Flare-On Challenge 8 Solution

By Moritz Raabe

Challenge 7: spel

Challenge Prompt

Pro-tip: start disassembling this one then take a nice long break, you've earned it kid.

Solution

This challenge was inspired by multiple malware samples we've analyzed over the last year. It all starts with a Windows 64-bit executable. To get the flag we need to understand and overcome various executable stages, anti-analysis techniques, and obfuscations.

This writeup focuses on the key components and does not describe every functionality in detail. The main analysis tools we use are FLARE VM, IDA Pro, Sysinternal Suite tools, FakeNet-NG, capa, FLOSS, and CyberChef.

Basic Analysis

With a file size of more than 4 MB this is a larger binary with many sections, imports, resources, and strings. In the file properties the program self-identifies as `Spell FON Application` (see Figure 1). Browsing through the strings the program appears to use the Microsoft Foundation Class (MFC) library which can be used to create applications with complex user interfaces. Malicious code can hide easily in statically linked MFC binaries which contain a lot of MFC library functions and binary resources.

Property	Value
CompanyName	FLARE <3
FileDescription	Spell FON Application
FileVersion	1, 0, 0, 1
InternalName	Spell
LegalCopyright	Copyright (C) 2021
OriginalFilename	Spell.EXE
ProductName	Spell Application

Figure 1: Challenge file properties

To get a first idea of the file, we run capa on the binary. Since version 2.0 capa can identify library code and is able to skip about 6,800 library functions (82% of all identified functions) in this binary. Library code identification focuses the results on program-unique functionality and significantly speeds up the analysis. The capa results are shown in Figure 2.

```
+------------------------------------------------------+---------------------------------------------+
| CAPABILITY                                           | NAMESPACE
|
|------------------------------------------------------+---------------------------------------------|
| contain obfuscated stackstrings                      | anti-analysis/obfuscation/string/stackstring
|
| log keystrokes via polling                           | collection/keylog
|
| contain a resource (.rsrc) section                   | executable/pe/section/rsrc
|
| contain a thread local storage (.tls) section        | executable/pe/section/tls
|
| extract resource via kernel32 functions (8 matches)  | executable/resource
|
| set environment variable                             | host-interaction/environment-variable
|
| delete file                                          | host-interaction/file-system/delete
|
| get file attributes                                  | host-interaction/file-system/meta
|
| get file size                                        | host-interaction/file-system/meta
|
| read .ini file                                       | host-interaction/file-system/read
|
| get graphical window text                            | host-interaction/gui/window/get-text
|
| get disk information                                 | host-interaction/hardware/storage
|
| print debug messages (17 matches)                    | host-interaction/log/debug/write-event
|
| allocate RWX memory                                  | host-interaction/process/inject
|
| create or open registry key (5 matches)              | host-interaction/registry
|
| query or enumerate registry value (3 matches)        | host-interaction/registry
|
| set registry value (3 matches)                       | host-interaction/registry/create
|
| delete registry key (2 matches)                      | host-interaction/registry/delete
|
| delete registry value                                | host-interaction/registry/delete
|
| link function at runtime (11 matches)                | linking/runtime-linking
|
| parse PE header (8 matches)                          | load-code/pe
|
+------------------------------------------------------+---------------------------------------------+
```

Figure 2: capa results for the challenge binary

capa identifies various interesting capabilities in the program. Before we investigate these in the disassembled file, we start the program and observe its run-time activities.

Instead of running the program in a sandbox we use FLARE VM and the included analysis tools Process Hacker, Process Monitor, and FakeNet-NG. This enables us to easily control and change the analysis environment. After starting the program, we see the application window shown in Figure 3.

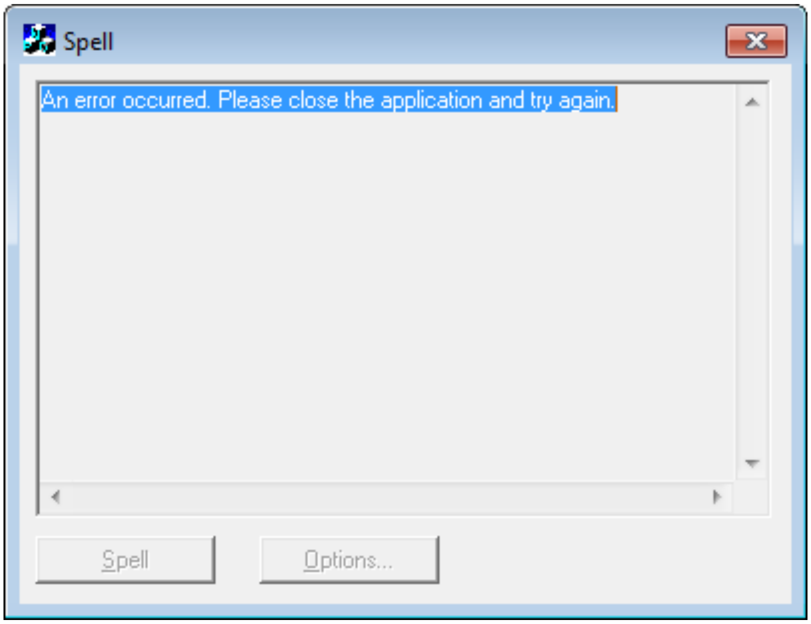

Figure 3: Challenge application window

At run-time nothing extra-ordinary happens, but after closing the application window the attentive analyst notices that the process continues to execute. If you're patient (or if your analysis environment shortcuts execution delays) eventually the process terminates but there's still no interesting activity observable in the dynamic analysis tools used here.

Advanced Analysis

After loading the binary into IDA Pro, it's time to take a break. On my system the initial analysis run took almost an hour![1] Even with IDA's library function identification there's potentially thousands of functions to analyze and it can be challenging to follow the execution flow of MFC applications. To find the interesting code sequences we use one of the verbose capa output modes (`-v` or `-vv`) or the capa explorer IDAPython plugin.

capa leads us to the suspicious function shown in Figure 4 that allocates RWX (read, write, execute) memory, contains a stackstring and links a function at runtime.

