A malware analysis of ThreeDollars and WebCobra

T.E.W. Bertens
t.e.w.bertens@student.tue.nl

M.C. Crone
m.c.crone@student.tue.nl

April 10, 2019

1 Introduction

These analyses were conducted as part of the course *Lab on Offensive Computer Security* taught by Dr. Luca Allodi at the Eindhoven University of Technology. Both authors did not have any prior experience with the technicalities of malware. This project is meant as an introduction to the respective field and therefore may also be useful to others beginning their journey in the world of malware analysis as a guideline for approaching these problems.

In this paper we analyse two malware samples; a relatively simple one and a more sophisticated one, in that order. *ThreeDollars* is the respective first malware we analysed. It is known to have been used in attacks in the Middle East against governments. ThreeDollars ultimately injects a separate malware sample into a process on the victim's system. *WebCobra* is the second and harder malware sample we analysed. It is malware of the type that mines cryptocurrency on the victim's computer, thus effectively stealing computing power.

We will proceed by first specifying our methodology and set-up before the actual analyses take place.

2 Methodology

In this paper we follow the malware analysis plan as set out by Ivan Kozlov[4]. This procedure begins with *lab setup*, to be followed by *sample acquisition, pre-analysis, static analysis* and *dynamic analysis*, in respective order. In addition we base ourselves on the fundamental knowledge pertaining malware analysis given in the book by Sikorski and Honig.[5]

3 Setup

Ivan Kozlov mentions *lab setup* as the first step in his malware analysis plan. In our case, the Security research group at the Eindhoven University of Technology has already set up a laboratory where malware analysis can take place.[1] We were invited to make use of this facility as part of the project. In the respective laboratory, all malware samples we analysed in this paper were also already present. Thus the first two phases of the malware analysis plan are fulfilled, which means that we start the analysis of every individual malware sample in the *pre-analysis* phase.

[1] `https://security1.win.tue.nl/doku.php?id=lab`

4 ThreeDollars

4.1 Pre-analysis

Before starting any analysis locally, a search for the hash of the ThreeDollars malware sample is first performed in VirusTotal.[2] This results in a response as shown in figure 1.

