It's A File Infector... It’s Ransomware... It's Virlock

Vlad Craciun, Andrei Nacu & Mihail Andronic

Bitdefender, Romania

Table of contents

Abstract

Win32.Virlock, with all its variations, is both a new kind of file infector and a piece of ransomware (screen-locker) at the same time. In this paper, we aim to cover the techniques used by this virus and discuss methods that can be used to detect and disinfect systems affected by it.

Virlock uses several techniques, including code obfuscation, staged unpacking, random API calls and large/redundant areas of decrypted code, to make it difficult to analyse. It also protects its code by decrypting only the sequences that are going to be executed. After a sequence of code is executed, Virlock encrypts it again. By staggering the decryption/encryption process, it ensures that a memory dump at a certain point will not reveal its features but only the piece of code that is being executed at that time.

There is also a moment in its first execution when it shifts its shape by changing certain instructions and encryption keys so that new generations will look different. Each new infection is different from any other, mostly because of the timestamps that play an important role in computing the encryption keys. Having these protection methods will also make any clean-up attempt quite a challenge. The disinfection process for this virus involves searching inside malware code for specific instruction arrangements.

We will present some ideas that could help in detecting and disinfecting a Virlock-infected system.

Introduction

Malware has grown significantly in the last decade, both in prevalence and complexity. It has developed from innocent bad jokes and simple trojans to advanced polymorphic file infectors, rootkits and ransomware. While security companies have studied all the types of malware and built specific categories for them, it can be difficult, today, to categorize a malicious application as a trojan, a piece of spyware, or even a file infector, as they tend to be more complex and to embed several different kinds of behaviour at once.

Security vendors have been forced to develop different kinds of engines to reach faster conclusions in malware analysis, be it static or dynamic, but security products by definition are usually a step behind the malware creators, even if we try to minimize that time-interval. The security industry had tried to figure out better solutions and

better engines to prevent malware execution in advance by using artificial intelligence, but no matter how hard we try, or how much time we invest in research, there is always something new which doesn't get caught. There are many cases in which we reach the conclusion that an engine is not doing the best to protect against a new piece of malware, or that making a small improvement will slow down the entire product. In some cases we reach the conclusion that a particular detection method is simply not adequate for a specific piece of malware.

1. Ransomware and file infector evolution

1.1 Old file infectors, behaviour and purpose

Known categories: appenders, prependers, EPO, polymorphic, interleaved.

Purpose:

The first file infectors were just bad jokes or proofs of concept. The earlier ones interleaved malicious code with original application code or prepended malware code to a clean application. By prepending the malicous code to a clean application, the authors increased the time needed for analysis, and also gained time for their malware to spread while users were searching for solutions. This is also a safe way to expose users' computers to hackers; file infectors act like agents, collecting confidential user data, or continuously delivering other kinds of malware to the infected system.

Behaviour:

Malicious code is executed first, infecting the system or ensuring it is running within another process or thread and eventually deploying any missing files, then it executes the original application. When a portion of the clean application is executed, the malware will also be executed at some point, this being triggered by a patched API import or by malicious code insertion. After the malicious code has finished running, the clean application's code continues to be executed from where it was left off.

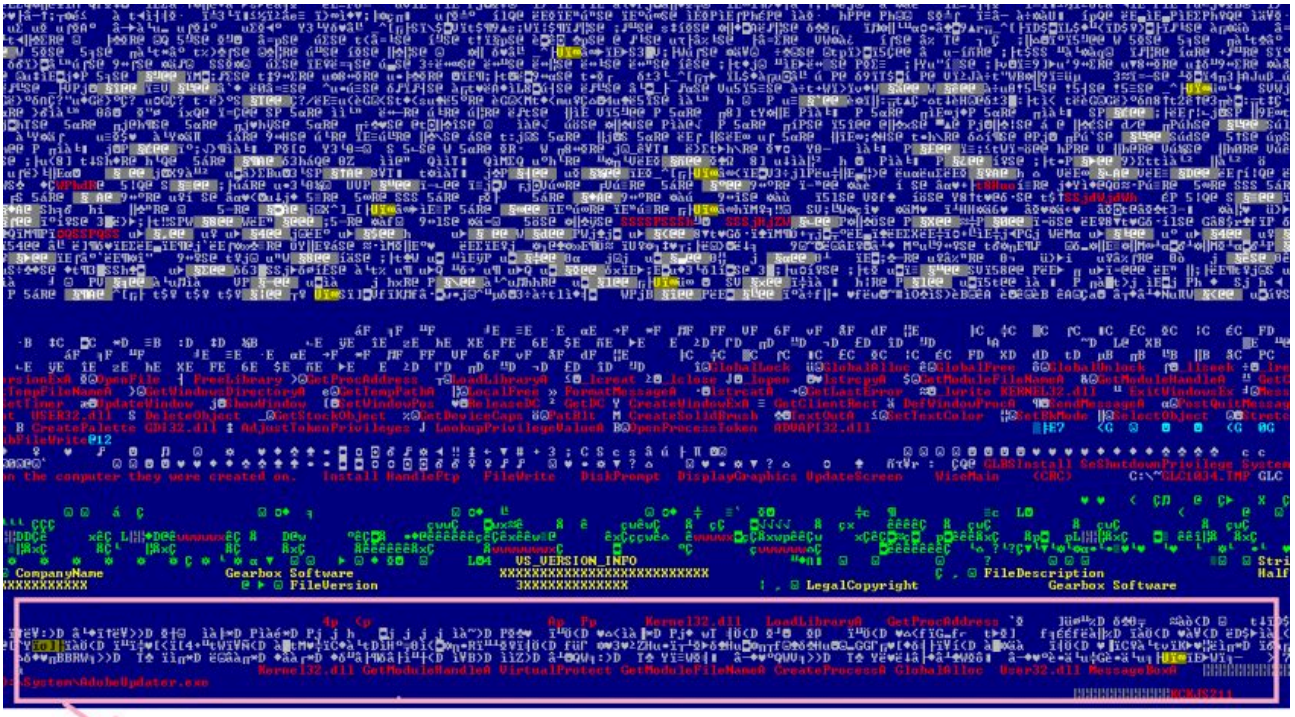

Figure 1: Example of a common file infector (appended code to clean application).

1.2 Old screen-lockers: behaviour and purpose

Purpose:

An easy way to get money from users by blocking access to their working environment. (Childish play for grownups!!!)

Behaviour:

This kind of malware creates an additional desktop and switches to the new environment, just as if another user had logged on. Some of them may encrypt user files, but most of them don't. The ones that do encrypt user files, like some CryptoLockers, do not lock the user's screen, because the damage is already at a stage where the user might wonder where the backup is, or whether a decryption tool is worth paying for.