[1] Side note: compiling the binary took almost as long.

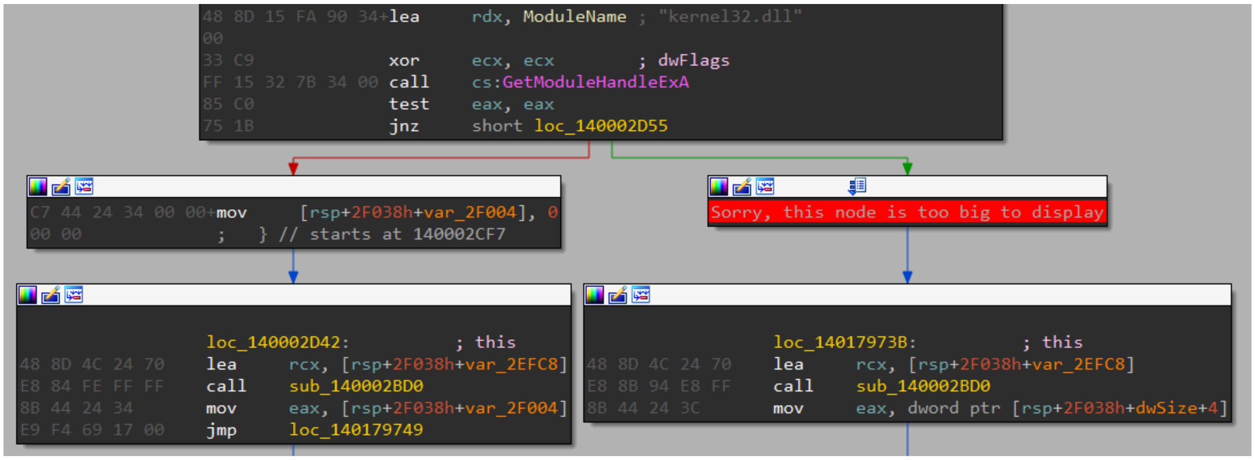

Figure 4: Suspicious function identified by capa

In graph view IDA Pro shows an unusual message indicating that a node is too big to be displayed. Switching to flat view (via the Space key) we see why. In the basic block a massive shellcode array is created byte by byte on the stack. The data is then moved to a newly allocated RWX memory region and executed as shown in Figure 5.

Figure 5: Moving shellcode array from the stack to RWX memory and executing it

Using a debugger, we can break before the call to the RWX memory and then dump the memory. Alternatively, we execute the program, suspend it, and then dump the RWX memory section at runtime. Note that the memory is not allocated until the user closes the application window. Figure 6 shows the RWX memory in Process Hacker.

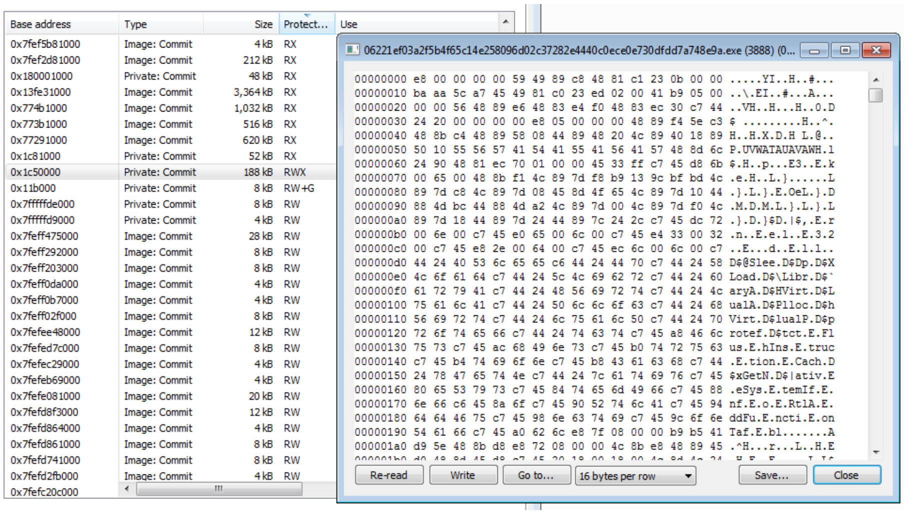

Figure 6: Viewing the RWX memory in Process Hacker

Shellcode Analysis

To see any useful strings from the shellcode we use FLOSS with the shellcode option (`-s`). The tool shows us that the file contains two DOS stub strings and a couple of stackstrings including `VirtualProtect`, `LoadLibraryA`, and `FlushInstructionCache`.

We disassemble the shellcode file as 64-bit code and notice the large function starting at offset 0x40. FLOSS can export an IDAPython script to annotate extracted stackstrings or you can use our ironstrings script whose annotations are shown in Figure 7.

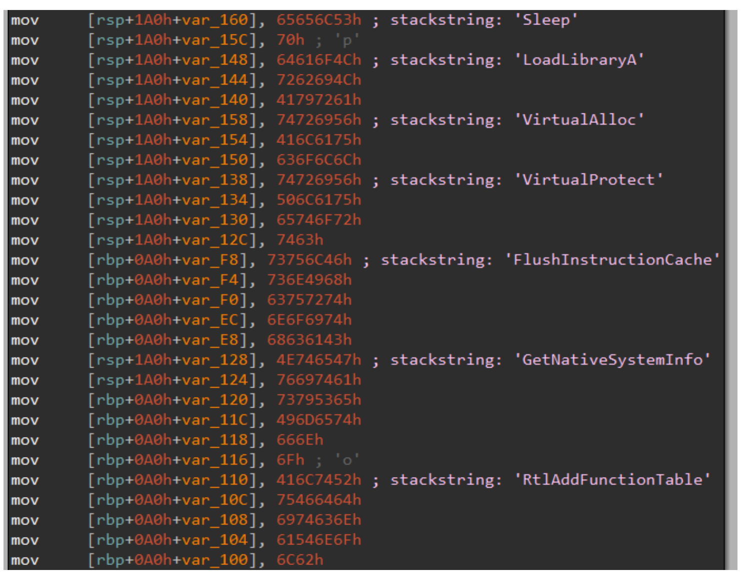

Figure 7: Annotated stackstrings in the function starting at shellcode file offset 0x40

This function loads a PE file into memory and executes it. The function receives the file data offset as its first argument in the `rcx` register. At the beginning of the shellcode `rcx` is set to the current memory location via a call/pop sequence. After adding 0xB23 `rcx` then points to the start of the PE file at file offset 0xB28. We extract the file using IDA or a hex editor.