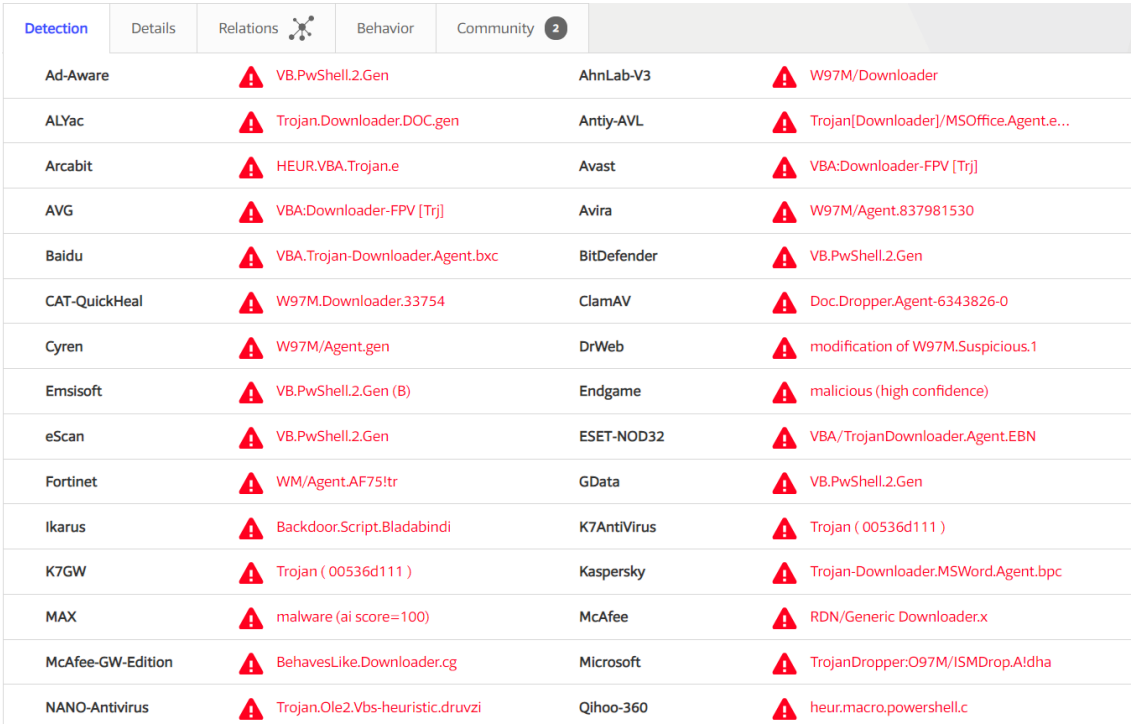

Figure 1: VirusTotal result for ThreeDollars sample

From the figure, it becomes clear that many anti-virus programs do indeed classify this sample as a trojan threat, which is the first clue as to its functionality.

Further research as part of the pre-analysis lead to the discovery of an article[1] describing technical details of the ThreeDollars sample. This articles was used as a guideline for the analysis conducted in this paper.

4.2 Static analysis

The static analysis is started by loading the sample into CFF Explorer, which retrieves basic information from the Portable Executable headers and calculates several hashes of the sample. The original sample is given as a (table 4) file, and from the output given by CFF Explorer as given in table 1, it can be concluded that it is not of a Windows executable format.

[2] `https://virustotal.com`

Property	Value
File type	unknown
File info	unknown
File size	840.51 KB
MD5	02306D629CA4092551081C4EBCBBD9B4
SHA-1	CC40A3BB20B17BB13E8B5888634EA9371D69EC01

Table 1: ThreeDollars CFF Explorer most important results

Thus the next step was to extract all static strings from the sample in order to see if any clues could be derived from it. To this end, FLOSS[3] was used. Table 2 lists some of these strings that are considered to be interesting.

$DATA = '[System.Convert]::FromBase64String(IO.File)::ReadAllText('%Base%')); [io.file]::WriteAllBytes('GBFil',$DATA); Start-Process 'GBFil"
Microsoft Shared/VBA/VBA7.1/VBE7.DL
Microsoft Office/Office16/MSWORD.OLB
large blob of base64 encoded text, preceded by ###$$$

Table 2: ThreeDollars: a few significant static strings

One of course recognizes the large blob of plain text as the payload encoded in base64. This is confirmed almost completely by the first string in the table, that signifies a conversion from base64, in order to write it as bytecode to a file. Thus, said blob will most likely be the executable payload, when converted back to bytes. Furthermore, the two strings in the middle of the table lead to the assumption that the malware sample has something to do with Visual Basic and Microsoft Word. This combination then makes a malicious macro for Microsoft Word quite likely, as Visual Basic is used as programming language for many macros. This selection of strings already gives a good grasp for the themes of the malware sample, guiding next steps in the analysis.

In order to investigate the payload, we copied it to a separate file, converted it back from base64 using a PowerShell script and wrote the resultant bytes to an executable file. Opening the executable payload in PEiD[4] confirms that it is of a valid Portable Executable format, and furthermore that it was compiled from a .NET environment. When comparing the `rsize` against the `vsize`, 0x36E00 and 0x36DAC respectively, it can be concluded that the executable is not packed, thus making one less obstacle for us analysts. However, when loading the executable into ILSpy[5], it becomes clear that all functions are obfuscated. Because we have already established that this payload was of type .NET, we try a common C# deobfuscator made available to us as `de4dot`[6], which successfully runs.

With a deobfuscated executable, we progress to actual static analysis of the code using ILSpy. The entry point gets specified by the program to be in `ns0.form0.Main`, which executes `class1.sMethod_0().run(args)`, where the respective method points to somewhere in `class4_1`. We considered following the full trail of function calls, but decided against it after exploring the haphazard way in which it jumped through the code. Instead, we scanned through all classes to look for potentially interesting behavior. We encountered a string array in `ns2.class5.sMethod_3`, and Rijndael encryption mechanisms in `ns2.class5.sMethod_32`. The analysis from Unit42[1] states that the payload dropped by ThreeDollars is basically another trojan which they tracked as *ISMInjector*. The article guided us through the most important actions of ISMInjector, which we verified in our own sample.