Figure 2: Ransomware blocking user screen and requesting payment.

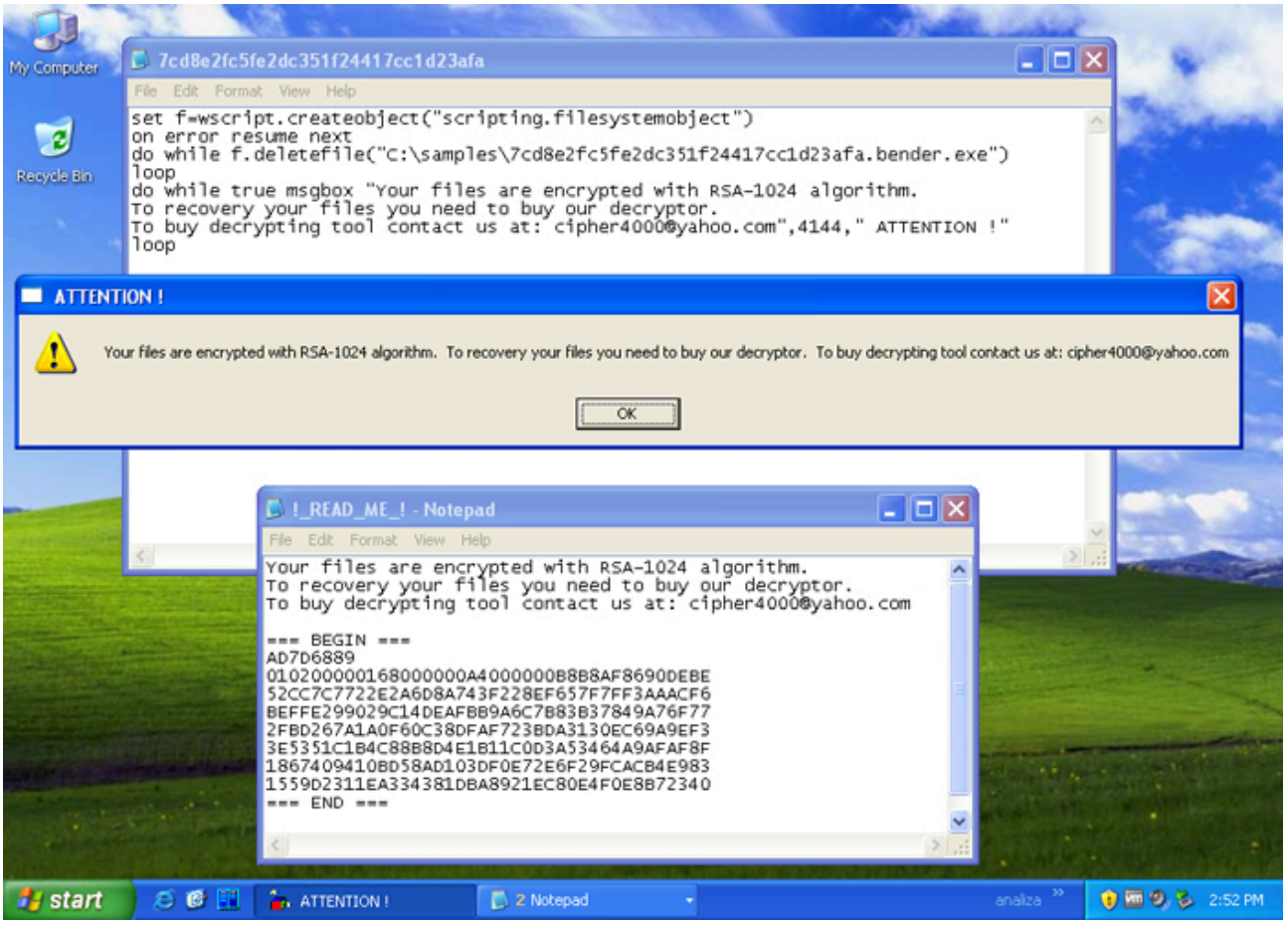

Figure 3: RSA1024 CryptoLocker displaying message to user.

Let us mention some of the well known pieces of ransomware among both families:

- ACCDFISA, PornoBlocker, Rannoh, IcePol, CryptoWall, CoinVault.

In the following chapters we will uncover the main features and components of Virlock; however we are not going to focus on the infection process. This type of malware has the vaccine within itself, but only applies it for each infected file at runtime. We will focus mainly on its design and its abilities to sneak past some security solutions.

2. Analysing Virlock, Refining Behaviour, Combining Purpose

Virlock combines the technology of file infection with the screen-locking features of regular screen-lockers. The authors embed both infection and disinfection tools, throwing away the management system to bind infected users to some private decryption keys. Their remaining concern is about users who are willing to pay their fee rated in bitcoins.

The screen-locking picture is very similar to that of those pieces of ransomware that pretend to be some higher authority with full rights to request certain amounts of money from home-users – for example as fines (see Figure 4). Most texts appearing on the locked screen are trying to scare the users, for example threatening them with prison for up to five years or more if they do not pay the money.

Figure 4: Virlock screen lock.

2.1 Analysing Virlock – refining behaviour

Virlock is changing the way in which the infection process takes place:

- It has an ingenious polymorphic engine (most file infectors don't come with such an engine), making the detection process more difficult with each infected system.
- It doesn't just insert a piece of code into the clean application as most file infectors do, but the entire clean application becomes a small piece of the malware itself (similar to Morto/Sality/ACCDFISA).
- It uses techniques to cheat users at first glance (seen in a few other pieces of malware), to bypass users' doubts that an infected file is really malicious.
- It has a lot of features (not new, but different) that make the reverse-engineering process more difficult, overload the analysts and annoy them.
- It has screen-locking (borrowed from screen-lockers) to increase the time taken to get to an infected sample – most home-users prefer to reinstall their operating system rather than trying to remove the malware.
- It uses multi-threading and rooting into the environment to get full control over the infected systems without the need for drivers, and to execute different paths inside the same application, but from different points of view (running processes/services/threads).

2.1.1 Not embedding malware code, but embedding a clean file

The infection process is somewhat different from the infection process of other known file infectors. However, there are small similarities between Virlock and both the Sality file infector and the ACCDFISA ransomware:

- Virlock and Sality: both replace the clean application with the malware which contains the original application packed or modified.
- Virlock and ACCDFISA: ACCDFISA uses the RAR archiver to make all the infections self-extractable – this is very similar to Virlock's behaviour but with the small difference that Virlock uses its own techniques to accomplish the same behaviour.