If you encounter files with this structure in the future there's a good chance that they've been generated using sRDI which converts DLLs to shellcode. By default, these files end with the string `dave` (here renamed to `flare`).

Intermediate DLL Analysis

The extracted PE file is a 64-bit DLL. Figure 8 shows it's disassembled `DllMain` function.

```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
DllMain proc near

var_28= dword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h
arg_0= qword ptr  8
arg_8= dword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_10], r8
mov     [rsp+arg_8], edx
mov     [rsp+arg_0], rcx
sub     rsp, 48h
mov     eax, [rsp+48h+arg_8]
mov     [rsp+48h+var_28], eax
mov     [rsp+48h+var_20], 17A00h
mov     rdx, [rsp+48h+var_20]
lea     rcx, unk_1800168F0
call    sub_180001FD0
mov     [rsp+48h+var_18], rax
lea     rdx, aStart     ; "Start"
mov     rcx, [rsp+48h+var_18]
call    sub_1800027D0
mov     [rsp+48h+var_10], rax
call    [rsp+48h+var_10]
xor     ecx, ecx        ; uExitCode
call    cs:ExitProcess
```

Figure 8: Disassembled DllMain function

We skip most of the details here, but in summary the first function loads the PE file located at 0x1800168F0 (file offset 0x14EF0) into memory and the second function resolves the loaded file's export named `Start`. Before exiting `DllMain` calls the resolved export. To load the binary in-memory and resolve its export this DLL uses code from the MemoryModule project.

We again extract the embedded PE file and continue analyzing it.

Main DLL Analysis

This PE file is another 64-bit DLL. Unfortunately, we don't see many useful strings and capa doesn't provide helpful results either. So, we disassemble the file.

As expected, the DLL exports one function called `Start` which seems to implement the main functionality. We call this function `MainFunction`. Browsing through the disassembly we notice that the file uses string and API obfuscation.

Deobfuscating Strings

The disassembly in Figure 9 shows the general string obfuscation pattern. Just before using a string, the program creates a stackstring and XOR decodes it.

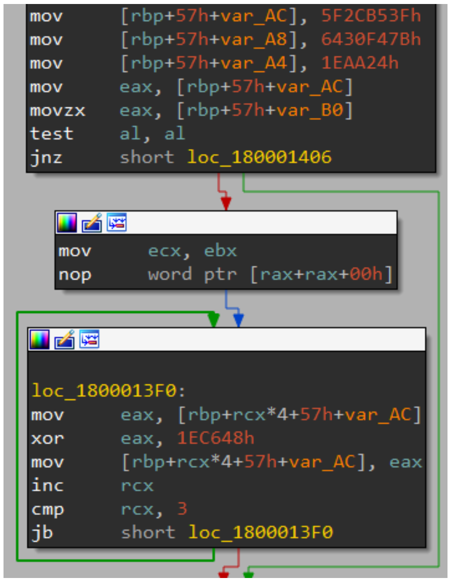

Figure 9: String deobfuscation pattern

Approaches to overcome this obfuscation include using the debugger or writing a decoding script for example in IDAPython. Here we use the script shown in Figure 10 that leverages flare-emu to semi-automatically deobfuscate strings. flare-emu integrates IDA Pro and the Unicorn emulator which is perfect for this task.

```
import idc
import flare_emu

BASE = 0x690000


def get_ea_after_xor_jb(ea):
    found_xor = False
    while True:
        if idc.print_insn_mnem(ea) == "xor":
            found_xor = True

        if found_xor and idc.print_insn_mnem(ea) == "jb":
            return idc.next_head(ea)

        ea = idc.next_head(ea)


def main():
    # user-selected start_ea, like
```

```
    # mov     [rbp+57h+var_AC], 5F2CB53Fh
    # mov     [rsp+1D0h+var_184], 667B585Ah
    start_ea = idc.here()
    end_ea = get_ea_after_xor_jb(start_ea)

    # init emulator and allocate memory
    eh = flare_emu.EmuHelper()
    eh.allocEmuMem(0x100, BASE)

    # emulate string deobfuscation code with "stack registers" set to allocated memory
    eh.emulateRange(start_ea, end_ea, registers={"rbp": BASE, "rsp": BASE})

    # read deobfuscated ASCII string
    string_ea = idc.get_operand_value(start_ea, 0) + BASE
    s1 = eh.getEmuString(string_ea)
    s1 = s1.decode("ascii")
    s = s1

    if len(s1) == 1:
        # may be a UTF-16LE string
        s2 = eh.getEmuWideString(string_ea)
        s2 = s2.decode("utf-16le")
        if len(s2) > 1:
            s = s2

    # annotate deobfuscated string
    idc.set_cmt(start_ea, s, False)


if __name__ == '__main__':
    main()
```

Figure 10: IDAPython script that uses flare-emu to deobfuscate strings

To deobfuscate a string we select the start address of the decoding sequence and run the IDAPython script. The script automatically determines the sequence end address just after the XOR loop. flare-emu emulates the instructions in the identified range and the script then adds the decoded string as a comment. An example result is shown in Figure 11.

```
mov     [rbp+57h+var_AC], 5F2CB53Fh ; ws2_32.dll
mov     [rbp+57h+var_A8], 6430F47Bh
mov     [rbp+57h+var_A4], 1EAA24h
mov     eax, [rbp+57h+var_AC]
movzx   eax, [rbp+57h+var_B0]
test    al, al
jnz     short loc_180001406
```

Figure 11: Deobfuscated string annotation after running the IDAPython script

While an analyst needs to manually run this for all string deobfuscation sequences it's possible to extend this to automatically decode all strings at once.