The ISMInjector trojan begins with copying itself to another location in order to have a more persistent foothold on the system. In figure 2 the paths for two files are declared; `%localappdata%\srvBS.txt`

[3] `https://github.com/fireeye/flare-floss`
[4] `https://www.aldeid.com/wiki/PEiD`
[5] `https://github.com/icsharpcode/ILSpy`
[6] `https://github.com/0xd4d/de4dot`

and `%localappdata%\srvHealth.exe`.

```
// ns4.Class8
using ...

private static string string_0 = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\";
internal static string string_1 = string_0 + "SrvHealth.exe";
internal static string string_2 = string_0 + "srvBS.txt";
internal static Joiner.Joiner joiner_0 = new Joiner.Joiner();
internal static Inner.Inner inner_0 = new Inner.Inner();
```

Figure 2: ISMInjector path declarations

`srvBS.txt` is the place where ISMInjector stores itself in base64 encoding such that it can always be referenced again. This is only done when the program is run for the first time and the file naturally does not exist yet. All subsequent times the procedure is skipped. This is illustrated by figure 3.

```
if (!File.Exists(Class8.string_1))
{
    goto IL_0015;
}
goto end_IL_0000;
IL_0015:
num2 = 3;
File.AppendAllText(Class8.string_2, Convert.ToBase64String(File.ReadAllBytes(Application.ExecutablePath)));
```

Figure 3: ISMInjector self copy for persistence

Two tasks are scheduled by the malware. These are named Tsk1 and Tsk2, defined in Class5.smethod_7 and Class5.smethod_24 respectively, as can be seen in figure 4.

```
if (!File.Exists(Class8.string_1))
{
    Interaction.Shell("cmd.exe /c " + Class5.smethod_7(), AppWinStyle.Hide);
    Thread.Sleep(500);
    Interaction.Shell("cmd.exe /c " + Class5.smethod_24(), AppWinStyle.Hide);
}
```

Figure 4: ISMInjector task scheduling for persistence

4.3 Dynamic analysis

These two tasks we then dynamically analysed in dnSpy. Tsk1 is scheduled to execute `srvHealth.exe` (which contained a copy of the malware) every four minutes, while Tsk2 is scheduled to save and decode the payload stored in `srvBS.txt` to `srvHealth.exe`. The behavior becomes clear from the definitions of both respective tasks as seen in table 3. They were stored as resource values, which we extracted by breaking at the correct point in the code and reading it off from memory.

Resource	Value
Tsk1	SchTasks /Create /SC MINUTE /MO 4 /TN \"ReportHealth\" /TR \"%localappdata%\\srvHealth.exe\" /f
Tsk2	SchTasks /Create /SC MINUTE /MO 2 /TN \"LocalReportHealth\" /TR \"cmd.exe /c certutil -decode %localappdata%\\srvBS.txt %localappdata%\\srvHealth.exe && schtasks /DELETE /tn LocalReportHealth /f && del %localappdata%\\srvBS.txt\""

Table 3: Resource values of Tsk1 and Tsk2

On subsequent executions, the actual functional code will be performed. This is divided in two separate dynamic link libraries called Inner and Joiner, both created by the malware. The Joiner module builds the payload by concatenating several resources together. Subsequently, the Inner module attempts to inject the payload into a process. The high level call can be seen in figure 5.

```
byte[] arg = Class5.smethod_25(800, Class8.joiner_0.Join());
Class8.inner_0.LoadDll("Run", arg, "C:\\Windows\\Microsoft.NET\\Framework\\v2.0.50727\\RegAsm.exe");
Application.Exit();
```

Figure 5: ISMInjector high level payload injection

The details of execution of the above described ISMInjector are outside the scope of this analysis, as they seem to be part of a distinct sample. Ultimately another trojan is injected, that is a variant of another malware sample known as ISMAgent Trojan[2], according to the authors at Unit42.

4.4 Conclusions

Our analysis has established the ThreeDollars malware sample to be a malicous Microsoft Word macro that is able to write a payload to an executable on the system and run it. This executable would then achieve persistence through setting scheduled tasks and finally inject another trojan into a process. The execution flow of the ThreeDollars malware sample is captured abstractedly in figure 6.