2.1.2 Anti-analysing techniques

At the moment we know about five different Virlock versions. They're not too different but they do differ in such a way that some simple checks will not catch them all.

2.1.2.1 Code obfuscation

One of the main techniques used to harden the reverse engineering and analysis process is obfuscation.

Obfuscation is present in all five versions and is similar between some and different between others. However, while obfuscation may contribute to detection, it is not a key-point in doing that.

Figure 5 shows some screenshots of obfuscated code from four different versions.

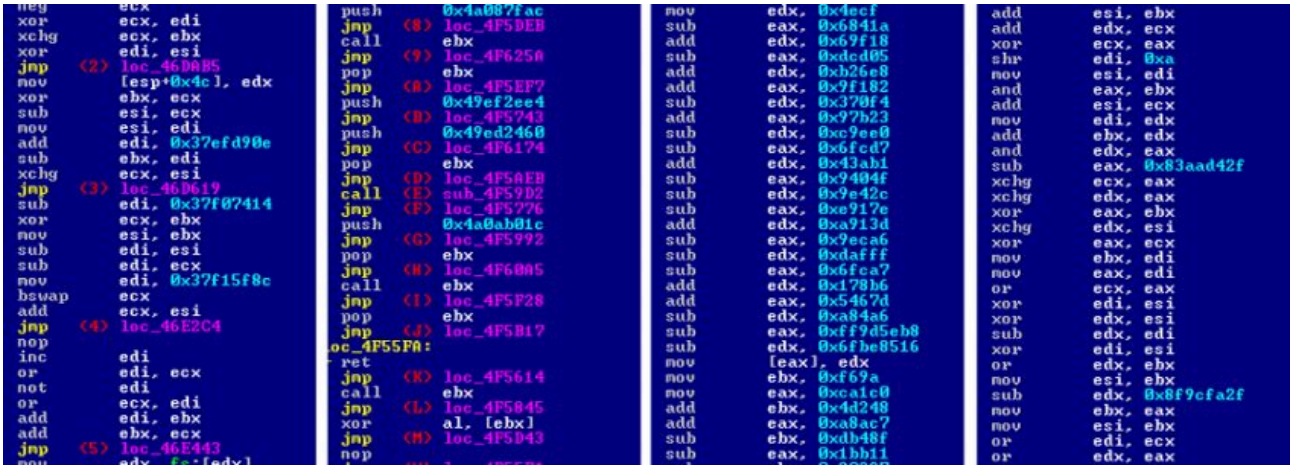

Figure 5: Obfuscated code inside four different Virlock versions.

If we are going to trace the entropy of those pieces of code, or count the number of some target instructions which repeat excessively, we can create some checkpoint conditions that Virlock infections will not pass. Code can be obfuscated in lots of configurations, but some of them are built based on some basic principles. It is not too difficult to observe the criteria with which an obfuscation engine was built.

We could also de-obfuscate some instruction blocks by following the true aim of an obfuscated piece of code. However, de-obfuscation becomes irrelevant when one can look at the execution traces. They are still a plus when building documents to reveal the true meaning of some code.

Obfuscation also contributes to making the static analysis procedure more difficult.

2.1.2.2 Anti-debugger

There are lots of anti-debugger techniques, and usually, malware creators combine those features with techniques to detect virtual machines, emulators or supervisor tools like *PIN* from *Intel* (which allows one to instrument an executed application), or API loggers which inject tracing modules or pieces of code into a target process.

Virlock does not combine all of these, but it uses the strongest of them all, in order to bring the analyst to a point where he/she could easily give up.

Multi-staged unpack

This is a known technique for making the reverse engineering procedures harder, for both static and dynamic analysis. If a piece of code is unpacked piece by piece, one at a time, while it is executed, then performing a static analysis could be very difficult. Following the modifications inside a debugger might also be tricky, as some debuggers simply refuse to disassemble the code at the point where they think that there is no code in the first place. If we add to that the fact that code might re-encrypt the previously executed code, then things get really interesting.

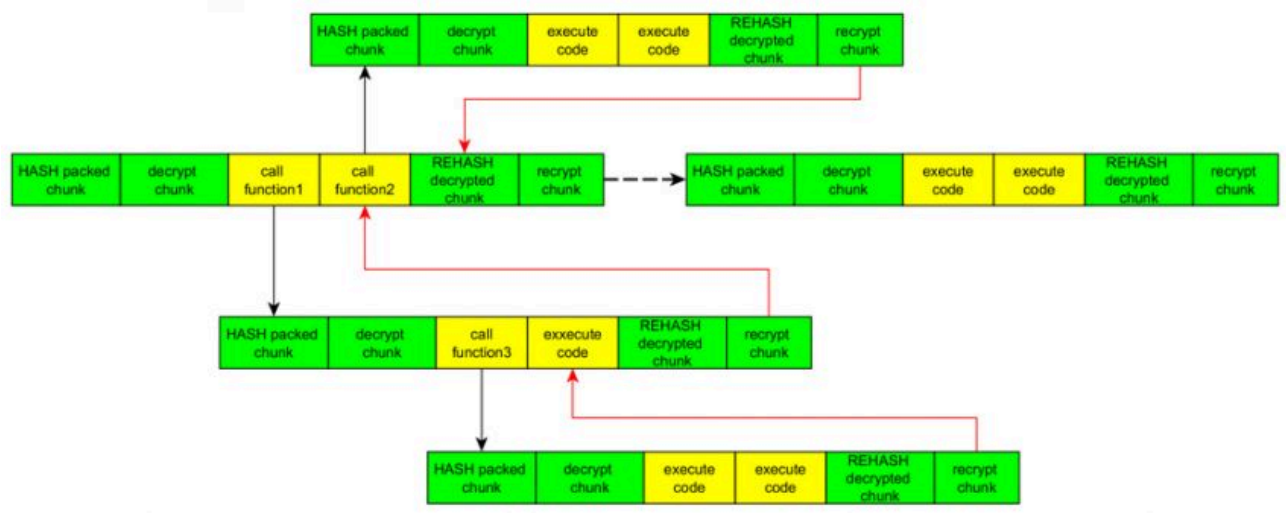

Figure 6: Short example of execution flow, following the chunk encryption/decryption template.

Staged unpack

Staged unpack is a feature which minimizes the ‘area’ of ‘plain-text’ code at any time. There is a piece of code, more like a template, which repeats itself along the execution of the malware, and at each step:

- It hashes the buffer to be unpacked
- It decrypts the next piece of code, only if the hashes match
- It executes the code inside the decrypted chunk (possible more function-templates)
- It rehashes the unpacked code and alters the hash, inside the code
- It re-encrypts the previously decrypted code.