Resolving APIs

After exploring the first function call sequences, we understand that APIs are resolved via function name hashing. The used hashing algorithm centers around the ROL and XOR instructions. Luckily the flare-ida shellcode-hashes plugin already comes with pre-calculated hashes for this algorithm (`rol7XorHash32`, see Figure 12).

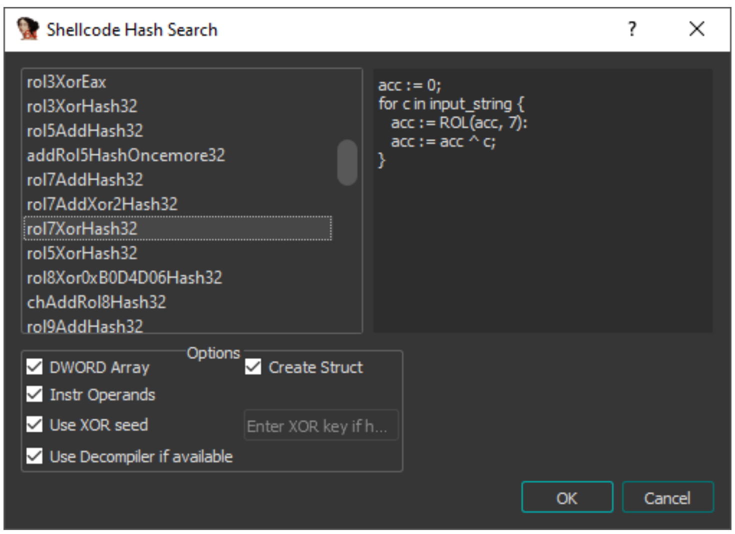

Figure 12: Running the Shellcode Hashes IDAPython plugin

The script automatically recovers and annotates about 120 locations with API names in the disassembly and in the decompiler view (see Figure 13). Thanks, Jay!

Figure 13: Recovered and annotated shellcode hashes in the disassembly and decompiler view

Core Analysis

Next, we focus on the third function called in `MainFunction`. This function is called a second time at the end of `MainFunction`. We name it `CoreFunction`. The function receives two arguments: a pointer to 0x1E0 allocated bytes and an integer.

The second argument determines the execution path taken in `CoreFunction`. For the first call the argument value is 1, so we follow the respective path first. Throughout execution there's many references to offsets into the 0x1E0 allocated bytes. To keep track of the references we create a struct named `struc_1E0h`.

Among other things the first execution path sets the following offsets in the struct:

- Offset 0x1A0: a pointer to the ASCII string `d41d8cd98f00b204e9800998ecf8427e`
- Offset 0x28: the module file path (obtained via `GetModuleFileNameA`)
- Offset 0x18: a pointer to data loaded from a resource with name PNG

`CoreFunction` then returns execution to `MainFunction`. The function called next receives the struct pointer and reads the module file path from it. The function returns `1` if the module file name is equal to `Spell.EXE`. Otherwise, the function returns `0`.

Back in `MainFunction` the program sleeps for five to six minutes before calling `CoreFunction` a second time. If the module file name is not `Spell.EXE` the second argument value is `8`. Otherwise, the program uses the value `2` defined during the first `CoreFunction` execution.

In `CoreFunction` the value `8` execution path terminates the process via the `ExitProcess` API. To continue executing the program expects to be run with the file name `Spell.EXE`.

We rename the file accordingly and perform another basic dynamic analysis run. After closing the application window and waiting for a couple of minutes the program sends a TCP packet with the ASCII character `@` (0x40) to `inactive.flare-on.com:888`.

The disassembled code sending the @ is shown in Figure 14.

