Reversing task

Author: Karol Baryła (kb406092)

Let's start with analyzing the program in ghidra. First thing to notice is that program reads 0x128 bytes from provided file, and only continues if it succeeds. This, and the way those bytes are accessed later, tells that those 0x128 bytes are a struct, some kind of file header. Upon further inspection of the code, this structure can be deduced to look like this:

```
struct __attribute__((__packed__)) header_struct {
    uint32_t MAGIC_NUMBER;
    uint32_t file_length;
    char original_filename[256];
    char encrypted_password[32];
};
```

`header_struct.MAGIC_NUMBER` determines if file is encrypted using easy or hard algorithm. After that, there are few more things happening before decrypting file contents - let's leave those for later, and see how decryption is performed. Encrypted file is read, 4 bytes at a time, those 4 bytes (interpreted as 32bit int), are passed to decryption function (which one, easy or hard, is determined by magic number), function returns decrypted int, and it is written to decrypted file. Code looks like this (in Ghidra):

```
i = 0;
if (file_header.file_length != 0) {
  do {
    read_chars = fread(&local_17c,4,1,encrypted_file);
    if (read_chars != 1) {
      perror("fread");
      LAB_00100e88:
      retval = 1;
      goto LAB_00100d69;
    }
    if (file_header.MAGIC_NUMBER == 0xb542020) {
      local_17c = decrypt_easy(local_17c);
```

```
    }
    else {
      local_17c = func_0x001030b0(local_17c);
    }
    remaining = file_header.file_length - i;
    if (4 < remaining) {
      remaining = 4;
    }
    read_chars = fwrite(&local_17c,(long)remaining,1,decrypted_file);    if (read_chars != 1)
{
      perror("fwrite");
      goto LAB_00100e88;
    }
    i = i + 4;
  } while (i < file_header.file_length);
}
```

where `func_0x001030b0` is hard decryption function.

Now we know how decryption functions are used, let's go trough rest of this function. First, program iterates trough part of memory, and decrypts it using easy decryption function.

```
target_value = (uint *)0x103254;
next_ptr = (uint32_t *)0x1030b0;
do {
  i = *next_ptr;
  next_ptr = next_ptr + 1;
  i = decrypt_easy(i);
  next_ptr[-1] = i;
} while (next_ptr != target_value);
```

Decoded region contains 2 functions, hard decryption function, and one other - not important for now (let's call it strange_function). We can get decoded bytes easily - set breakpoint in gdb after decryption is complete, dump decrypted bytes to file. Getting Ghidra to analyze it is more tricky - byte view allows replacing single bytes, and should allow to replace byte range, but there were some errors no matter what I did. Fortunately, there is a way around it - let's automate inserting those bytes from keyboard - Ghidra doesn't have problem with that. After copying hexlified bytes, use this command: `sh -c 'sleep 10; xdotool type "$(xclip -o -selection clipboard)"'`. It waits 10 seconds, and

then simulates typing clipboard content on keyboard. Now, we have decrypted code in Ghidra, let's look briefly into decryption functions. Code of decrypt_easy:

```
uint decrypt_easy(uint param_1)
{
  uint uVar1;
  uint uVar2;

  uVar2 = param_1 ^ decrypt_buffer[0];
  uVar1 = decrypt_buffer[2] * 0x2137 + uVar2;
  decrypt_buffer[0] = decrypt_buffer[1];
  decrypt_buffer[1] = decrypt_buffer[2];
  decrypt_buffer[2] = decrypt_buffer[3];
  decrypt_buffer[3] = uVar1 >> 6 | uVar1 * 0x4000000;
  return uVar2;
}
```

It uses some global buffer (as some king of cyclic buffer), and does some operations on it - it's not really important what exaclty it does. The important part is: output depends on input and contents of this buffer, and this buffer is modified during call. decrypt_hard is similiar, just as bit more complicated.

```
uint decrypt_hard(uint param_1)
{
  uint uVar1;
  uint uVar2;

  uVar2 = param_1 ^ decrypt_buffer[0];
  uVar1 = decrypt_buffer[2] * 0x2137 + uVar2;  decrypt_buffer[0] = decrypt_buffer[2] * 0x7a69 +
decrypt_buffer[1];
  decrypt_buffer[1] = decrypt_buffer[2];  decrypt_buffer[2] = decrypt_buffer[2] * 0x1234567 +
decrypt_buffer[3];
  decrypt_buffer[3] = uVar1 >> 6 | uVar1 * 0x4000000;
  return uVar2;
}
```

There are 2 more things happening in main function. Buffer is modified, and strange_function is called:

```
decrypt_buffer[0] = getpid();
decrypt_buffer[1] = getppid();
strange_function(&file_header);
```

Password is processed. It is unhexlified, and written to different location. The tricky part is, the bytes are not saved just to some buffer, but also to other addresses, which can be hard to notice in default decompilation:

```
password_fragment = *__s;__isoc99_sscanf(password_fragment,"%x",unhexlified_pass_buffer + 4);
password_fragment = __s[1];__isoc99_sscanf(password_fragment,"%x",unhexlified_pass_buffer + 3);
password_fragment = __s[2];
__isoc99_sscanf(password_fragment,"%x",0x1030d7);password_fragment = __s[3];
__isoc99_sscanf(password_fragment,"%x",0x1030bd);password_fragment = __s[4];
__isoc99_sscanf(password_fragment,"%x",unhexlified_pass_buffer + 2);password_fragment = __s[5];
__isoc99_sscanf(password_fragment,"%x",0x1030dc);password_fragment = __s[6];
__isoc99_sscanf(password_fragment,"%x",unhexlified_pass_buffer + 1);password_fragment = __s[7];
__isoc99_sscanf(password_fragment,"%x",unhexlified_pass_buffer);
```