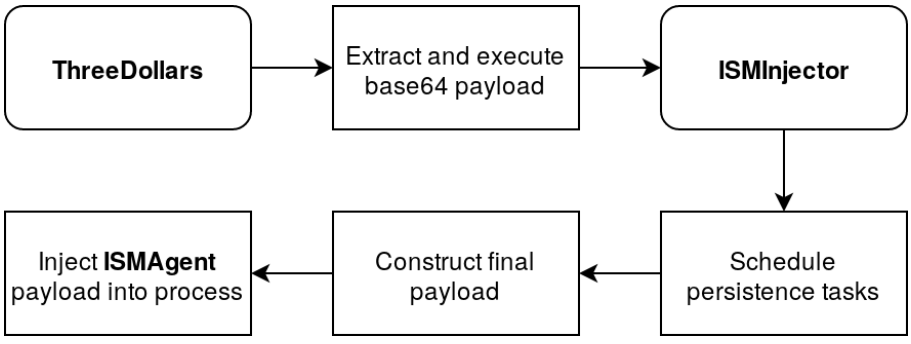

Figure 6: High-level execution flow of the ThreeDollars malware

5 WebCobra

5.1 Pre-analysis

Similarly to the analyses of ThreeDollars, an online search of the sample's hash is first performed to gain some basic understanding of the malware. A VirusTotal lookup resulted in the response as shown in figure 7.

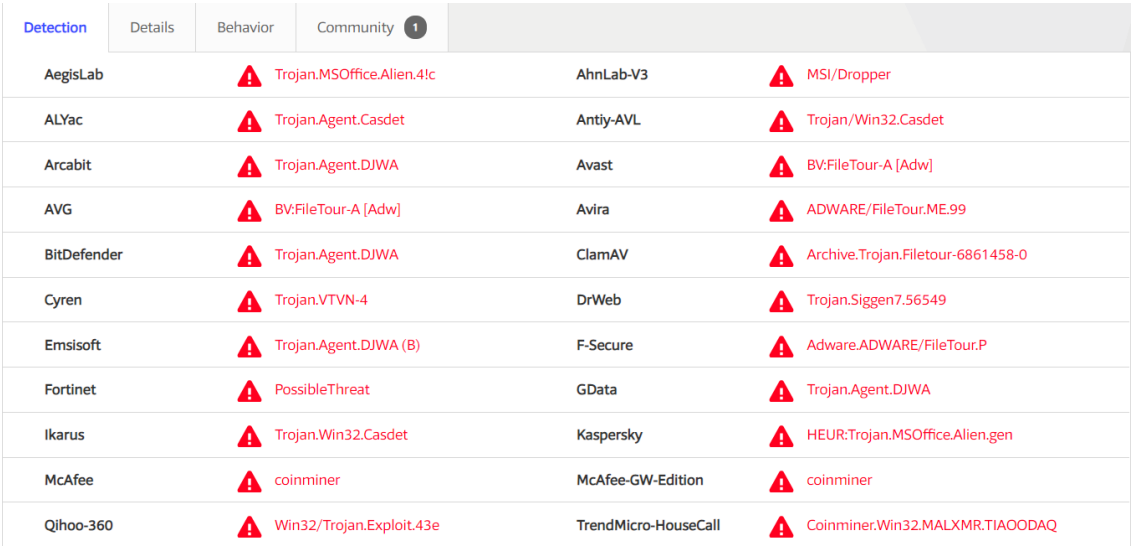

Figure 7: VirusTotal result for WebCobra sample

In these results, the sample is mostly recognised as a trojan. Furthermore, an analysis report written by security researchers at McAfee Labs[3] reveals that the true intention of the malware is to mine cryptocurrencies on victims' computers. On x64 systems, it does so by downloading and configuring legitimate mining software on a victim's computer and silently running it in the background. On x86 systems, different mining software is used and the malware does not perform a download.

5.2 Static analysis

To find out more details about the sample file, which was given as a `.bin` file, the static analysis started with an analysis of the sample in CFF Explorer. The most important results of this analysis are shown in table 4. Clearly, the sample is not a portable executable.

