrypted_text` 的值为 `\x04\x0d\x16\x1e\x1b\x0b\x1c\x1d\x16\x16\x15`。

   GDB 调试查看内存证明了结论正确

```
0x7fffffffde38: 0xff    0xfb    0xeb    0xbf    0x00    0x04    0x0d    0x16
0x7fffffffde40: 0x1e    0x1b    0x0b    0x1c    0x1d    0x16    0x16    0x15
```

   然后根据后面的算法，写一个 Python 脚本来求出反汇编代码中的 `s`：

```python
result = [ord(i) for i in "qwertyuiopa"]
s1 = [ord(i) for i in "\x04\x0d\x16\x1e\x1b\x0b\x1c\x1d\x16\x16\x15"]
result1 = [(i - 97 + 86) for i in result]
s = [i ^ j for i, j in zip(result1, s1)]
s_str = "".join([chr(i) for i in s])
print(s_str)
```

得到 `s=baLyrevCrsC`。将其输入，发现是正确的。

```
~/S/3-Junior-2/I/SysSec-l/L/basic ./flag
please input the flag:
baLyrevCrsC
flag is correct!
```

4. 尝试使用gdb动态分析程序的输入如何被处理。

   在输入完毕后，查看相应位置的内存。可以看到 `0x7fffffffde3d` 到 `0x7fffffffde478` 上存储的是 `encrypted_text`，而 `0x7fffffffde49` 到 `0x7fffffffde54` 上存储的是输入的 `flag` 的值。

```
pwndbg> x/50bx 0x7fffffffde30
0x7fffffffde30: 0x02    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde38: 0xff    0xfb    0xeb    0xbf    0x00    0x04    0x0d    0x16
0x7fffffffde40: 0x1e    0x1b    0x0b    0x1c    0x1d    0x16    0x16    0x15
0x7fffffffde48: 0x00    0x62    0x61    0x4c    0x79    0x72    0x65    0x76
0x7fffffffde50: 0x43    0x72    0x73    0x43    0x00    0x00    0x00    0x00
0x7fffffffde58: 0x00    0x2f    0x33    0x3a    0x9e    0x54    0x2e    0xe3
0x7fffffffde60: 0x01    0x00
```

   执行异或操作的循环后，可以看到 `0x7fffffffde3d` 到 `0x7fffffffde478` 上存储的是异或操作的结果，而 `0x7fffffffde49` 到 `0x7fffffffde54` 上依然存储着 `flag` 的值。

```
pwndbg> x/50bx 0x7fffffffde30
0x7fffffffde30: 0x02    0x00    0x00    0x00    0x0b    0x00    0x00    0x00
0x7fffffffde38: 0xff    0xfb    0xeb    0xbf    0x00    0x66    0x6c    0x5a
0x7fffffffde40: 0x67    0x69    0x6e    0x6a    0x5e    0x64    0x65    0x56
0x7fffffffde48: 0x00    0x62    0x61    0x4c    0x79    0x72    0x65    0x76
0x7fffffffde50: 0x43    0x72    0x73    0x43    0x00    0x00    0x00    0x00
0x7fffffffde58: 0x00    0xfd    0x5e    0x4d    0xbe    0xbd    0x97    0x40
0x7fffffffde60: 0x01    0x00
```

   执行完接下来的操作循环后，可以看到 `0x7fffffffde3d` 到 `0x7fffffffde478` 上存储的是最终的结果，而 `0x7fffffffde49` 到 `0x7fffffffde54` 上依然存储着 `flag` 的值。

```
pwndbg> x/50bx 0x7fffffffde30
0x7fffffffde30: 0x02    0x00    0x00    0x00    0x0b    0x00    0x00    0x00
0x7fffffffde38: 0x0b    0x00    0x00    0x00    0x00    0x71    0x77    0x65
0x7fffffffde40: 0x72    0x74    0x79    0x75    0x69    0x6f    0x70    0x61
0x7fffffffde48: 0x00    0x62    0x61    0x4c    0x79    0x72    0x65    0x76
0x7fffffffde50: 0x43    0x72    0x73    0x43    0x00    0x00    0x00    0x00
0x7fffffffde58: 0x00    0xfd    0x5e    0x4d    0xbe    0xbd    0x97    0x40
0x7fffffffde60: 0x01    0x00
```

   最终将该结果与 `qwertyuiopa` 进行比较，并做出对应的输出。

2 Challenge

> 这里有一个PDF文件加密程序chall，把秘密文件加密成了pdf.enc，能否将加密文件还原呢？
>
> 逆向分析chall文件并将pdf.enc文件解密。
>
> （chall1和chall2是不同架构的相同程序，选一个分析即可。arm架构如需动态调试需要了解qemu模拟）