The template follows the data structure of a linear linked list, where each node is itself a linear linked list of many possible function calls. We are seeing linked lists inside linked lists mainly because each function call inside

such a code-chunk calls another unpack-execute-repack template.

Figure 7 shows the code template for the mentioned trick inside a particular infection, which starts by checking the integrity of the packed chunk-code at 40193F, decrypts the buffer at 4019C0, jumps to unpacked code at 401A7E, and finally rebuilds the HASH for the unpacked code which it overwrites at the beginning of the code template and re-encrypts the entire code starting at 401A7E.

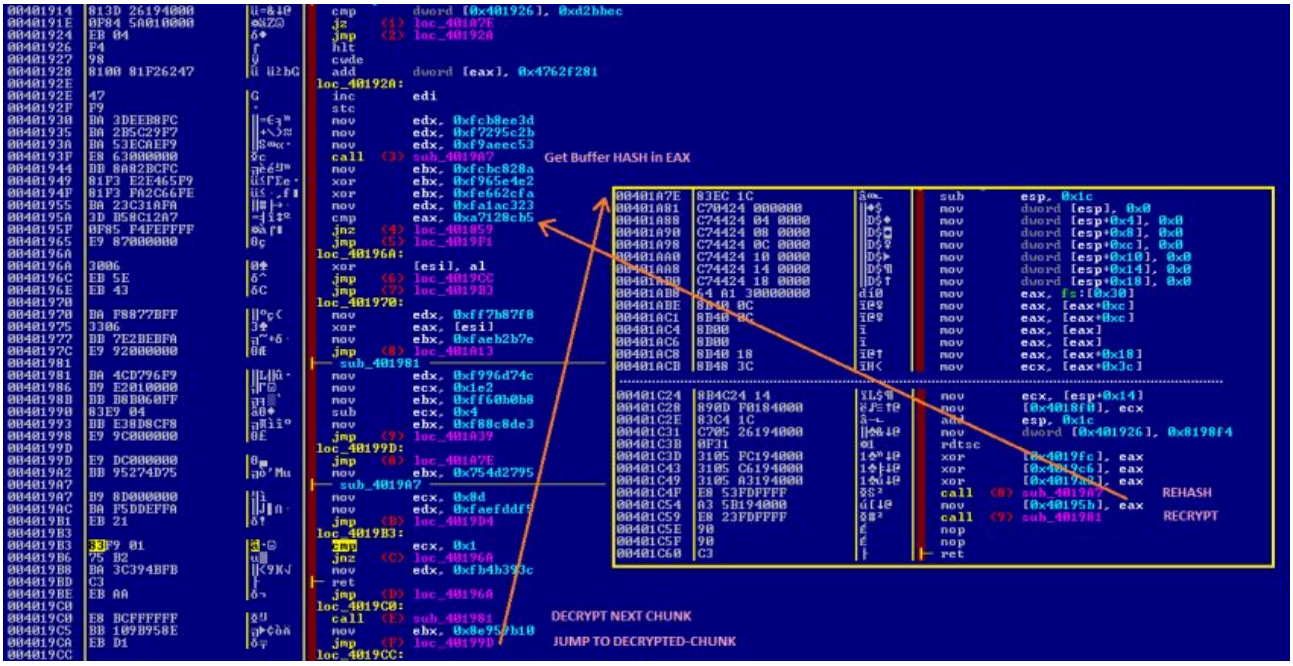

Figure 7: Template-code for staged unpack (yellow square -> unpacked code).

If someone is trying to make some process-dumps to have a look at the code inside the malware while it's executing, they might be surprised to find that the malware is almost fully packed, just as it was in the first place. The surprise gets bigger, as one is thinking that the malware might have some running threads which did not get dumped at the time of the process dump and while trying to grab all the memory pieces, one will obtain nothing more than the first process dump.

Checking for the presence of a debugger

Every infected sample checks for the presence of a debugger at some point. There is a standard way to do that, which is by querying a flag inside PEB, called isDebuggerPresent at [fs:[30h]+2], bit 0 (see Figure 8).

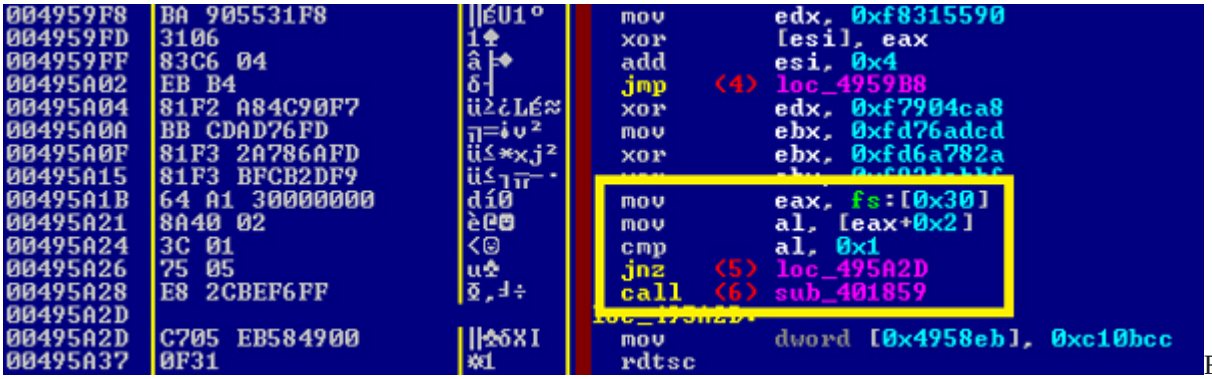

Figure 8: Anti-debugger checking inside PEB.

In our example, if it's being debugged, the code jumps to 0x495A2D . If we are taking a closer look we can see in Figure 9 that the code is being executed in those conditions.

