Zloader String Obfuscation

blag.nullteilerfrei.de/2020/05/24/zloader-string-obfuscation/

born

This blag post describes my though-process during identification of the string deobfuscation method in a sample belonging to the Zloader malware family. Specifically, I wanted to identify the function or functions responsible for string deobfuscation only using static analysis and Ghidra, understand the algorithm, emulate it in Java and implement a Ghidra script to deobfuscate all strings in a binary of this family. The target audience of this post are people that have some experience with static reverse engineering and Ghidra but who always asked themselves how the f those reversing wizards identify specific functionality within a binary without wasting hours, days and weeks. ## Target Sample We will be looking at the sample with SHA256 hash `4029f9fcba1c53d86f2c59f07d5657930bd5ee64cca4c5929cbd3142484e815a`, probably created on 2020-04-08 18:19:58. According to people on the internet, this sample leverages string obfuscation, API hashing, a Domain Generation Algorithm (DGA) and, code-level obfuscation techniques like constant unfolding, dead code insertion or arithmetic substitutions to hinder analysis. Right now, we only care about the string obfuscation and try to avoid looking at anything else. The malware family was first mentioned publicly by Fortinet in mid 2016: Their blog calls it _DELoader_ based on suspected targeting of Germany which in turn is based on geo-information of IPs in log files exposed by the operators in an open directory ("DE" is Germany's country code). The post also draws a connection to a handle _Aleksandr_ and usage of the banking Trojan _Zeus_, which seems to motivate the later name _Zloader_. ## Identify the Deobfuscation function Following the list of heuristics from a previous blag post, we start at the entry point and while trying to avoid code that's too complicated, find a function that is called in a lot of other places too and, which adheres to certain requirements on data flowing into and out of it. Without even traversing into any of the functions called after the entry point, we click every function and list its references (use `X` if you have the best Ghidra Keybindings available on the free market). Here is a table of called functions together with their number of references (that is, not only calls at the entry point but in the whole binary): | Function | Xrefs | |--- |--- | | `FUN_030a3170` | 190 | | `FUN_030ba440` | 55 | | `FUN_030a3340` | 33 | | `FUN_030ba030` | 30 | | `FUN_030b8710` | 29 | | `FUN_030b1760` | 19 | | `FUN_030a3400` | 14 | | `FUN_030ba300` | 10 | | `FUN_030ba9d0` | 7 | | ... | ... | The list is sorted by the number of references and we will work our way down from the highest number of cross references (is a handy script to generate such a list of functions together with their number of cross-references). ## First Things First The first candidate is `FUN_030a3170`. It seems to receive two arguments, both of which are used in conjunction with arithmetic operators like `%` and `<`. This makes it plausible that Ghidra correctly guessed their types to be numbers. So if this is indeed a string deobfuscation function, it needs to access some sort of global variable containing the obfuscated variant of the string. In order for this function to be able to deobfuscate more than one string, at least one of the

two arguments should determine the concrete string within that global variable. But before we dive into that, let us double check that the data types of the arguments are correct by listing a few calls of the function:

```
pcVar1 = (code *)FUN_030a3170(0,0x6aa0e84);
iVar2 = (*pcVar1)(2,0);
[...]
pcVar1 = (code *)FUN_030a3170(0,uVar3);
uVar3 = (*pcVar1)(iVar2,local_23c);
[...]
pcVar1 = (code *)FUN_030a3170(0,0xfed02a7);
iVar4 = (*pcVar1)(iVar2,local_23c);
```

