Flare-On 5: Challenge 4 Solution - binstall.exe

Challenge Author: Tyler Dean (@spresec)

Background

On the FLARE team, we reverse engineer many types of malware. One class of malware I have been looking at recently is banking trojans. Specifically, ones that use webinjects. For those not too familiar with the term webinjects, essentially it is a way to man-in-the-middle a web session with the intent to modify or steal content. For a simple example, a banking trojan may modify a banking website's login page to send login credentials to another server by replacing the POST action URL. With this background, let's start the analysis.

.NET Installer

After taking a quick look at the .NET binary, it appears to be obfuscated using the ConfuserEx[1] project. This is a common .NET obfuscator seen regularly by the FLARE team. There are a few great open-source projects that we use to deobfuscate this .NET obfuscator. NoFuserEx[2] is one such tool. NoFuserEx successfully decrypts and replaces the strings. The tool de4dot[3] is then used to rename symbols to human readable names.

After deobfuscating the .NET binary, we have readable code. From looking at the screenshot in Figure 1, we see two functions. Both delete browser cache, one for Google Chrome and the other Mozilla Firefox. This is our first hint that this challenge has something to do with browsers.

[1] https://yck1509.github.io/ConfuserEx/
[2] https://github.com/CodeShark-Dev/NoFuserEx
[3] https://github.com/0xd4d/de4dot

```
// Token: 0x0600003C RID: 60 RVA: 0x00016858 File Offset: 0x00014A58
private static void smethod_2()
{
    string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
    try
    {
        string arg = string.Empty;
        foreach (string text in File.ReadAllLines(string.Format("{0}\\AppData\\Roaming\\Mozilla\\Firefox\\profiles.ini",
          folderPath)))
        {
            if (text.Contains("Path=Profiles/"))
            {
                arg = text.Replace("Path=Profiles/", "").Trim();
            }
        }
        Array.ForEach<string>(Directory.GetFiles(string.Format("{0}\\AppData\\Local\\Mozilla\\Firefox\\Profiles\\{1}\\cache2\
          \entries", folderPath, arg)), new Action<string>(File.Delete));
    }
    catch
    {
    }
}

// Token: 0x0600003D RID: 61 RVA: 0x000168F4 File Offset: 0x00014AF4
private static void smethod_3()
{
    try
    {
        Class5.smethod_0(Environment.ExpandEnvironmentVariables("%LOCALAPPDATA%\\Google\\Chrome\\User Data\\Default\\Cache\\"));
        Class5.smethod_0(Environment.ExpandEnvironmentVariables("%LOCALAPPDATA%\\Google\\Chrome\\User Data\\Default\\Cache2\
          \entries\\"));
        Class5.smethod_1(Environment.ExpandEnvironmentVariables("%LOCALAPPDATA%\\Google\\Chrome\\User Data\\Default\\Media
          Cache"));
    }
    catch
    {
    }
}
```

Figure 1: Deobfuscated strings and renamed symbols

Looking through this binary further, we also see that it drops a payload DLL to the path `%APPDATA%\Microsoft\Internet Explorer\browserassist.dll`. It also sets a few registry keys. The registry key for `AppInit_DLLs` is a way to load a DLL into every newly loaded process that loads the `user32.dll` Microsoft library. From Microsoft's documentation: *Only a small set of modern legitimate applications use this mechanism to load DLLs, while a large set of malware use this mechanism to compromise systems. … therefore usage of AppInit_DLLs is not recommended.*[4]

browserassist.dll

The `browserassist.dll` library is a 32-bit DLL. Using a tool such as Detect It Easy[5], we also see that it was likely written using Visual Studio. Something to notice when loading this challenge binary into IDA Pro is the lack of imports to a CRT library and no FLIRT detections for these library functions. As the

[4] https://docs.microsoft.com/en-us/windows/desktop/dlls/secure-boot-and-appinit-dlls
[5] https://github.com/horsicq/Detect-It-Easy

challenge author, this was not intended. However, we can still tell IDA to search for these type signatures by loading a few signatures. To do this using IDA Pro, choose View -> Open subviews -> Signatures. I added the following signatures: `vc32_14` and `vc32ucrt`. After adding these it makes it much easier for us to reverse engineer the DLL. We don't want to waste time analyzing library code :)

The `DllMain` function is located at virtual address 0x100027D0. In `DllMain`, it first retrieves the full path of the current process using `GetModuleFileNameA`. A `strrchr` is used to perform a reverse search in the process path for the character '`\`' which gets the process name. All the characters in the process name are lowercased in `sub_100026C0` and then passed into a function that creates a simple hash of the filename. The resulting hash is then compared with a hard-coded value 0x4932B10F. This technique is common in malware to not reveal the process name. Other examples for use of this technique in malware is to hash all the process names and compare each process name hash with analysis tool names. Normally, I would port the hashing code to a scripting language such as Python and hash a large set of process names. In this challenge, we might guess that this hash might be a browser process name. In fact, it is. The resulting hash of the process name `firefox.exe` is 0x4932B10F.