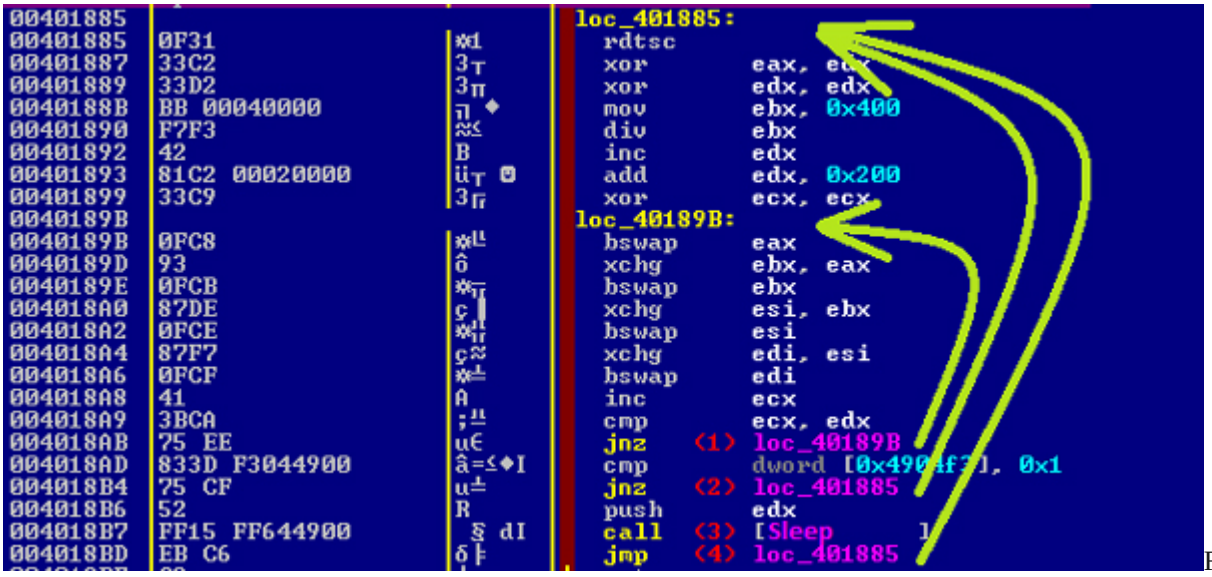

Figure 9: Code executed when debugger is found.

Eventually we find a piece of code looping on itself and calling Sleep.

Most of the time, we can trick the application by changing the condition flags; and thus the condition itself or the value being compared. However, the time spent getting one's hands on that piece of code is sometimes too much to continue with the dynamic analysis that way.

Rooting inside the execution environment

We mentioned earlier that the malware does not use all known methods to harden the analysis procedure, but it uses the strongest of all methods gathered together to at least discourage analysts or to create problems for automated tools.

The technique described in this section does not refer to a behaviour that rootkits are using, but rather to a behaviour which spreads the infection inside the infected system, making self-copies and additional processes or services, each of them with a couple of threads. If the malware gets to execute inside such a configuration, then the synchronization policies between processes and threads will enable it to do its main job, otherwise one will not get anything useful from it.

At the beginning of the execution, an infected sample will first create two copies (of the original infection core – morphed) inside hidden folders with random names but constant length (eight characters), one located in %AllUsersProfile% and one inside %UserProfile%:

[%UserProfile%\[a-zA-Z]{8}\[a-zA-Z]{8}.exe]

[%AllUsersProfile%\[a-zA-Z]{8}\[a-zA-Z]{8}.exe]

The copy located in the %UserProfile% folder is executed first using CreateProcess and it is also set as a starting point inside the startup key:

[HKCU\Software\Microsoft\Windows\CurrentVersion\Run].

Second and (in some cases) third copies are written in the %AllUserProfile% folder inside different subfolders. One of them is executed like the first copy in order to work together with it (one of the copies ensures that the other is not killed, and if that happens then it just recreates it), and the other is created as a service to supervise some tasks and gain privileged access to operating system components.

It is important at that point to note that the malware copies are not only different from the first one (using a polymorphic packer), but also have some key-flags changed. The changing of flags will enable, for example, one of the copies to execute a slightly different path inside the malware just like a switch-case block. For example, the malware self-disinfects the file inside it, only if a certain flag located at a hard-coded address says that this can be done.

A series of batch-files and VBS scripts are written on the disk temporarily to help the malware infect files by first making a backup and then overwriting the target file. Scripts are also used to change security policies inside the registry, in order to hide the malware or to disable default security features.

The following is a list of commands altering registry entries:

reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced /f /v HideFileExt /t REG_DWORD /d 1

reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced /f /v Hidden /t REG_DWORD /d 2

reg add HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System /v EnableLUA /d 0 /t REG_DWORD /f

Straight after the installation, the malware tries to brute-force the user logon account password with at least a few thousand common password templates, and straight after that creates a new user with a random name and full administrator rights.

Figure 10: New account created by Virlock after successfully brute-forcing the administrator password.

The following are just a few examples of passwords that had been tried by the malware:

password, P@ssw0rd, 1234, Password1, 123456, admin, 12345, Passw0rd, p@ssw0rd, Pa$$w0rd, !QAZ2wsx, test, sunshine, P@ssword, 1qaz@WSX, 123456789, 12345678, abc123, qwerty, letmein, changeme, master,

Password!, passw0rd, 1q2w3e4r, Password01, password1, hunter, qazwsx, welcome, Welcome123, secret, orig_Administrator, princess, dragon, pussy, baseball, football, monkey, 696969, operator123, N0th1n9, !qaz@wsx, 1q2w3e4r5t6y7u8i, abcd12345, 7654321, Administrator, q1w2e3r4, q1w2e3r4t5.

A process created with the following command line will discard any possible API-tracer or debugger following the process execution. However, we can still trick such behaviours by altering the code at the entry-point and forcing a debugger to enter first, modifying the parameters for CreateProcess, or using some advanced environment emulators:

```
CreateProcessW("%TEMP%\AccMwMEs.bat", " "%TEMP%\AccMwMEs.bat" "C:\samples\virlock.exe" ", ……)
[AccMwMEs.bat]
echo WScript.Sleep(50)>%TEMP%/file.vbs
cscript %TEMP%/file.vbs
del /F /Q file.js
del /F /Q %1
del /F /Q %0
```

When an infected sample gets to execute on a clean system, we say that the sample is the original one which is the primary cause of the infection. This sample is almost like any other fresh infected sample, which was not executed after the infection. There are some flags hard-coded into the malware so that it knows, at runtime, whether the sample being executed is a fresh infection that has not been executed before, or a drop made by malware targeted as a service or a malicious process running on the user's system. Figures 11 and 12 illustrate that behaviour.

Hard-coded value	Meaning
0	Installed malware process, usually two synchronized processes
1	Original sample, installs malware components
2	Intermediate actions (while rooting into environment), brute-force user account password
3	Multithreading and synchronization (screen-locking, online payment)
4	Sample is running as service

Table 1: Associations between hard-coded values and their meaning.