So the first argument seems to be a small number and the second one a large one. What is more interesting though is, that the return value of this function is not used as a string but is _called_ directly after. This suggests that the function we are looking at is not responsible for string deobfuscation but merely to resolve some API functions, potentially with the help of API hashing (see a post on API hashing if you want to learn about this technique in general). Let's rename the function to `pr_ResolveApi` and not investigate it any further, we are here for string deobfuscation! ## First Try, Second Attempt `FUN_030ba440` is a very short function that just calls `FUN_03091c50` if it didn't receive the `NULL` pointer as an argument. This function in turn calls two other functions, one of which is our `pr_ResolveApi`. The other called function seems to be junk code but overall I'm confident that this aren't the droids, we are looking for. ## Third Time's a Charm Let's take a look at `FUN_030a3340`: Ghidra determined that the function receives two pointer arguments. Looking at a few calls to this function, the first argument always seems to be a global variable while the second argument is a local array variable. So if this is a string deobfuscation function, the first argument could be the obfuscated data while the second is a pointer where the result is written to. Following data flow within `FUN_030a3340` corroborates the second part of this hypothesis: the content of `param_2` is copied to a local variable which is later returned. If the hypothesis is correct _and_ the malware sample uses an encryption scheme that needs a key, `FUN_030a3340` would need to access some global variable to be used as a key - because there is no parameter left for the key: according to our hypothesis, the first parameter is the obfuscated string while the second is an output parameter. The only global variable (shown in purple in Ghidra) within the function is `PTR_DAT_030be000`. This variable is used in two lines within `FUN_030a3340`:

```
uVar4 = (short)(char)*PTR_DAT_030be000 ^ *param_1;
[...]
uVar1 = FUN_030aba90((uint)*(ushort *)((int)param_1 + iVar5), (short)
(char)PTR_DAT_030be000[uVar6 % 0x11]);
```

Hence this variable probably contains a pointer to an array. This array is indexed with `uVar6 % 0x11` suggesting a length of 17:

```
00000000  59 49 2c 72 54 66 79 23  46 33 4d 61 71 31 33 69  |YI,rTfy#F3Maq13i|
00000010  66                                                |f|
```

At this point, instead of reverse engineering the whole function in detail, let's take a leap: In the first of the two code lines above, `PTR_DAT_030be000` is Xor-ed with the first element of the `param_1` array. Hence it may point to an Xor-key of length 17. Let's look up one of the passed arguments and Xor it with the above key: `puVar1 = FUN_030a3340((ushort *)ARRAY_030bc900,local_52);` references `ARRAY_030bc900` which contains the following data:

```
00000000  0a 00 26 00 4a 00 06 00  23 00 07 00 0b 00 46 00  |..&.J...#.....F.|
00000010  1a 00 7e 00 24 00 02 00  03 00 5e 00 40 00 06 00  |..~.$.....^.@...|
00000020  00 00 2d 00 49 00                                 |..-.I.|
```

sadly, Xoring results in:

```
00000000  53 49 0a 72 1e 66 7f 23  65 33 4a 61 7a 31 75 69  |SI.r.f.#e3Jaz1ui|
00000010  7c 59 37 2c 56 54 64 79  20 46 6d 4d 21 71 37 33  ||Y7,VTdy FmM!q73|
00000020  69 66 74 49 65 72                                 |iftIer|
```

But you might have noticed that every second byte in the alleged obfuscated data is a zero-byte. This suggest that the data is in fact a wide string with all upper bytes set to zero. After removing the zero-bytes, Xoring results in the following:

```
00000000  53 6f 66 74 77 61 72 65  5c 4d 69 63 72 6f 73 6f  |Software\Microso|
00000010  66 74 00                                          |ft.|
```

We have found a string deobfuscation function! And we could also already determine that it uses Xor-encryption with a hard-coded key of length 17. Don't forget to rename `FUN_030a3340` to something like `ev_WideStringDeobfuscate`. It is also plausible, and can be confirmed by reversing `ev_WideStringDeobfuscate` a bit more, that the obfuscated data is null-terminated. So instead of passing the length of the obfuscated data as an argument, the length is simply determined by the first occurrence of the `\0`-character. ## Due Diligence Now that we know that the global array pointed to by `PTR_DAT_030be000` contains an Xor-Key, let's check for other references. As it turns out, the only other function referencing it, is `FUN_030a3400`. This function is also on our list above (with 14 references) and we just do the same leap of faith, we did above: look up a reference to it and Xor the data passed as the argument

```
00000000  32 2c 5e 1c 31 0a 4a 11  68 57 21 0d 71           |2,^.1.J.hW!.q|
```

with the hard-coded key:

```
00000000  6b 65 72 6e 65 6c 33 32  2e 64 6c 6c 00           |kernel32.dll.|
```

There are no zero bytes in the obfuscated data, so maybe this is the non-wide-string variant of `ev_WideStringDeobfuscate`. So let's rename `FUN_030a3400` to `ev_StringDeobfuscate`. ## Automation As always, let us automate the process of finding all function references and deobfuscate the passed buffers. Roughly, we will follow this plan: * ask the user for name of deobfuscation function and parse its assembly to determine the