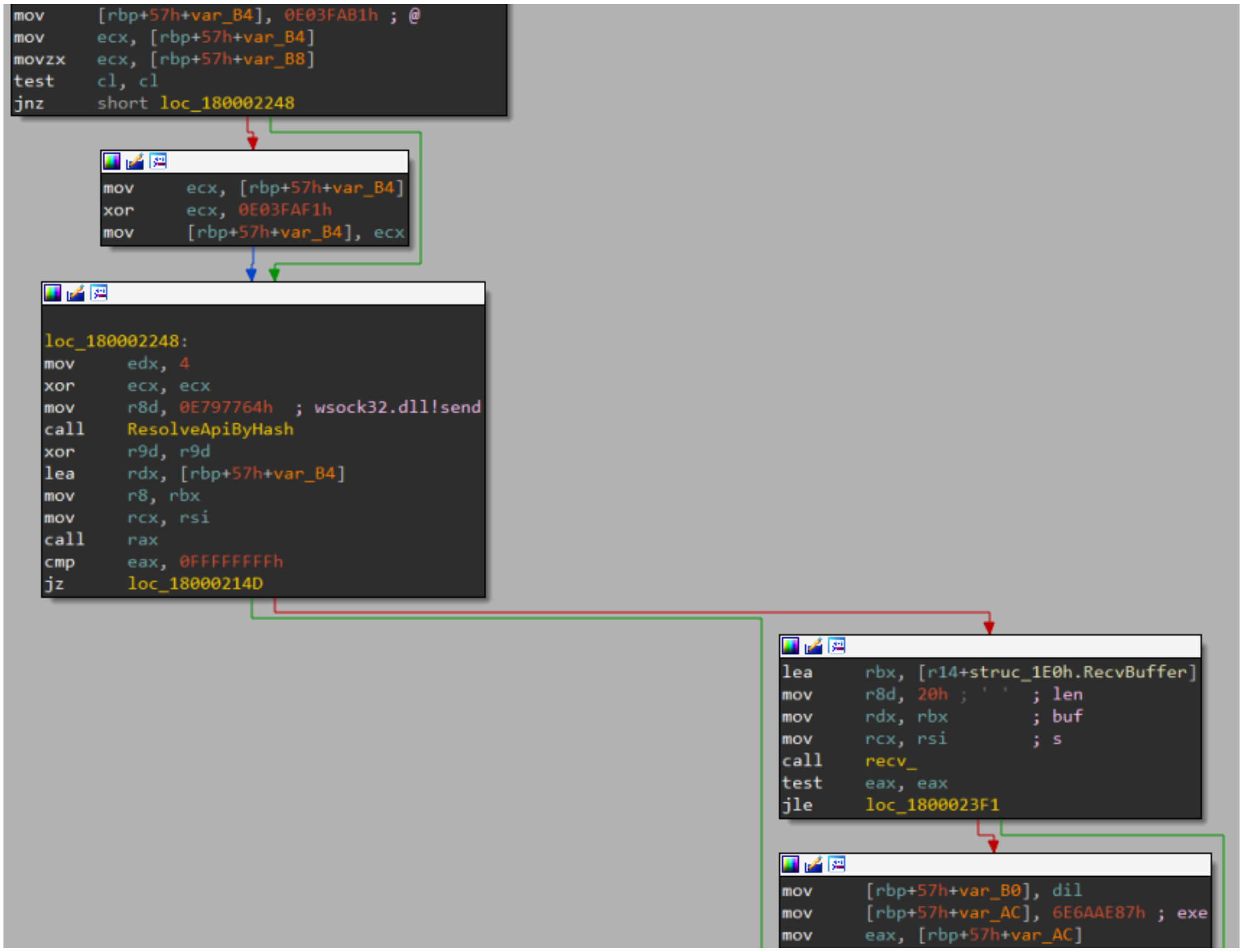

Figure 14: Disassembly of sending @ character and receiving data

After sending data, the program stores up to 32 received bytes into `struc_1E0h` at offset 0x1C0. The program then compares the received data to the strings `exe`, `run`, or `flare-on.com`. If the received data is equal to the string `flare-on.com`, the function returns 1. Otherwise, it returns 0.

We run the program one more time and now provide the expected TCP response data. This blog post describes how to set up a custom TCP response in FakeNet-NG. In short, we:

- Edit `fakenet/configs/default.ini` to enable the custom response settings via the `sample_custom_response.ini` file
- Edit `fakenet\configs\sample_custom_response.ini` to configure the `TcpRawFile` custom response via the file `flare_command.txt`
- Create `fakenet\configs\flare_command.txt` with the custom response data `flare-on.com`

Figure 15 shows the edited and created configuration files in the FLARE VM setup. Alternatively, to this approach we can use other tools like `netcat` or an interactive proxy to respond with arbitrary data.

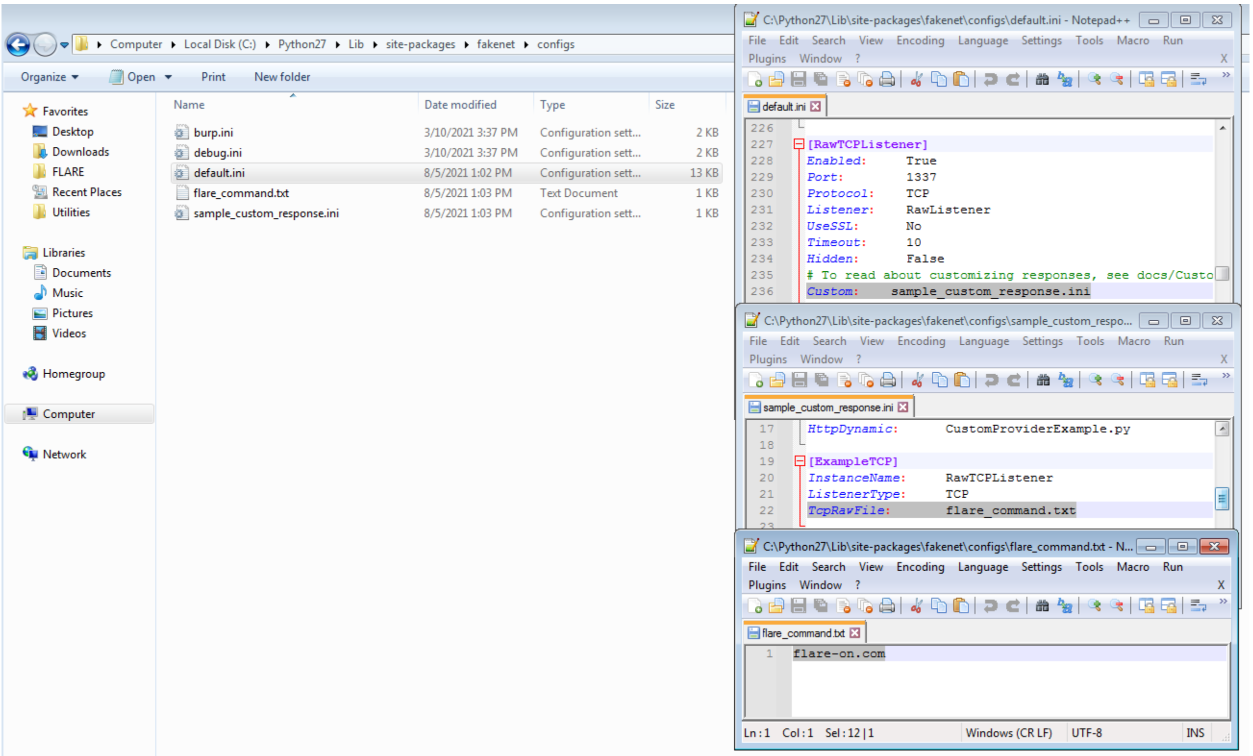

Figure 15: Configuring a custom TCP response in FakeNet-NG

Figure 16 shows the filtered Process Monitor events of this execution.