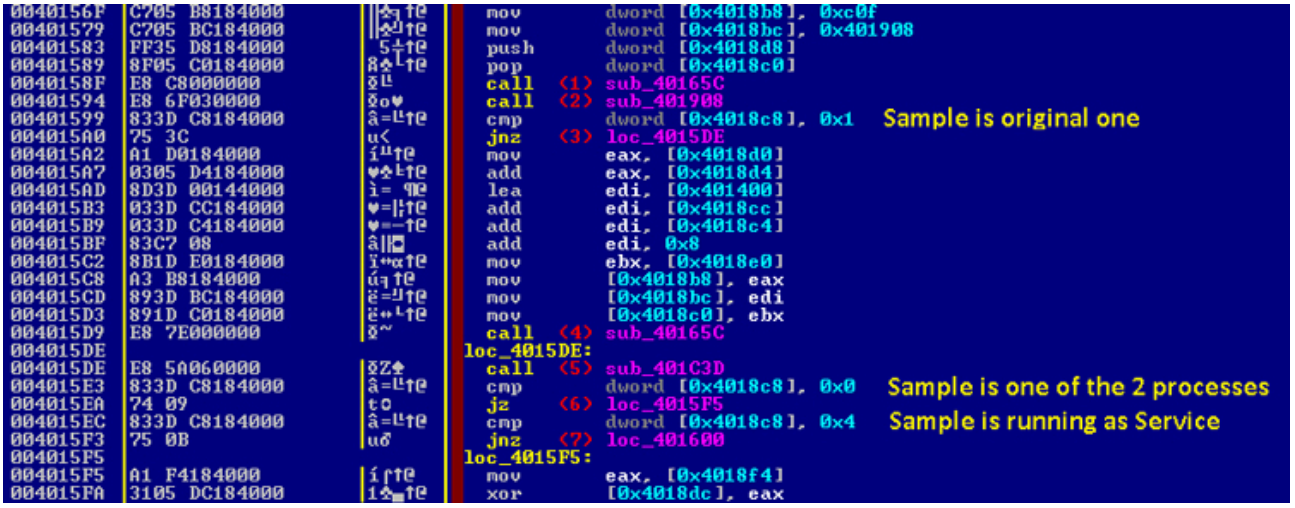

Figure 11: First context switching actions.

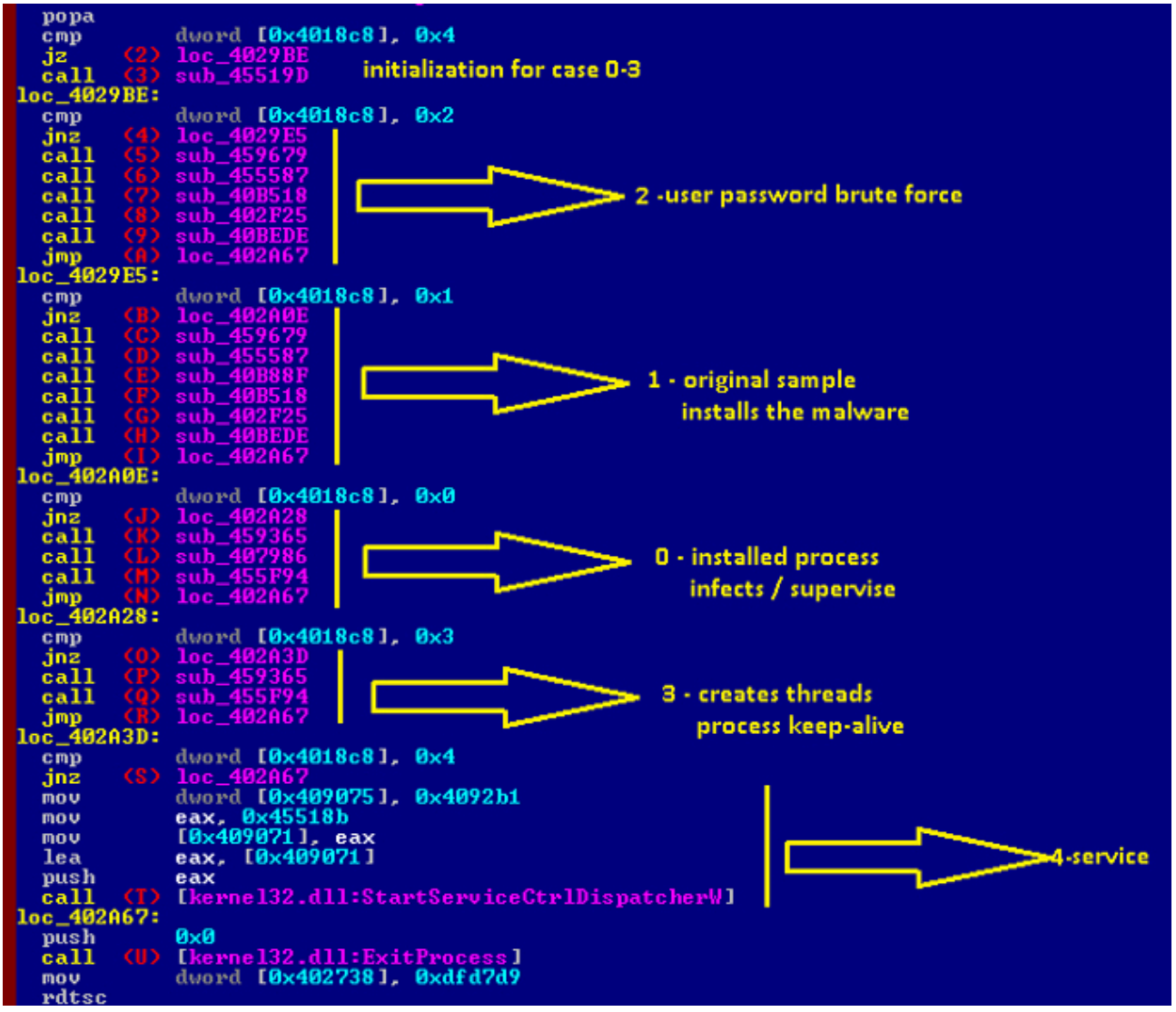

Figure 12: Last context switching actions.

2.1.2.3 Anti-emulation

Most malware creators integrate into their applications techniques to escape emulation and/or virtual machines. There are a number of known methods to accomplish that, we won't discuss all of them, but mainly those used by Virlock.

Among all the techniques which can cause emulators not to work, there are time constraints and unimplemented emulated API calls. Some emulators which are at the beginning, might have problems overcoming both of these, others might give up over time constraints (mainly because authors consider this a performance hit), and other advanced emulators could solve all of these in more efficient ways. However, most emulators are somewhere in the middle most of the time. We have to consider the possibility that from time to time malware creators reverse our engines and create malware which might target some of these security engines. If that is the case, then no matter how strongly an emulator is built, it might become useless if it's being targeted by malware.

Randomly chosen API calls

In an attempt to morph itself, Virlock rebuilds itself inside each infection, decorating the core of functionalities with things like random API calls from randomly chosen modules. The malware uses some tables, meaning that it does not choose from a huge set of possibilities but from a finite set. It chooses a random number of libraries which the future infection will import, and from those libraries, some random APIs inside each of them are chosen as imports.