Xor-Key * find all calls to the deobfuscation function and determine the first argument passed to the function * deobfuscate the data and enrich the different Ghidra views (namely, assembly listing, decompiled code and the bookmarks list) Let's first look at the only part that wasn't handled in other blag posts: parsing assembly and determining the Xor-Key. Both deobfuscation functions use special `MOV` instructions - namely, `MOVZX` (Move with Zero-Extend) and `MOVSX` (Move with Sign-Extension) - to read the Xor-Key from memory. Both instructions accept two operands, a source and a destionation, and copy the contents of the source operand to the destination operand (while extending the value in some way that we don't care about). Below, are the two instructions in question from the `ev_StringDeobfuscate` and `ev_WideStringDeobfuscate` functions respectively:

```
030a3435  0F B6 30  ..0  movzx esi, byte ptr [eax]
[...]
030a3363  0F BE 19  ...  movsx ebx, byte ptr [ecx]
```

The first one copies the value referenced by the register `eax` to `esi` while the second one does the same for `ecx` and `ebx`. Since there is only one of those move instruction in both functions, our goal is to search for it and try to determine the value that was moved. This situation also already demonstrates that compilers may use different registers in very similar situation. It also means, that we need to do some extra work if we want to automate discovery of the Xor key: We cannot simply use the value from a fixed register but are merely going to iterate over all instructions within the function while tracking register values. For tracking register values let us use the following simple Java helper class:

```
private class InvalidRegisterNameException extends Exception {
    public InvalidRegisterNameException(String registerName) {
        super(String.format("Invalid register name: %s", registerName));
    }
}

private class RegisterValues {
    private int[] values;
    public boolean debug;

    public RegisterValues() {
        values = new int[8];
        debug = false;
    }

    private int nameToIndex(String registerName) throws InvalidRegisterNameException
{
        if (registerName.equals("EAX") || registerName.equals("AL") ||
registerName.equals("AH")) {
            return 0;
        } else if (registerName.equals("EBX") || registerName.equals("BL") ||
registerName.equals("BH")) {
            return 1;
        } else if (registerName.equals("ECX") || registerName.equals("CL") ||
registerName.equals("CH")) {
            return 2;
        } else if (registerName.equals("EDX") || registerName.equals("DL") ||
registerName.equals("DH")) {
            return 3;
        } else if (registerName.equals("EBP") || registerName.equals("BL") ||
registerName.equals("BH")) {
            return 4;
        } else if (registerName.equals("ESI")) {
            return 5;
        } else if (registerName.equals("EDI")) {
            return 6;
        } else if (registerName.equals("ESP")) {
            return 7;
        } else {
            throw new InvalidRegisterNameException(registerName);
        }
    }

    public void set(String registerName, int value, Address address) throws
InvalidRegisterNameException {
        if (debug) {
            println(String.format("0x%x writing 0x%x to %s", address.getOffset(),
value, registerName));
        }
        values[nameToIndex(registerName)] = value;
    }

    public int get(String registerName, Address address) throws
InvalidRegisterNameException {
        int registerValue = values[nameToIndex(registerName)];
```

```
        if (debug) {
            println(String.format("0x%x reading %s as 0x%x", address.getOffset(),
registerName, registerValue));
        }
        return registerValue;
    }
}
```

And now, we can use this class to implement the actual algorithm to search for the Xor-Key:

```
public byte[] readXorKey(Function func, int searchDepth) throws MemoryAccessException
{
    int i = 0;
    RegisterValues currentValues = new RegisterValues();
    for (Instruction instruction :
currentProgram.getListing().getInstructions(func.getEntryPoint(), true)) {
        try {
            if (instruction.getMnemonicString().equals("MOVZX")) {
                // MOVSX EBX,byte ptr [ECX]=>BYTE_ARRAY_030bc4f0 =
                // Index 0: EBX
                // Index 1: ECX
                String registerName = instruction.getOpObjects(1)[0].toString();
                int registerValue = currentValues.get(registerName,
instruction.getAddress());

                byte[] dataPtr = getOriginalBytes(toAddr(registerValue), 0x4);
                if (dataPtr != null) {
                    byte[] data = getOriginalBytes(unpackAddressLE(dataPtr), 0x11);
                    if (data != null && data.length == 0x11) {
                        return data;
                    }
                }
            } else if (instruction.getMnemonicString().equals("MOV")) {
                // MOV ECX,dword ptr [030be000 == OBFU_PTR]
                // Index 0: ECX
                // Index 1: 030be000
                String registerName = instruction.getOpObjects(0)[0].toString();
                int copiedValue = instruction.getInt(1);
                currentValues.set(registerName, copiedValue,
instruction.getAddress());
            } else if (instruction.getMnemonicString().equals("RET")) {
                break;
            }
        } catch (InvalidRegisterNameException e) {
            println(String.format("Exception: %s", e.toString()));
        }
        i++;
        if (i > searchDepth)
            break;
    }

    byte[] defaultKey = { 0x59, 0x49, 0x2c, 0x72, 0x54, 0x66, 0x79, 0x23, 0x46, 0x33,
0x4d, 0x61, 0x71, 0x31, 0x33,
            0x69, 0x66 };
    return defaultKey;
}
```