2.1 前期准备

首先查看两个文件的架构，发现 chall1 是 x86 而 chall2 是 arm，因此这里我选择 chall1 进行分析。

```
/mnt/D/SJ/3-Junior-2/I/SysSec-l/L/challenge file chall1 ✓ 20:52:46
chall1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, f
or GNU/Linux 2.6.32, BuildID[sha1]=4d30fee3c063cc67e90bd90d328b07ae18bf9ec3, not stripped
/mnt/D/SJ/3-Junior-2/I/SysSec-l/L/challenge file chall2 ✓ 20:52:48
chall2: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf
.so.3, BuildID[sha1]=c7365afd0a9fac85286ba86eccbbcf2fd56a84fe, for GNU/Linux 3.2.0, stripped
```

2.2 反编译[1][2][3]

1. `main` 函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  if ( argc == 2 )
    encrypt_file(argv[1]);
  return 0;
}
```

2. `encrypt_file` 函数

```
unsigned int __cdecl encrypt_file(char *a1)
{
  char ptr; // [esp+1Bh] [ebp-1Dh] BYREF
  unsigned int i; // [esp+1Ch] [ebp-1Ch]
  unsigned int v4; // [esp+20h] [ebp-18h]
  FILE *stream; // [esp+24h] [ebp-14h]
  FILE *s; // [esp+28h] [ebp-10h]
  unsigned int v7; // [esp+2Ch] [ebp-Ch]

  v7 = __readgsdword(0x14u);
  seed(a1);
  v4 = file_size(a1);
  stream = fopen(a1, "rb");
  s = fopen("pdf.enc", "wb");
  for ( i = 0; i < v4; ++i )
  {
    fread(&ptr, 1u, 1u, stream);
    ptr ^= rand();
    fwrite(&ptr, 1u, 1u, s);
  }
  fclose(stream);
  fclose(s);
  return __readgsdword(0x14u) ^ v7;
}
```

3. `seed` 函数

```
FILE *__cdecl seed(char *filename)
{
  FILE *result; // eax
  FILE *stream; // [esp+Ch] [ebp-Ch]

  result = fopen(filename, "rb");
  stream = result;
  if ( result )
  {
    fread(&RandSeed, 1u, 8u, result);
    result = (FILE *)fclose(stream);
  }
  return result;
}
```

4. `file_size` 函数

```
int __cdecl file_size(char *filename)
{
  int v2; // [esp+8h] [ebp-10h]
  FILE *stream; // [esp+Ch] [ebp-Ch]

  v2 = 0;
  stream = fopen(filename, "rb");
  if ( stream )
  {
    fseek(stream, 0, 2);
    v2 = ftell(stream);
    fclose(stream);
  }
  return v2;
}
```

5. `rand` 函数

```
unsigned __int64 rand()
{
  unsigned int v0; // ebx

  v0 = (unsigned __int64)(12142511 * RandSeed) >> 32;
  LODWORD(RandSeed) = (12142511 * RandSeed) ^ 0x3039;
  HIDWORD(RandSeed) = v0;
  return __PAIR64__(v0, RandSeed) >> 16;
}
```

2.3 分析

1. `seed` 函数将该文件的前 8 bytes 存储为 `RandSeed`。查找资料得知，PDF 文件的前 8 bytes 为 `%PDF=1.7` 形式（这里 `1.7` 是一个例子，表示版本号）。
2. `rand` 函数使用 `RandSeed` 进行一系列计算后返回一个伪随机数，同时更新 `RandSeed` 的值。
3. `file_size` 函数获得该文档的大小，单位为 byte。
4. `encrypt_file` 函数将原文件的每一个 byte 与 `rand` 函数返回值进行异或操作，再将其输出到加密后的文件。

2.4 思路

首先确定该 PDF 文件的版本号：对每个版本号对应的前 8 位字符串 `FirstEight` 进行测试，如果某个满足以下条件，则代表找到了正确的版本号。此处的 `FirstEightEnc` 代表加密后的前 8 位字符串，可以从 `pdf.enc` 文件中读取。

```
For i in [0..7]
    FirstEightEnc[i] == FirstEightEnc[i] ^ rand()
```

然后根据版本号来算出每位加密用的 `rand()` 函数返回值，再将加密后的每位都与其对应的 `rand()` 函数返回值进行异或，即得到原文件。

2.5 解密程序

在实现上遇到了一些困难，因此没能实现，非常抱歉。

3 感悟

我对 C 语言的类型转换实在是掌握得不熟练，需要多加练习。

---

1. https://reverseengineering.stackexchange.com/questions/8296/whats-the-function-of-lodword-and-hidword ↩

2. https://reverseengineering.stackexchange.com/questions/8317/what-does-pair-mean-in-hex-rays-decompiler-output ↩

3. https://wlaport.top/archives/231 ↩