These addresses are located in decrypt_hard, in fact they correspond with constants found in decrypt_hard's code - so those constants will be replaced with parts of password. After that, unhexlified password is "decrypted" (in this case we should probably say encrypted?) using decrypt_easy, and compared with a field from header (if they are equal, password is correct).

```
global_pwd = password_global;password_global[0] = decrypt_easy(unhexlified_pass_buffer[4]);
password_global[1] = decrypt_easy(unhexlified_pass_buffer[3]);password_global[2] =
decrypt_easy(0x2137);
password_global[3] = decrypt_easy(0x7a69);password_global[4] =
decrypt_easy(unhexlified_pass_buffer[2]);password_global[5] = decrypt_easy(0x1234567);
password_global[6] = decrypt_easy(unhexlified_pass_buffer[1]);password_global[7] =
decrypt_easy(unhexlified_pass_buffer[0]);retval =
memcmp(file_header.encrypted_password,global_pwd,0x20);
```

The way Ghidra decompiles this code is very confusing, and in my opinion incorrect - it shows that numeric literals are passed to decrypt_easy, but in fact it uses values under addresses (and the addresses are in writeable part of memory, so no idea why it shows literals):

```
        00100c14 8b 3d a3        MOV        phVar1,dword ptr [LAB_001030bc+1]                24
```

```
00 00        00100c1a 89 05 68        MOV        dword ptr
[password_global[2]],read_chars        = null
                 26 00 00        00100c20 e8 db 03        CALL
decrypt_easy                                      uint decrypt_easy(uint param_1)
00 00
```

The point is, constants in decrypt_hard are modified, so they are more like variables, and we need to account for this while recreating program. That's pretty much all that we need to know, to create source of program (we didn't look at strange_function yet, but it's not important how it works). Let's copy all code to .c file, fix compilation errors, and remember that we need to perform decryption of the code, because it modifies the buffer. The result is code that works as original (+ prints some debug info). Link to code

Now we can work with this source, binary is no longer relevant. Let's clean up and simplify this code. Apart from typical cleanup, there is one thing we can do to eliminate some code. Decryption of encrypted functions is unnecessary - we don't use the result, but still need the procedure, because it modifies the encryption buffer. So let's print the buffer after decryption, and use it as starting buffer, then we can skip this whole thing. Link to cleaned up code.

Now let's see what is strange_function and how to simplify it. Internally it uses a function I called `push_data_to_buffer`, defined as follows:

```
void push_data_to_buffer(uint32_t param_1){
  uint32_t retval = param_1 ^ decrypt_queue[0];
  decrypt_queue[0] = decrypt_queue[1];
  decrypt_queue[1] = decrypt_queue[2];
  decrypt_queue[2] = decrypt_queue[3];
  decrypt_queue[3] = retval >> 6 | retval << 0x1a;
  return;
}
```

It treats decrypt_queue (this name is not really good) as cyclic buffer. Operations here are pretty simple. Binary first sets 2 elements of this buffer, then call `strange_function`:

```
decrypt_buffer[0] = getpid();
decrypt_buffer[1] = getppid();
strange_function(&file_header);
```

`strange_function` first uses `push_data_to_buffer` 64 times - on each 4 bytes of `original_filename` field of header struct. Then it opens `/proc/file/stat`, and searches for fields ending with `Pid` - which will be `Pid`, `PPid`, `TracerPid`. It seems to be some form of anti-debugging - if `TracerPid` is different from 0, decryption will fail. The interesting part is: pid() and ppid() will give different results every run. Why anything works here? It is caused by how `push_data_to_buffer` works. If `buffer[0] = x`, then it is called 64 times, let's say with `paramn_1 = 0` for simplicity, then again `buffer[0] = x` (buffer will cycle 64 / 4 = 16 times, so x will be periodically bit-shifted by 6 * 16 = 96 bits, which is a multiply of 32). So the `Pid` and `PPid` fields will zero out with values from `/proc/self/status`. Which means we can use any values insted of `pid()` and `ppid()` as long as we call `push_data_to_buffer` again with same values after 64 calls. That means we can simplify it a bit. Let's delete `pid()` and `ppid()` calls, set first 2 elements of buffer to 0's and change `strange_function` to this:

```
void strange_function(struct header_struct* header) {        char* orig_filename = header-
>original_filename;
    char* enc_password = header->encrypted_password;
    do {
        push_data_to_buffer(*(uint32_t *)orig_filename);
        orig_filename += 4;
    } while (orig_filename != enc_password);

    for(int i = 0; i < 3; i++) {
        push_data_to_buffer(0);
    }
}
```

Yay, everything still works. Only thing that's left is to add decrypting without password. There is nothing problematic about it - it is a simple xor. Earlier we saw, that return value of `decrypt_easy` is argument xored with first element of buffer. And we know that "decrypted" password has to be equal to specific field in file header. So, goal is:
`decrypt_easy(unhexlified_pass_buffer[i]) == (int*)file_header.encrypted_password[i]` . So, for index 0 we have: `decrypt_queue[0] ^ unhexlified_pass_buffer[i] == (int*)file_header.encrypted_password[0]`. Well, we know `decrypt_queue` and `file_header.encrypted_password`, so we can easily decrypt password:

```
for(int i = 0; i < 8; i++) {  // Password cracking - originally there was a sscanf that unhexlified entered password  unhexlified_pass_buffer[i] = ((int*)file_header.encrypted_password)[i] ^ decrypt_queue[0];  password_processed[i] = decrypt_easy(unhexlified_pass_buffer[i]);
 }
```

Link to final code

Revision #6
Created Tue, Nov 24, 2020 11:49 AM by Lorak_
Updated Thu, Jan 7, 2021 11:09 PM by Lorak_