Property	Value
File Type	Unknown format
File Info	Unknown format
File Size	2.82 MB
PE Size	Not a Portable Executable.
MD5	B47EB604AB96F02380F64B553199A981
SHA-1	C97521AECD0D08A4827B844AB42BCB02D96ECB35

Table 4: WebCobra CFF Explorer most important results

More analysis was necessary to find out what type of file the sample was. To this end, FLOSS was used to extract all static strings from the sample file. Some significant results of this can be found in table 5.

The static strings from the sample are littered with references to a toolset called Wix[7]. See for example the first two strings in table 5. After some online searching we found that Wix is a toolset for creating custom windows installation packages, leading us to believe that the sample is actually an installer. This hypothesis is further supported by some other strings that are windows installer related, like the third string in the table. Furthermore, the strings contained a .Net copyright notice, which gives a clue as to the programming language that was used to create the sample.

C:/agent/_work/8/s/build/ship/x86/wixca.pdb
wixca.dll
Windows installer XML Toolset
.Net copyright notice

Table 5: WebCobra: a few significant static strings

To test whether the sample is actually an installer we created a copy of the sample and changed it's file extension to `.msi`. Running the new file will start up a windows installer, however, it crashes quite quickly after launch and shows an error pop-up that is in Russian. Not much could be concluded from this, though there must have been some kind of configuration that was correctly loaded, since a Russian error was produced on our English-configured virtual machine.

To further investigate we extracted the new `.msi` using 7-zip. This produced a new folder containing two files: `ERDNT.LOC.zip` and `unzip.exe`. At this point, the hypothesis that the sample is actually an installer is confirmed.

5.2.1 Unzipping ERDNT.LOC.zip

An initial attempt to unzip the zip file ERDNT.LOC brought up a window asking for a password. None of the files in the zip could be accessed without entering the correct password.

The first course of action was to investigate unzip.exe, as this is presumably the file that the installer used to extract the archive file. FLOSS was used, again, to extract the static strings from unzip.exe. The static strings contained a manual page including credits to the author of the software, along with a URL to the official web-page of the software: `www.info-zip.org`. Since this was clearly a piece of legitimate off-the-shelf software, the password to ERDNT.LOC was probably not going to be in there.

The unzip executable must be executed at some point during the installation, so the next step was to look back at the original installer file. With some direction from the McAfee article, we were led to the string in figure 8. The string is a command to launch unzip with some arguments, one of which is given after `-P`, which, from the manual page we found earlier, we knew to be the argument to pass a password. The string `iso100` was therefore concluded to be the password.

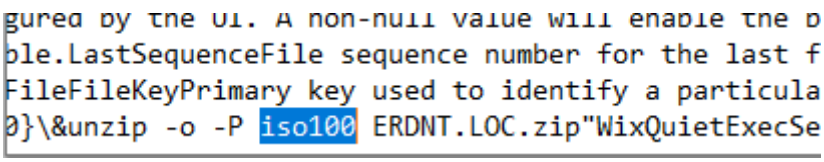

Figure 8: The password to ERDNT.LOC.zip in the static strings of the installer file.

5.2.2 Decrypting data.bin

Unzipping ERDNT.LOC.zip using the password `iso100` produces a folder with two new files: `data.bin` and `ERDNT.LOC`.

Running CFF Explorer on the data file leads to the conclusion that it not an executable. Furthermore, all static strings that could be retrieved by FLOSS were unreadable nonsense. As

[7] http://wixtoolset.org/

this initial analysis produces little information about data.bin, we turn our attention towards ERDNT.LOC.

ERDNT.LOC on the other hand, was revealed to be an executable by CFF Explorer. More precisely, it is an unpacked 32bit portable executable written in Borland Delphi (4.0). We made a copy of ERDNT.LOC and changed the extension to `.exe` to try and identify the purpose of this executable. Running it, however, did not work.

Like we did with unzip.exe, we turn our attention back to the original installer file for more clues as to the execution of ERDNT.LOC. Searching the static strings of the installer for appearances of the name ERDNT.LOC, we found the string displayed in figure 9. This string specifies a RunDLL32 command to execute the dll ERDNT.LOC using entry-point `TModuleEntry`. Clearly, ERDNT.LOC is actually a dynamic link library rather than a standalone executable, which is why it wouldn't execute as a `.exe` earlier.