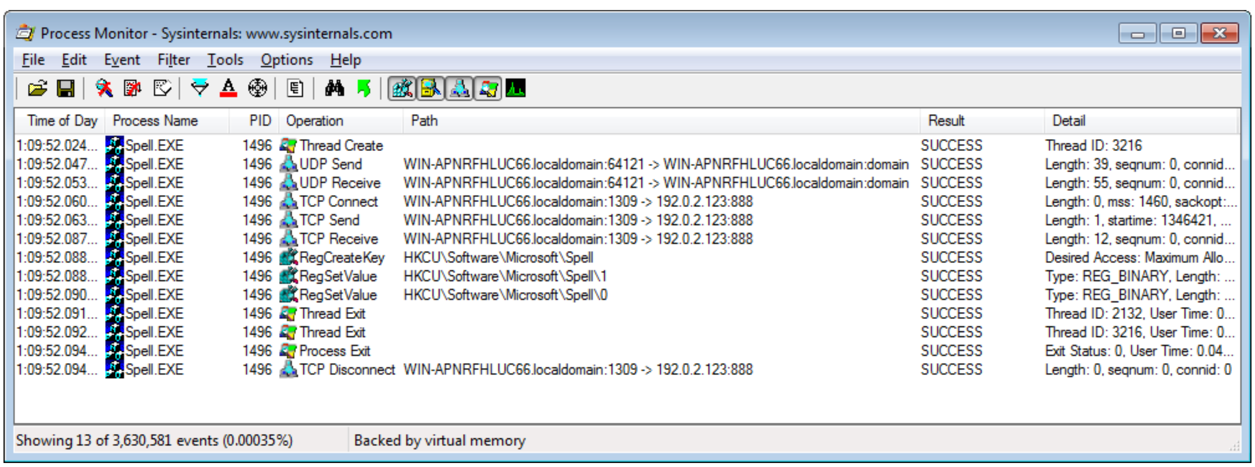

Figure 16: Filtered Process Monitor events after sending TCP response data `flare-on.com`

The program now additionally sets binary data for two registry values under `HKEY_CURRENT_USER\Software\Microsoft\Spell\` (see Figure 17).

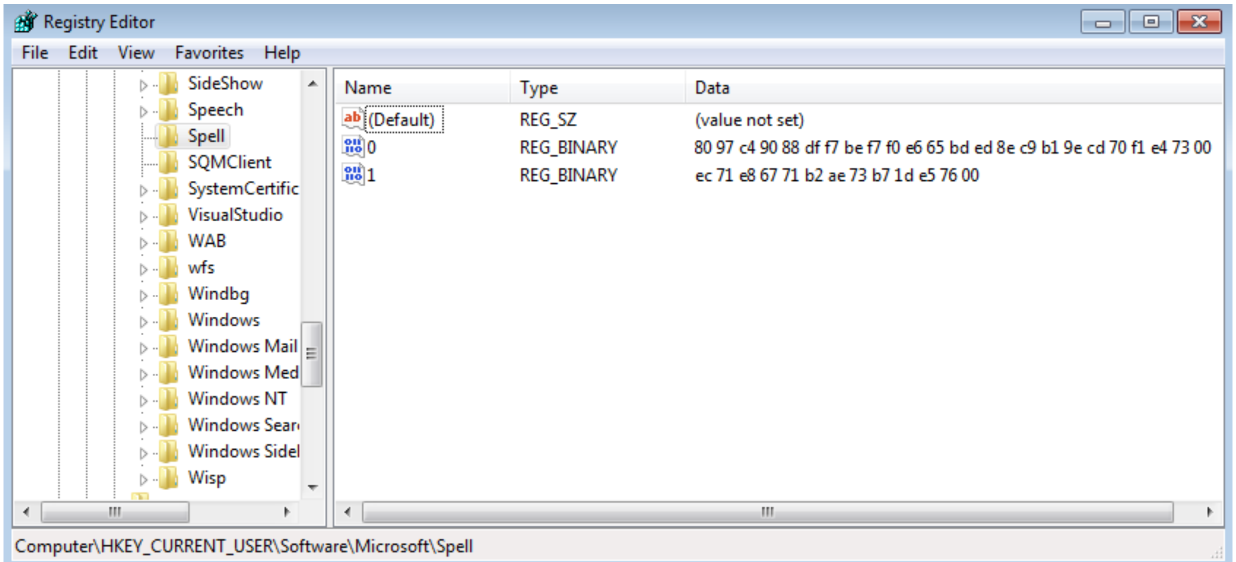

Figure 17: Registry Editor showing the created registry values

In IDA Pro we determine that there's one function that uses the `RegSetValueExA` API[2]. This function is called twice in the program. We name the function `SetRegistryValue`. `SetRegistryValue` takes four arguments: a `struc_1E0h` pointer, a data pointer, the data size, and the value name pointer.

Recovering Registry Value 1

`CoreFunction` calls SetRegistryValue to set the registry value 1. The written registry data is stored in `struc_1E0h` and XORed with a globally defined key.

After browsing to the global address of the XOR key, we use IDA's export dialog (Shift + E) to export the data as a hex string (see Figure 18).

---

[2] I recommend using the ApplyCalleeType plugin to get function prototype annotations for obfuscated API calls.

