Project0x01: Ransomware

Task 1: Files pattern

> The binary searches for files with a certain pattern and only encrypts those that match. Find out what the pattern is

The `ltrace` command shows 3 interesting library calls.

It seems that a part of the filename (i.e. `c19cf...`) in the working directory is compared with the string `encrypt_me_baby_one_more_time\377\377`:

```
strlen("c19cf21d23c2a054462451047b202711"...)                                    = 32
strlen("encrypt_me_baby_one_more_time\377\377")                                  = 31
strcmp("19cf21d23c2a054462451047b202711", "encrypt_me_baby_one_more_time\377\377") = -52
```

Decompiling the program using IDA, I found the code sequence from the previous `ltrace` command in the `sub_40152C` function.

The function returns `true` if:

- the length of the filename is greater or equal than the length of `encrypt_me_baby_one_more_time\377\377`
- the last 31 characters of the filename are `encrypt_me_baby_one_more_time\377\377`

```
v4 = strlen(file_name);
v3 = strlen(expected_file_name);
return v4 >= v3 && !strcmp(&file_name[v4 - v3], expected_file_name);
```

The previous function is called in `sub_401BA5`, having the filename and the expected filename as parameters. The `expected_file_name` is a global variable, having the value seen in the `ltrace` call.

If the returned value is `true`, then the encryption process will start:

```
result = will_enc(file_name, expected_file_name);
if ( result )
{
  start_enc_file(dir_name, file_name);
  return sleep(1u);
}
```

Debugging the program using gdb/peda, I set a breakpoint at address of the second `strlen` call, `0x401090` . Indeed, the argument is `encrypt_me_baby_one_more_time\377\377`:

```
   0x40154f:    mov    rdi,rax
=> 0x401552:    call   0x401090 <strlen@plt>
   0x401557:    mov    DWORD PTR [rbp-0x8],eax
   0x40155a:    mov    eax,DWORD PTR [rbp-0x4]
   0x40155d:    cmp    eax,DWORD PTR [rbp-0x8]
   0x401560:    jl     0x40158c
Guessed arguments:
arg[0]: 0x7fffffffde81 ("encrypt_me_baby_one_more_time\377\377")
```

Task 2: Encryption

> Describe how the encrypted files are internally structured (what bytes are written in the encrypted files and how the encryption is done)

I started by debugging the program using gdb/peda. Because I couldn't rename a file to contain the bytes `\377\377`, I had to bypass the filename verification process described in the previous section:

- set a breakpoint at address `0x401543`, where the string lengths are compared and set the `rdi` register to `0x7fffffffde81` (the value of `encrypt_me_baby_one_more_time\377\377`)

```
RSI: 0x7fffffffde81 ("encrypt_me_baby_one_more_time\377\377")
RDI: 0x405323 ("c19cf21d23c2a054462451047b202711")
RBP: 0x7fffffffde60 --> 0x7fffffffdee0 --> 0x7fffffffdf20 --> 0x7fffffffdf30 --> 0x401d90 (push   r15)
RSP: 0x7fffffffde40 --> 0x7fffffffde81 ("encrypt_me_baby_one_more_time\377\377")
RIP: 0x401543 (call   0x401090 <strlen@plt>)
R8 : 0x78 ('x')
R9 : 0x0
R10: 0x3
R11: 0x7ffff7a8bb30 (<__strcmp_sse2_unaligned>: mov    eax,edi)
R12: 0x4011a0 (xor    ebp,ebp)
R13: 0x7fffffffe010 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401538:    mov    QWORD PTR [rbp-0x20],rsi
   0x40153c:    mov    rax,QWORD PTR [rbp-0x18]
   0x401540:    mov    rdi,rax
=> 0x401543:    call   0x401090 <strlen@plt>
   0x401548:    mov    DWORD PTR [rbp-0x4],eax
   0x40154b:    mov    rax,QWORD PTR [rbp-0x20]
   0x40154f:    mov    rdi,rax
   0x401552:    call   0x401090 <strlen@plt>
Guessed arguments:
arg[0]: 0x405323 ("c19cf21d23c2a054462451047b202711")
```