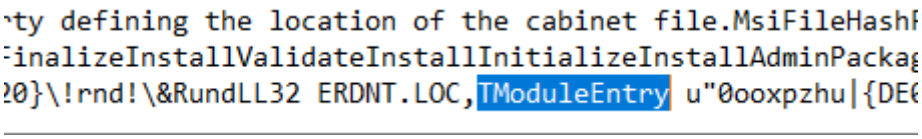

Figure 9: The entry point for ERDNT.LOC in the static strings of the installer file.

We ran the sample again, this time using the command from figure 9. To our surprise, shortly after running the execution command, all files in the folder with data.bin and ERDNT.LOC were deleted. We restored to a previous snapshot, and tried running ERDNT.LOC in a folder without data.bin. This time, execution terminated without deleting any files. This change in behaviour is evidence that ERDNT.LOC has some dependency on- or interaction with data.bin.

Further attempts at analysing the inner workings of ERDNT.LOC did not lead to a lot of new insight. The problem was that the program is compiled from Delphi, and we were unable to reconstruct readable code using the de-compilation tools available in FlareVM. The only other option was inspecting the assembly, but this is a very time-consuming process. We decided to consult the article by McAfee for more information on ERDNT.LOC and data.bin, which taught us that data.bin is actually the encrypted payload and that ERDNT.LOC is used to decrypt and run it. The data file is encrypted using a XOR based encryption, which is reversed by a hardcoded decryption operation in ERDNT.LOC. (1)

$$PlainByte_i = (((CipherByte_i + 46) \oplus 46) + 46) \ (mod\ 256) \tag{1}$$

Since attempts at running ERDNT.LOC resulted in the deletion of the sample files, the decryption had to be done manually. To this end, we wrote the following python script that loops over all individual bytes of data.bin and applies the decryption operation to compute the corresponding plaintext byte.

decryptor.py

```
import os

f_in = open("data.bin", "rb")
f_out = open("decrypted.exe", "wb")

b = f_in.read(1)
a = int(46)

while b:
    c = int.from_bytes(b, "big")
    c = (((c + a) ^ a) + a) % 256

    f_out.write(bytes([c]))
    b = f_in.read(1)
```

```
f_in.close()
f_out.close()
```

5.2.3 Unpacking the payload

Running the aforementioned python script creates a new file called `decrypted.exe`, which contains the decrypted payload of data.bin. Following standard procedure, the executable was first analysed using CFF Explorer (table 6). The results verify that the decryption was successful, since the resulting file is a correctly formatted executable. Furthermore, the "file Info" field references a piece of packing software called ACProtect by Risco Software Inc. This, together with the presence of a UPX[8]0 section in PEiD with an "rsize" of zero, proves that the executable is packed.

Property	Value
File Type	Portable Executable 32
File Info	ACProtect 1.3x - 1.4x DLL ->Risco Software Inc.
File Size	2.34 MB
PE Size	2.34 MB

Table 6: Decrypted.exe CFF Explorer most important results

Luckily, CFF Explorer has built in support for unpacking UPX executables, which we used to create the final decrypted and unpacked payload: `decrypted-unpacked.exe`. Much like ERDNT.LOC, the final payload is also written in Borland Delphi 4.0, as can be seen in table 7.

Property	Value
File Type	Portable Executable 32
File Info	Borland Delphi 4.0.
File Size	3.37 MB
PE Size	3.37 MB

Table 7: Decrypted-unpacked.exe CFF Explorer most important results

5.3 Conclusions

Unfortunately, our analyses of this malware ended here. Due to a lack of time and experience, we were unable to find meaningful discoveries in the disassembled code of the final payload. Our attempts at dynamic analysis of the payload in multiple debuggers, left us stranded in infinite loops or with self destruction of the sample. As described in the article by McAfee, WebCobra implements multiple anti-anlysis techniques that prevent the payload from running in a test environment like ours. The biggest handicap of our test setup in this regard, was the lack of an internet connection. WebCobra downloads its mining software from a remote server, which was unreachable in our setup. Hence, the code would never reach the injection nor installing of the mining software. It was simply not in our time-budget to try and circumvent these anti-analysis measures, given that the code was only available to us as assembly.