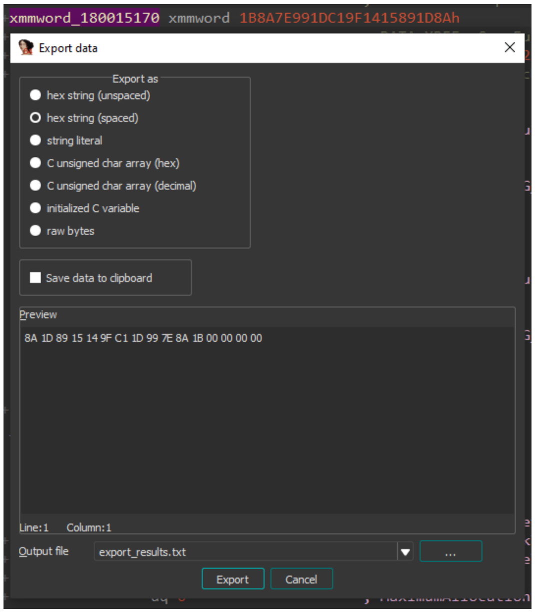

Figure 18: Exporting the XOR key as hex string

Using CyberChef we XOR the `HKEY_CURRENT_USER\Software\Microsoft\Spell\1` data with the extracted key. Figure 19 shows the resulting output **`flare-on.com`**.

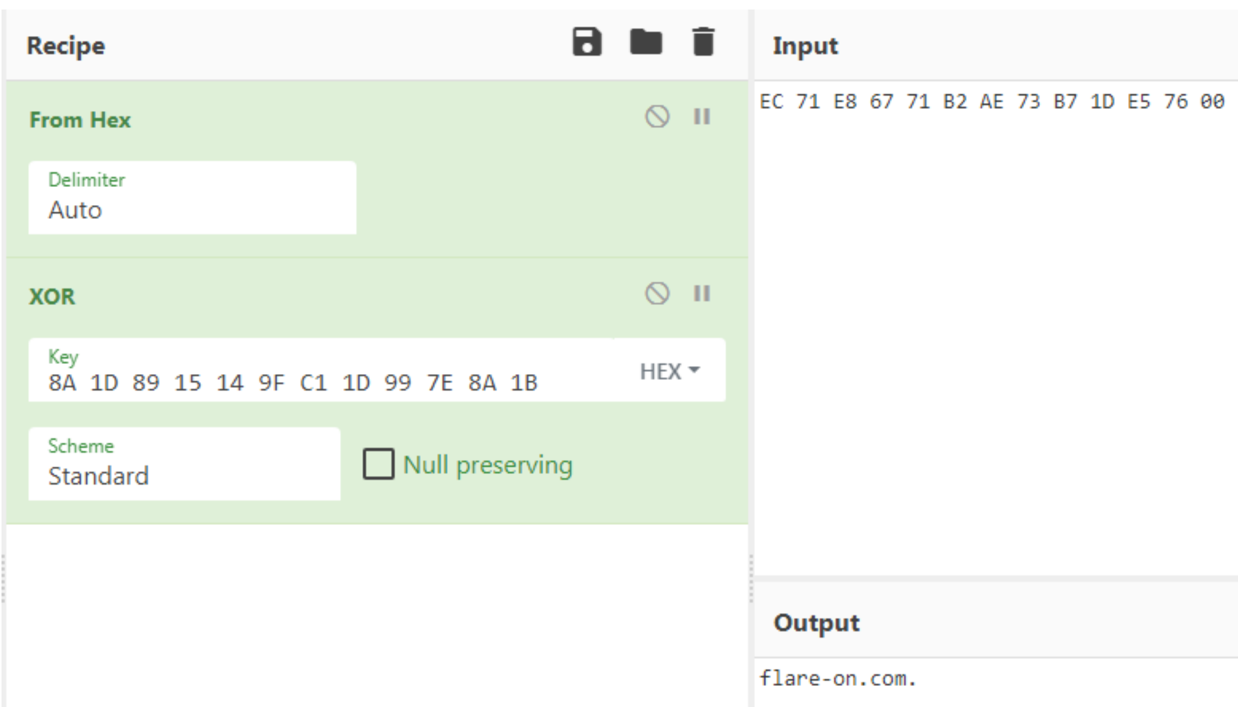

Figure 19: XOR decoding the registry data (1) using CyberChef

Recovering Registry Value 0

The function shown in Figure 20 contains a large switch case statement and then calls `SetRegistryValue` to set `HKEY_CURRENT_USER\Software\Microsoft\Spell\0`.

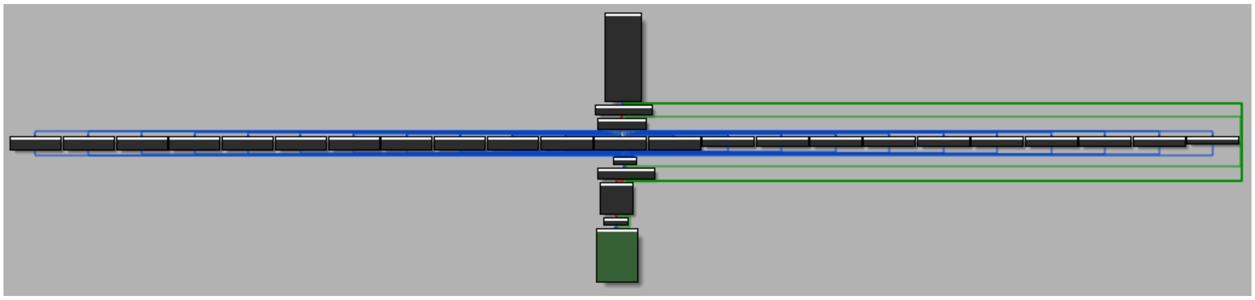

Figure 20: Graph overview of function setting registry value 0

The function first initializes the registry data it writes with globally defined bytes. The function then XORs the data byte-wise with values obtained from `struc_1E0h`. An annotated disassembly of this is shown in Figure 21.