After the hash check, the DLL executes another function that contains calls to `GetFileVersionInfoA` and `VerQueryValueA`. These calls are used to check the version of the current running process. In this case, we know the process is Mozilla Firefox. The check specifically queries the major version and checks if the version is less than 55 shown at virtual address 0x100027B3 in Figure 2.

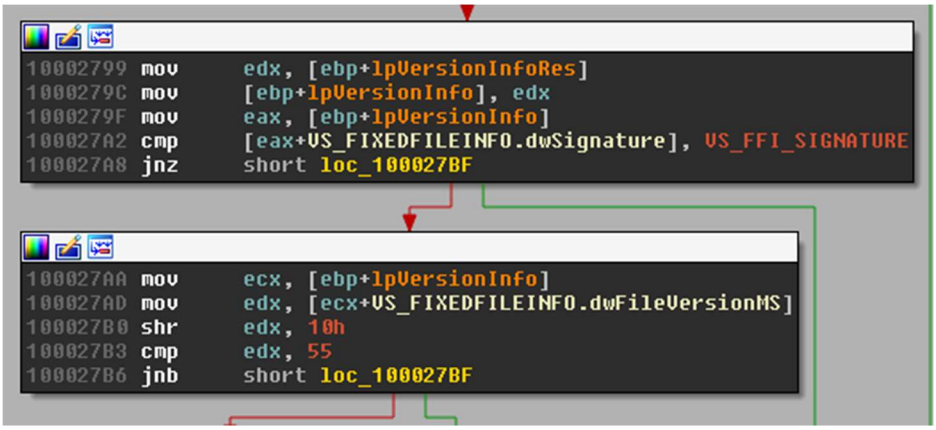

Figure 2: Checking the version number

Next, the DLL starts a new thread. This thread contains two functions. The first function is used to download a configuration, the second is used to install the function hooks to achieve the man-in-the-middle functionality of webinjects. Let's dig into how this works. But first, let's take a look at the string obfuscation techniques used in this challenge DLL.

String obfuscation

String obfuscation is common in malware to hide important strings from basic static analysis. The string obfuscation technique in this challenge uses the stack to build a string, one character at a time. This technique is often referred to as stack strings. However, this challenge takes stack strings a step further. After building a string on the stack, the string is then XOR decoded using a single byte XOR key.

When working on the malware family known as Gootkit, I encountered a similar technique. In the Gootkit family, there were a lot of these types of encoded strings and I needed to decode all of them. There are a lot of possible ways to decode these strings, but one technique is to use an emulation library such as unicorn[6].

Figure 3 is an example script that when run, gets the address from the cursor, reads the code bytes into emulator memory, and emulates until exiting the XOR loop.

[6] https://www.unicorn-engine.org/

```
import string
import unicorn
import unicorn.x86_const
from idc import *

STACK_ADDR = 0xf0000000
STACK_SIZE = 0x10000
STACK_PTR_START = STACK_ADDR + (STACK_SIZE / 2)

def is_printable(s):
    return all(c in (string.printable+'\0') for c in set(s))

def find_longest_printable_string(uc):
    res = ''
    stack_data = str(uc.mem_read(STACK_ADDR, STACK_SIZE))
    data_chunks = stack_data.split('\0\0')
    data_chunks = filter(None, data_chunks)
    for chunk in sorted(data_chunks, key=len, reverse=True):
        if is_printable(chunk):
            res = chunk
            break
    if len(res) > 2 and res[1] == '\0':
        res = (res + '\0').decode('utf-16le')
    return res.strip('\0')

def unicorn_code_hook(uc, address, size, user_data):
    if address not in user_data:
        user_data[address] = 0
    # keep track of number of times an address is executed
    user_data[address] += 1

    longest_string = ''

    if user_data[address] == 1:
        for count in user_data.values():
            if count > 1:
                longest_string = find_longest_printable_string(uc)
                break
    if len(longest_string) > 0:
        print repr(longest_string)
        uc.emu_stop()

start_addr = ScreenEA()

uc = unicorn.Uc(unicorn.UC_ARCH_X86, unicorn.UC_MODE_32)
```

```
# Read the section containing the code
addr = SegStart(start_addr)
size = SegEnd(start_addr) - addr
data = GetManyBytes(addr, size)

# Write the code bytes into the memory of our emulator
uc.mem_map(addr, size)
uc.mem_write(addr, data)

# Let's create some stack space in our emulator
uc.mem_map(STACK_ADDR, STACK_SIZE)

# Initialize
uc.reg_write(unicorn.x86_const.UC_X86_REG_EBP, STACK_PTR_START)
uc.reg_write(unicorn.x86_const.UC_X86_REG_ESP, STACK_PTR_START)
user_data = {}
uc.hook_add(unicorn.UC_HOOK_CODE, unicorn_code_hook, user_data)
uc.emu_start(start_addr, start_addr-1)
```