If emulators are only emulating a certain set of APIs, then that might impede their ability to continue at the point of an unknown API call, or an API call not implemented accordingly (Figure 13).

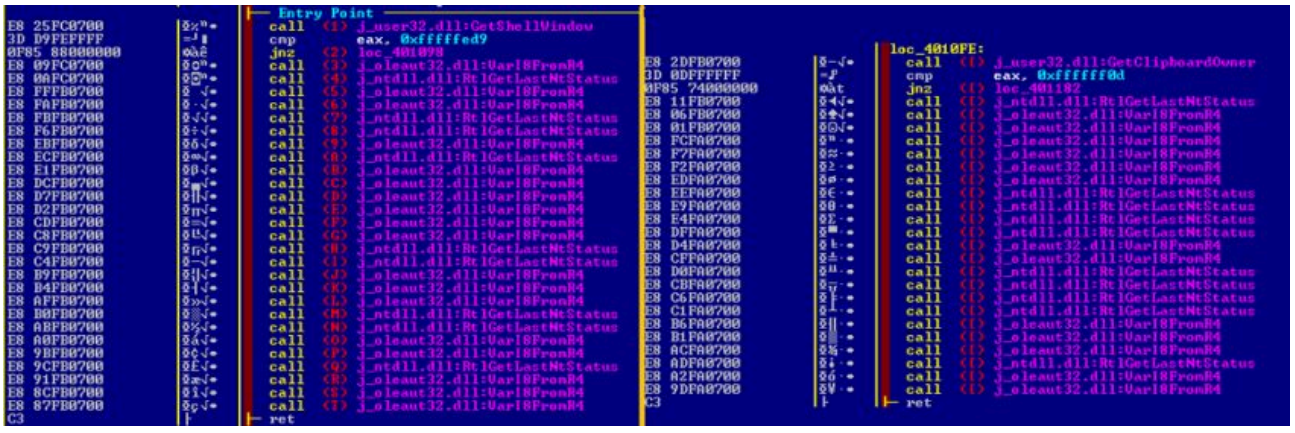

Figure 13: Consecutive blocks of random API calls, trying to escape emulators from the beginning.

Increasing the number of executed instructions

Most malware, be it packed or unpacked, does not require more than a few million instructions to be executed. At that point there are optimizations such as binary translation, which tries to improve performance over emulated loops like decryption blocks which get to be executed by the real processor and not by the emulator. Binary translation is sometimes combined with file-read operations – the best emulators will try to reduce the number of read operations and at the same time the maximum number of instructions allowed to be executed.

All versions of Virlock have a first stage decryption. Without it, any further code execution is basically impossible. There is currently no version that executes fewer than 60M instructions for that purpose, and the

number of instructions increases for bigger files and larger obfuscated loops, to hundreds of millions of instructions. Some infections also spread the obfuscated loops over a large area of the infected file, thus passing to emulators the pain of consecutive file reads, which also is a hit for performance.
There are many cases where the binary translation for loops is almost impossible if we are not first going to de-obfuscate the code being executed by the loop. Figure 14 shows such a case where just three calls to load more than 180 APIs from different modules is taking at least 500k instructions.

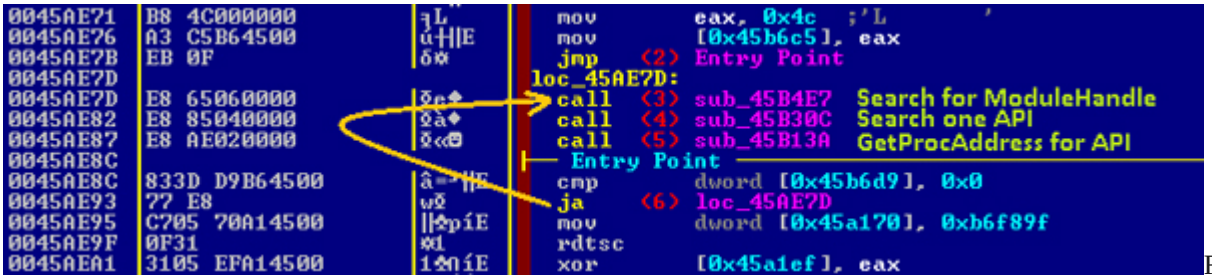

Figure 14: Loading some APIs (calling is based on templates discussed in 2.1.2.2).

2.1.3 Cheating users

Very rarely seen in other pieces of malware of this kind (which embed the clean file into a totally different file), Virlock tries to cheat users into thinking that an infected file is actually what its icon claims it to be. There is a stage in the infection process where the malware searches inside the registry for the application associated with an extension type, in order to get to the file containing the icon of the associated application. This is a primary step for grabbing the icon and embedding it into the final infected file as an icon-resource. At a first glance, there is no difference between the original file and the infected one.

Straight after the infection, the malware will set a registry setting to hide extensions for known filenames. That way users will see their original files with their relevant icons and no EXE extension, so no one will ever doubt the actions of the file.