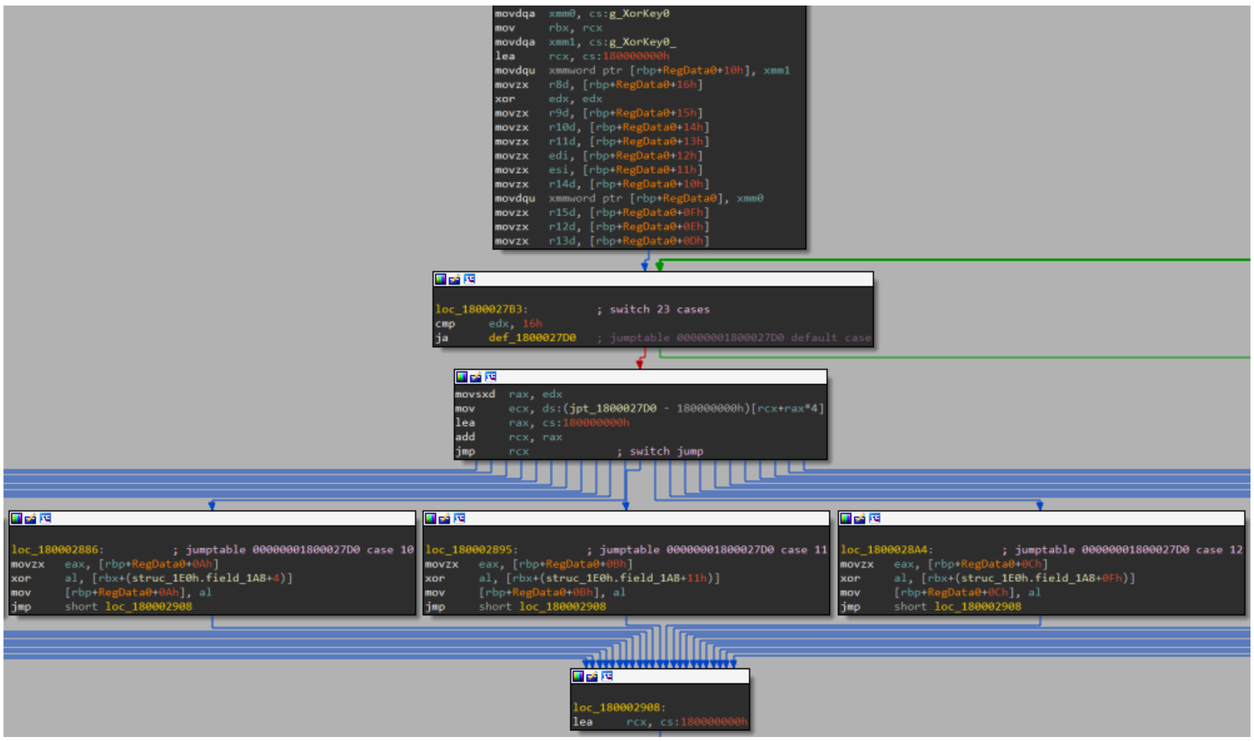

Figure 21: Data initialization and byte-wise XOR before setting the registry data

We follow the same steps as above to export the XOR key and use CyberChef to decode the `HKEY_CURRENT_USER\Software\Microsoft\Spell\0` data. Figure 22 shows the results of this.

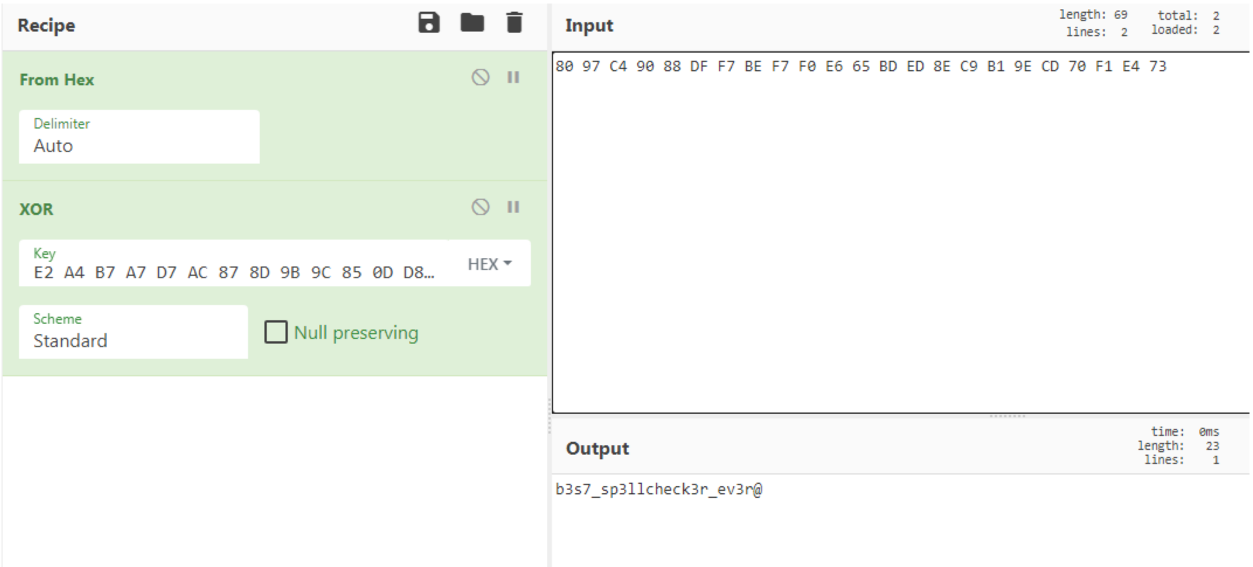

Figure 22: XOR decoding the registry data (0) using CyberChef

We combine both decoded registry values and obtain the challenge flag: `b3s7_sp3llcheck3r_ev3r@flare-on.com`.

Following the solution approach provided here we were able to skip over a bunch of details in the program. If you got lost and would like to learn more please contact the challenge author directly, for example on Twitter.