The function just returns a default key if identifying the value is not successful. Asking the user for a function, finding all references as well as tracking argument values of those calls has been covered thoroughly in previous blag posts. So the only thing left is the obfuscation itself, which is a simple Xor with a multi-byte key:

```
private byte[] cryptXor(byte[] data, byte[] key) {
    final byte[] ret = new byte[data.length];
    for (int k = 0; k < data.length; k++)
        ret[k] = (byte) (data[k] ^ key[k % key.length]);
    return ret;
}
```

As always, you can find the fully working script to deobfuscate all strings in Zloader on github. ## Summary The string obfuscation used in Zloader is quite generic: the analysed sample contains two different functions that use the same global hard-coded Xor-key to decrypt zero-terminated obfuscated data. It was possible to identify these functions without actual reverse engineering a lot of code in detail by starting at the entry point and looking at all calls sorted by number of other references of the called function. ## Deobfuscated Strings: For google-ability: | Address | Deobfuscated String |--- |--- | `0x0309111C` | `kernel32.dll` | `0x0309133F` | `Software\Microsoft` (wide) | `0x03091AFF` | `Software\Microsoft` (wide) | `0x03091CE0` | `BOT-INFO` | `0x03091CF3` | `It's a debug version.` | `0x03091D0F` | `Proxifier.exe` (wide) | `0x03091D4A` | `BOT-INFO` | `0x03091D60` | `Proxifier is a conflict program, form-grabber and web-injects will not works. Terminate proxifier for solve this problem.` | `0x0309218F` | `SeSecurityPrivilege` (wide) | `0x030923E5` | `Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.88 Safari/537.36` | `0x030928F9` | `Software\Microsoft` (wide) | `0x0309559F` | `/post.php` | `0x030955F4` | `https://` | `0x03095D99` | `C:\Windows\SystemApps\*` (wide) | `0x03095E1D` | `Microsoft.MicrosoftEdge` (wide) | `0x03095E57` | `6.3` | `0x030962E4` | `HideClass` (wide) | `0x03096332` | `HideWindow` (wide) | `0x03096348` | `HideClass` (wide) | `0x03096E37` | `.exe` (wide) | `0x03096E5B` | `.dll` (wide) | `0x03096E7A` | `.dll` (wide) | `0x03096EAB` | `.exe` (wide) | `0x03096F5F` | `.exe` | `0x03096FC8` | `>>` (wide) | `0x03097022` | `.dll` | `0x030972E1` | `Software\Microsoft\` (wide) | `0x0309741D` | `UNKNOWN` (wide) | `0x03097492` | `Software\Microsoft\Windows NT\CurrentVersion` (wide) | `0x030974A5` | `InstallDate` (wide) | `0x030974CB` | `DigitalProductId` (wide) | `0x030974F1` | `%s_%08X%08X` (wide) | `0x0309754F` | `INVALID_BOT_ID` (wide) | `0x03097790` | `rundll32.exe %s,DllRegisterServer` (wide) | `0x030977C5` | `Software\Microsoft\Windows\CurrentVersion\Run` (wide) | `0x030977F9` | `Software\Microsoft\Windows\CurrentVersion\Run` (wide) | `0x03097CF2` | `Software\Microsoft\` (wide) | `0x03098065` | `Software\Microsoft\Windows\CurrentVersion\Run` (wide) | `0x0309809E` | `.dll` (wide) | `0x03098174` | `S:(ML;;NRNWNX;;;LW)` (wide) | `0x030986F1` | `Software\Microsoft\` (wide) | `0x0309BD3F` | `.com`
Tags: ghidra - malware - string-deobfuscation - zloader