We have, however, managed to reach the actual executing payload of WebCobra. In doing so, multiple layers of protection have been bypassed, including password protection, encryption and packing. Even though there was little code involved in the analysis of this sample, the experience gained in our pursuit of the payload is valuable and touches on a part of reverse-engineering that was not very prevalent in the ThreeDollars analysis. A high-level diagram of the workings of WebCobra is given in figure 10, summarizing the findings as presented in this analysis.

[8]UPX stands for: Ultimate Packer for Executables

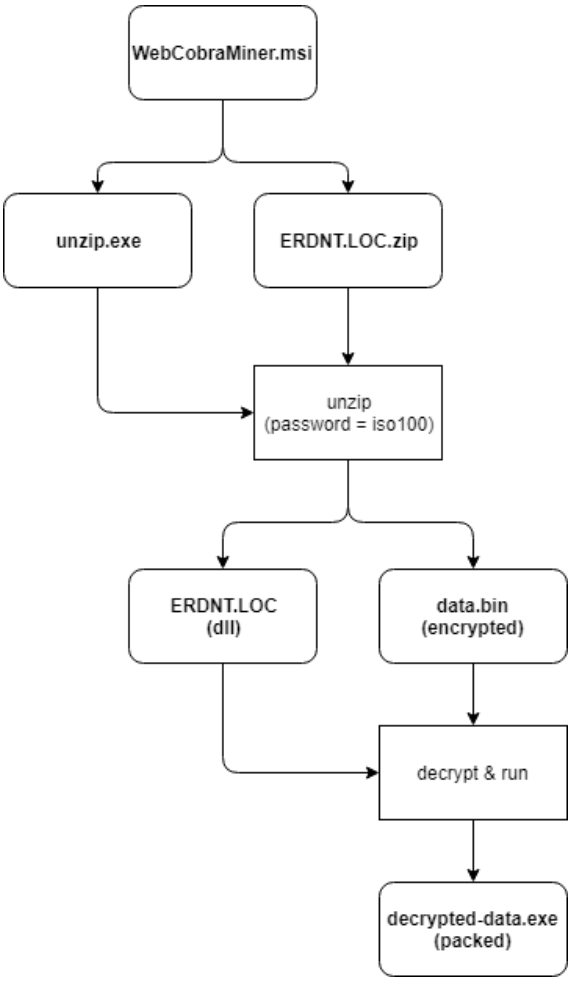

Figure 10: High-level overview of WebCobra.

6 Conclusion

This paper presented the analyses of two malware samples performed according to a generally known methodology in this field. The workings of the ThreeDollars malware sample could largely be figured out through static analysis, while only for a few aspects we did a dynamic analysis. For the WebCobra malware sample we have overcome all hurdles to get to the final payload. However, because of the multitude of anti-analysis options employed by the final executable and the lack of internet connection in the setup, we have not been able to fully analyse the retrieval of the coin miner software, its configuration and its execution. Due to our limited experience in interpreting disassembler output and the time constraints we had to work under, we were neither able to statically complete these analyses. This therefore concluded the analysis we could do on the WebCobra sample.

Ultimately, we have shown and reported our path in analysis of these two malware samples. From this first experience for us, we hope other people who are also interested in starting in the field of malware analysis can gain a head start in knowledge and motivation.

References

[1] Robert Falcone and Bryan Lee. Oilrig group steps up attacks with new delivery documents and new injector trojan. https://unit42.paloaltonetworks.com/unit42-oilrig-group-steps-attacks-new-delivery-documents-new-injector-trojan/, October 2017.

[2] Robert Falcone and Bryan Lee. Oilrig uses ismdoor variant; possibly linked to greenbug threat group. https://unit42.paloaltonetworks.com/unit42-oilrig-uses-ismdoor-variant-possibly-linked-greenbug-threat-group/, July 2017.

[3] Kapil Khade and Xiaobing Lin. Webcobra malware uses victims' computers to mine cryptocurrency. https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/webcobra-malware-uses-victims-computers-to-mine-cryptocurrency/, November 2018.

[4] Ivan Kozlov. An introduction to malware analysis. Technical report, Eindhoven University of Technology, Deloitte Nederland, 2019.

[5] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

A List of tools

- FLOSS
- CFF Explorer
- PEiD
- ILSpy
- dnSpy
- OllyDbg
- IDA (free)
- Ghidra
- de4dot
- IDR