Figure 3: Emulation script to decode stack strings

Finding the webinjects

Using this script, we decode the encoded strings to make more sense of the purpose of this DLL. Let's continue our analysis by digging into sub_10004360, the first function called inside the first function in the thread. Here we have an extremely long stack string. After decoding, we get the following string that appears to be Base64 encoded shown in Figure 4.

```
KnpH5dyFR+pl1leFmkOft8ZcUVD0exIc2DdY92fRxFNBTWCr6cKd/U2rBp1JeJLsyqCqRXS38p3yZcbLUbe6Cs7WgJS
o9/KiPSIxTatRge1fPDplXZrdTeiiqTTqHwWm2mzzYVBcTIHiOH7Ssy1iD4HwjC5mF3g6APCpQDPAXafvTabg8wa3Z4
cwhfkf7H6N+0e6LKxonV9StCutB/6bcD/3t+DSeCa1kfqXITz3U6VcqQhS/K3mBWAHvFPVe2o7QPPAxpFj48Eyv6mXj
X9nFK9WVD7w+fA1SVVAP78jBNTQOHuXqpcFQz0IRDQNSm5T6aJa28lBJ5EgLrNNjWuzoh4f2yqeD+LpBBPxXbfy6zB3
L13/CwOFoDSwe7CnifGDB2keofpEW9P5BnH1sPSbizc/UhglELv1m5PZu3KOt3eY+IU3yOWDszmBe8fPdGhozZx3qe0
T4pZLhRV7U1VXDghNFNQ0k7+Qc+Tdkf/O81MIZJzAlDp+7kJsEM2wCsPgKzzsHYE1EyaAI+1YGHn1Uz/bxy+SzTJ2xa
djdADa7ryiROLLEhOS5rcyL5KftQvQ+WeS7fVvZAwVyZvfGDQc5UFq+/ZulieCDBeFYWcXitIp9DRfO+x28HMXORNfA
8BYFYtFuH9+f8oeruoyIckHAmzPQZLPc2NmNoTN5BTmNMsbrogQ
```

Figure 4: Decoded stack string data

However, Base64 decoding it, reveals random looking bytes. Let's look a little further. At the end of this function, we see another function that appears to decode our Base64 data. First, it gets the process filename again, using `GetModuleFileNameA` and uppercases this string. It then modifies the process

name by inserting characters into the character array. We know this character array must be `FIREFOX.EXE`, after modifications it becomes `FL@R3ON.EXE`. The large string is Base64 decoded using the library function `CryptStringToBinaryA` and then RC4 decrypted using additional Microsoft crypto APIs. The RC4 key is the MD5 hash value of the created string `FL@R3ON.EXE`.

After decoding and decrypting the large Base64 string, it contains a domain `pastebin.com` and what appears to be a URL path. The DLL then uses this data to send an HTTP GET request to the domain and URL path and uses the same string decode process to decode the HTTP response data. Figure 5 shows the decoded data.

```
{
    "fg_blacklist": [
        "[removed]"
    ],
    "injects": [
        {
            "content": [
                {
                    "code": "function readIn",
                    "after": "",
                    "before": "[removed]"
                },
                {
                    "code": "changeDir( val, tab, $input );",
                    "after": "[removed]",
                    "before": ""
                },
                {
                    "code": "} else if( model.dirList[dir] ) {",
                    "after": "",
                    "before": "[removed]"
                },
                {
                    "code": "var rendered = Mustache.render( $( template ).filter( '#' +
tmplt ).html(), vars );",
                    "after": "[removed]"
                }
            ],
            "path": "/js/controller.js",
            "host": "*flare-on.com"
        },
        {
```

```
            "content": [
                {
                    "code": "prevDir: '~',",
                    "after": "[removed]",
                    "before": ""
                }
            ],
            "path": "/js/model.js",
            "host": "*flare-on.com"
        },
        [removed]
    ]
}
```

Figure 5: Decoded webinject JSON data

The result is what appears to be a JSON webinject configuration. It is common for these webinject configurations to contain a marker of some sort and instructions for adding code before or after this marker. Looking through this configuration, we see this type of format. We see the host `flare-on.com` with URL paths and webinject instructions.