- set a breakpoint at address `0x40157c`, where the `strcmp` function is called and again set the `rdi` register to `0x7fffffffde81`

```
RSI: 0x7fffffffde81 ("encrypt_me_baby_one_more_time\377\377")
RDI: 0x405323 ("c19cf21d23c2a054462451047b202711")
RBP: 0x7fffffffde60 --> 0x7fffffffdee0 --> 0x7fffffffdf20 --> 0x7fffffffdf30 --> 0x401d90 (push   r15)
RSP: 0x7fffffffde40 --> 0x7fffffffde81 ("encrypt_me_baby_one_more_time\377\377")
RIP: 0x40157c (call   0x4010c0 <strcmp@plt>)
R8 : 0x78 ('x')
R9 : 0x0
R10: 0x3
R11: 0x7ffff7a8bb30 (<__strcmp_sse2_unaligned>: mov    eax,edi)
R12: 0x4011a0 (xor    ebp,ebp)
R13: 0x7fffffffe010 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401572:	mov    rax,QWORD PTR [rbp-0x20]
   0x401576:	mov    rsi,rax
   0x401579:	mov    rdi,rdx
=> 0x40157c:	call   0x4010c0 <strcmp@plt>
   0x401581:	test   eax,eax
   0x401583:	jne    0x40158c
   0x401585:	mov    eax,0x1
   0x40158a:	jmp    0x401591
Guessed arguments:
arg[0]: 0x405323 ("c19cf21d23c2a054462451047b202711")
```

However, the program stops its execution. I found the function `sub_401593` being called multiple times during the encryption process. The function is responsible for blocking the debugging by calling `ptrace` with `PTRACE_TRACEME` parameter:

```
for ( i = 0; i <= 999; ++i )
{
  if ( dword_4040E4 != 1 && ptrace(PTRACE_TRACEME, 0LL, 1LL, 0LL) == -1 )
    _exit(1);
  dword_4040E4 = 1;
}
```

To bypass the anti-debugging mechanism, I set a breakpoint at address `0x4015cd` which corresponds to the `if` condition and set the `rax` register to 0.

```
   0x4015c8:	call   0x401110 <ptrace@plt>
=> 0x4015cd:	cmp    rax,0xffffffffffffffff
   0x4015d1:	jne    0x4015dd
   0x4015d3:	mov    edi,0x1
   0x4015d8:	call   0x401050 <_exit@plt>
```

Now, we can continue the debugging.

The program starts by calling the function `sub_401C7F` with the current directory as parameter:

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  enc_dir(".");
  return 0LL;
}
```

The function searches recursively in all subdirectories of the input directory. For each file found, it calls the function `sub_401BA5`, which checks if the filename matches the pattern and calls the function that will start the encryption, as seen in the previous task:

```
if ( dir->d_type == 4 )
{
  // is dir
  if ( strcmp(dir->d_name, ".") && strcmp(dir->d_name, "..") )
  {
    asprintf(&ptr, "%s/%s", dir_name, dir->d_name);
    enc_dir(ptr);
    free(ptr);
  }
}
else
{
  // is file
  prepare_enc_file(dir_name, dir->d_name);
}
```

The function `sub_4019D2` starts the encryption, by defining a new temporary filename. The final encrypted file will have a different name, as we will see in the next task.

```
asprintf(&old_path, "%s/%s", dir_name, file_name);
ptrace_block();
asprintf(&new_path, "%s/%s_temp", dir_name, file_name);
```

Also, the function which encrypts the files, `sub_40169A`, is called.

For the encryption, each byte starting from the end to the beginning of the input file, is read and summed with a random value. Then, this new value is written to the new file:

```
fseek(old_f, -1LL, 2);
*(_DWORD *)&v11[7] = 0;
while ( *(int *)&v11[7] < st_size )
{
  fread(&ptr, 1uLL, 1uLL, old_f);
  ptrace_block();
  fseek(old_f, -2LL, 1);
  ptrace_block();
  ptr += rand();
  ptrace_block();
  fwrite(&ptr, 1uLL, 1uLL, new_f);
  ptrace_block();
  ++*(_DWORD *)&v11[7];
}
```

Next, the string `fmi_re_course` is appended to the new file:

```
strcpy((char *)v11, "fmi_re_course");
for ( *(_DWORD *)&v11[7] = 0; ; ++*(_DWORD *)&v11[7] )
{
  v4 = *(int *)&v11[7];
  if ( v4 >= strlen((const char *)v11) )
    break;
  fwrite((char *)v11 + *(int *)&v11[7], 1uLL, 1uLL, new_f);
}
```

Finally, each byte from the old filename is summed again with a random value and appended to the new file:

```
for ( *(_DWORD *)&v11[7] = 0; ; ++*(_DWORD *)&v11[7] )
{
  v5 = *(int *)&v11[7];
  result = strlen(file_name);
  if ( v5 >= result )
    break;
  v9 = file_name[*(int *)&v11[7]];
  v9 += rand();
  fwrite(&v9, 1uLL, 1uLL, new_f);
}
```

Task 3: Renaming & Decryption

> Figure out how the file renaming process works and describe how decryption could theoretically be done

Task 3.1: Renaming

The renaming file is encrypted by the function `sub_4015F7`. Each byte from `0x401842` to `0x4019D2` is "encrypted", by xoring its value with `0x42`:

```
for ( i = (__int16 *)rename_file; ; i = (__int16 *)((char *)i + 1) )
{
  result = i;
  if ( i >= (__int16 *)start_enc_file )
    break;
  *(_BYTE *)i ^= 0x42u;
}
```

To decrypt the function, I used the following IDC script in IDA:

```
#include <idc.idc>

static decrypt(from, to){
  auto x;
  while (from < to) {
    x = Byte(from);
    x = (x^0x42);
    PatchByte(from, x);
```

```
      from = from + 1;
    }
  }
```

And called with the values:

```
decrypt(0x401842, 0x4019D2);
```

Before analyzing the renaming function, `seed` is a global variable which is set before encrypting any file:

```
seed = time(0LL) ^ 0xDEADBEEF;
srand(seed);
```

The renaming starts by computing `seed * seed * seed * seed`. The resulting value is stored in a `unsigned __int128` variable. Each of the these 16 bytes are represented as a hex string of length 2 and appended to the new filename. The resulting filename will have 32 (16 * 2) characters:

```
*(_QWORD *)new_file_name = '/';
memset(&new_file_name[8], 0, '\x03\xE0');
v11 = 1;
v2 = seed * (unsigned __int128)seed;
v3 = *((_QWORD *)&v2 + 1) * seed;
v4 = seed * (unsigned __int128)(unsigned __int64)v2;
v5 = (*((_QWORD *)&v4 + 1) + v3) * seed;
v6 = seed * (unsigned __int128)(unsigned __int64)v4;
*(_QWORD *)&rename_bytes = v6;
*((_QWORD *)&rename_bytes + 1) = *((_QWORD *)&v6 + 1) + v5;
while ( rename_bytes != 0 )
{
  sprintf(&new_file_name[v11], "%02x", (unsigned __int8)rename_bytes);
  rename_bytes >>= 8;
  v11 += 2;
}
new_file_name[v11] = 0;
ptr = 0LL;
ptrace_block();
asprintf(&ptr, "%s%s", dir_name, new_file_name);
return rename(file_name, ptr);
```

Task 3.2: Decryption

We saw that the encryption uses the `rand` function. However, `srand` is used before encrypting each file. The seed is obtained from the current time:

```
seed = time(0LL) ^ 0xDEADBEEF;
```

Using this information, if we know the time when the file was encrypted we can generate the same seed and apply the reversed process to decrypt the file.

Using the `stat` function we can obtain the last time when a file was modified. The data is stored in the field `st_mtim` of the returned *stat* structure. The seed for the `srand` function can now be computed by xoring the obtained time with `0xDEADBEEF`.

Next, all we have to do is to read the encrypted file byte by byte until the occurence of *fmi_re_course* sequence, subtract the value of `rand` function and write the new byte to the decrypted file. Notice that if we use the same seed, the same "random" numbers will be generated in order.

However, the bytes in the decrypted file are in reverse order. Read these bytes from the end to the beginning and write them in a new file. Now, the file should be decrypted.

Task4: Decryption script

> Create a program/script that decrypts any given encrypted file including the target file in the archive

The process described in the section Task 3.2 is implemented in decrypt.c.