Understanding this, let's next look at the second function in the thread. The second function decodes more strings. Some of these strings include `PR_Read` and `PR_Write`. These functions are responsible for reading and writing data contents for an established session. They are often hooked by malware to perform webinjects.

At this point, we have a good understanding of what is going on. The next step is to figure out what all that ugly JavaScript is doing from the webinject config. The JavaScript looks difficult to analyze statically, so let's get this code running in a JavaScript debugger in a browser. This challenge targets Firefox versions below 55, so let's download a 32-bit version of FireFox 54. The reason for choosing the 32-bit version is that our challenge DLL is a 32-bit DLL, so it would load into a 32-bit process.

JavaScript

Once Firefox is installed and the system is infected, we browse to the `flare-on.com` site to view the results of the webinjects. Firefox contains developer tools built into the browser which are accessible

through the Developer menu item or the `Ctrl+Shift+I` keyboard shortcut. In the Debugger tab of the developer tools, we see the various JavaScript files loaded with the webinjects in place. Starting with `controller.js`, we see where the command line names are parsed and dispatched to their appropriate handler functions. For example, we see `cd`, `ls`, `sl`, `he`, and `cl`. An additional command is added by the webinjects named `su`. When the command is `su`, the `view.js` file's `askPassword` function is called. The `askPassword` function simply adds an input field with the type `password` and sets a value `model.passwordEntered` after the user types a password. Looking back at the code in `controller.js`, we see a handler in the command line parsing code for the case if the `model.passwordEntered` value is set. When this condition is true, it passes the input password to a function named `cp` shown in Figure 6.

```
function cp(p){if(model.passwordEntered=!1,10===p.length&&123==(16^p.charCodeAt(0)
```

Figure 6: Beginning of cp function

This function looks very messy, but let's break it down to see what's going on. First, we check that a password was entered and that the length of the input string (the password) is 10 characters in length. Next, we work through each character:

Character 0:

The first character must make the following equation true: `123 == (16 ^ p.charCodeAt(0))`

Using the JavaScript Console, entering the following provides us the result:

```
>> String.fromCharCode(123 ^ 16)
"k"
```

Character 1:

`p.charCodeAt(1) << 2 == 228`

```
>> String.fromCharCode(228 >> 2)
"9"
```

Character 2:

```
p.charCodeAt(2) + 44 === 142
```

```
>> String.fromCharCode(142-44)
"b"
```

Character 3:

```
p.charCodeAt(3) >> 3 == 14
```

This code snippet doesn't contain enough information to solve at this time. We can easily brute-force this character later or see if there is more information later in this large if statement.

Character 4:

Here's where things start to get interesting. This character is compared to the result of a call to `parseInt` with a messy amount of code. Using the JavaScript console, this can be evaluated:

```
>> parseInt(function() { var h = Array.prototype.slice.call(arguments),  k = h.shift();
return h.reverse().map(function(m, W) {  return String.fromCharCode(m - k - 24 - W)
}).join("")}(50, 124) + 4..toString(36).toLowerCase(), 31)
66
```

And then:

```
>> String.fromCharCode(66)
"B"
```

Resulting password

So that wasn't too bad. We continue to apply this analysis technique and arrive at the following password: `k9btBW7k2y`.

Solving the challenge

We enter the password after typing `su` and it appears that we now have `root` on the server. But where's the key? We need to go back and look at more JavaScript. If the correct password is entered, two values are set, `model.root` is set to `1` and `model.password` is set to the entered password. Looking

through the code for references to `model.password`, the function `de` uses this password in some way. When is `de` called? The function `de` is called in the `lsDir` function in `view.js`. To get to this call though, the directory to be listed must be equal to code shown in Figure 7.

```
(27).toString(36).toLowerCase().split('').map(function(A) {
        return String.fromCharCode(A.charCodeAt() + (-39))
    }).join('') + (function() {
     var E = Array.prototype.slice.call(arguments),
         O = E.shift();
    return E.reverse().map(function(s, j) {
        return String.fromCharCode(s - O - 52 - j)
    }).join('')
})(7, 160) + (34).toString(36).toLowerCase()
```

Figure 7: Encoded directory name

Using the same technique as before, we run this in the JavaScript console in Firefox and we get the string `Key`. So it appears we need to be in a directory named `Key`. We now enter `cd Key` which changes our directory into the `Key` directory. Now we just need to type `ls` to see the result.

```
Enter a command or type "help" for help.
[user@server ~]$ su
Password: ••••••••••
[root@server ~]# cd Key
[root@server Key]# ls
c0Mm4nD_inJ3c7ioN@flare-on.com
[root@server Key]#
```

Figure 8: The final key