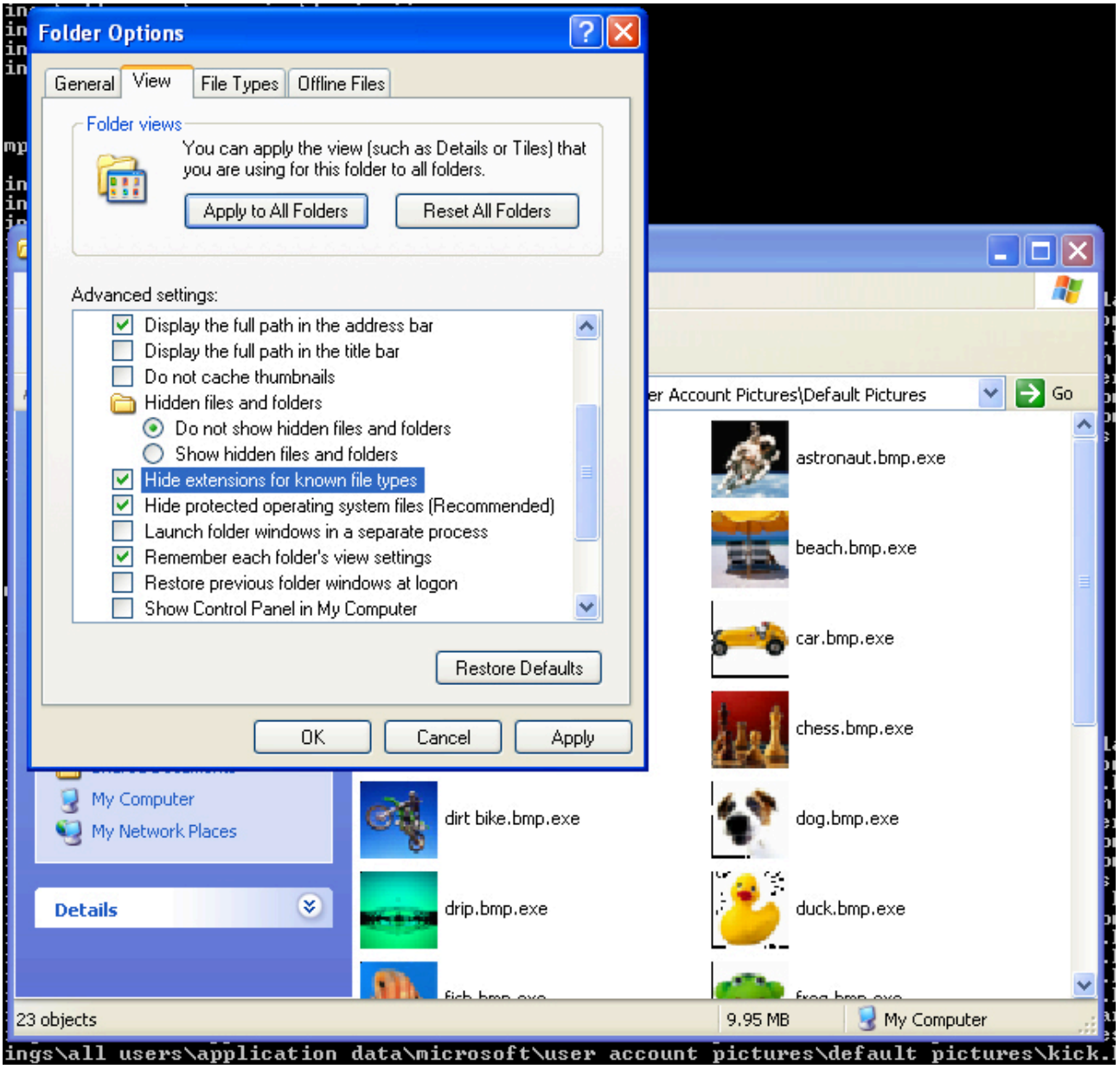

Figure 15: Infected files with extensions revealed.

2.1.4 Polymorphic engine

The thing that makes Virlock so special is that it has a polymorphic engine which mutates its shape in future infections. In this section we reveal the techniques used by the malware to accomplish this task.

Straight after the API-loading process, the malware allocates two buffers (one of them big enough to hold the core of the malware) to prepare the morphing process for the infections to come. The core of the malware is somewhere inside the infected application, but only visible after a few stages of successive decryption procedures. Figure 16 shows the schematics of the core, which resides packed, layered inside any infected file.

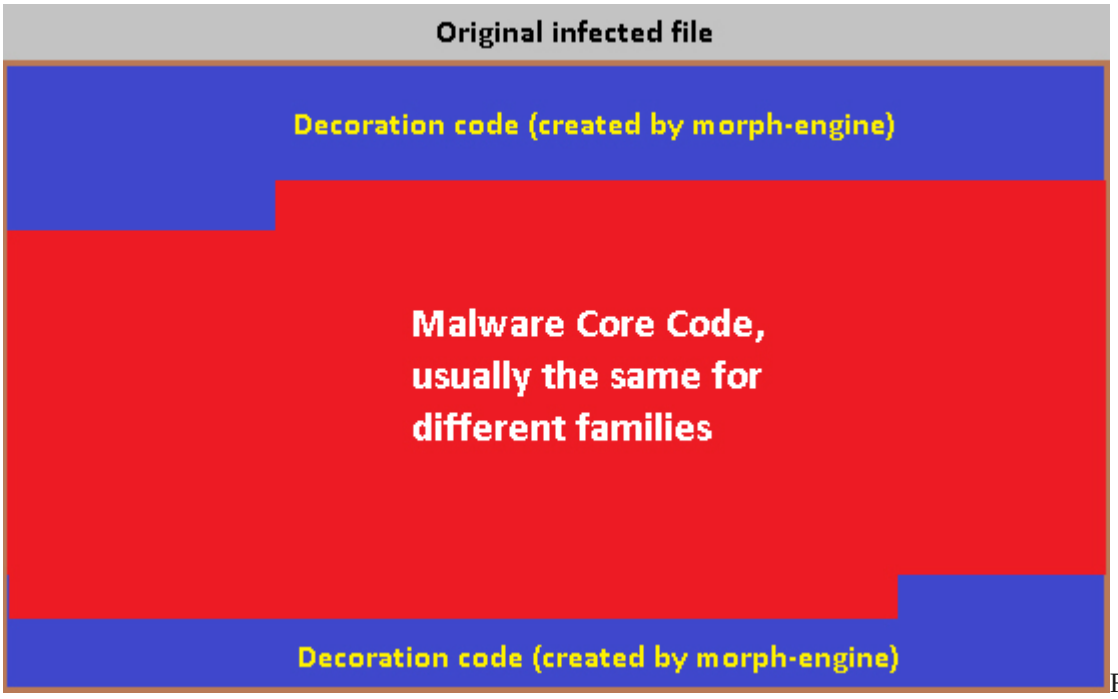

Figure 16: Virlock core with embedded clean application.

A polymorphic engine is located in our example at 0x45E636 and it is called several times during the installation of the malware into the newly infected system. Each new malware copy will also have modified the flags discussed previously, accordingly.

```
push    0x80
push    dword [0x459818]
call    (3) [sub_45B751]        SetFileAttributes(<dropped file>,HIDDEN)
push    0x0
push    0x0
push    0x0
push    0x0
push    0x0
push    dword [0x459818]
push    0x0
call    (4) sub_45E636          MorphicEngine
push    0x10
lea     eax, [0x459311]
push    eax
call    (5) [sub_45B711]
push    0x44  ;'D               '
lea     eax, [0x459321]
push    eax
call    (6) [sub_45B711]
push    0x7
push    dword [0x459818]
call    (7) [sub_45B751]        SetFileAttributes(<dropped file>,HIDDEN)
push    0x459311
push    0x459321
push    0x0
push    0x0
push    0x0
push    0x0
push    0x0
push    0x0
push    0x0
push    dword [0x459818]
call    (8) [sub_45B761]        CreateProcess(<installed process>)
mov     eax, 0x1
ret
```

Figure 17: Code calling the polymorphic engine.

The process of shape-changing is accomplished in two steps, for each of the two dropped files which are going to do the real infection. Figure 18 shows the preparation for the reshaping of a self-copy.

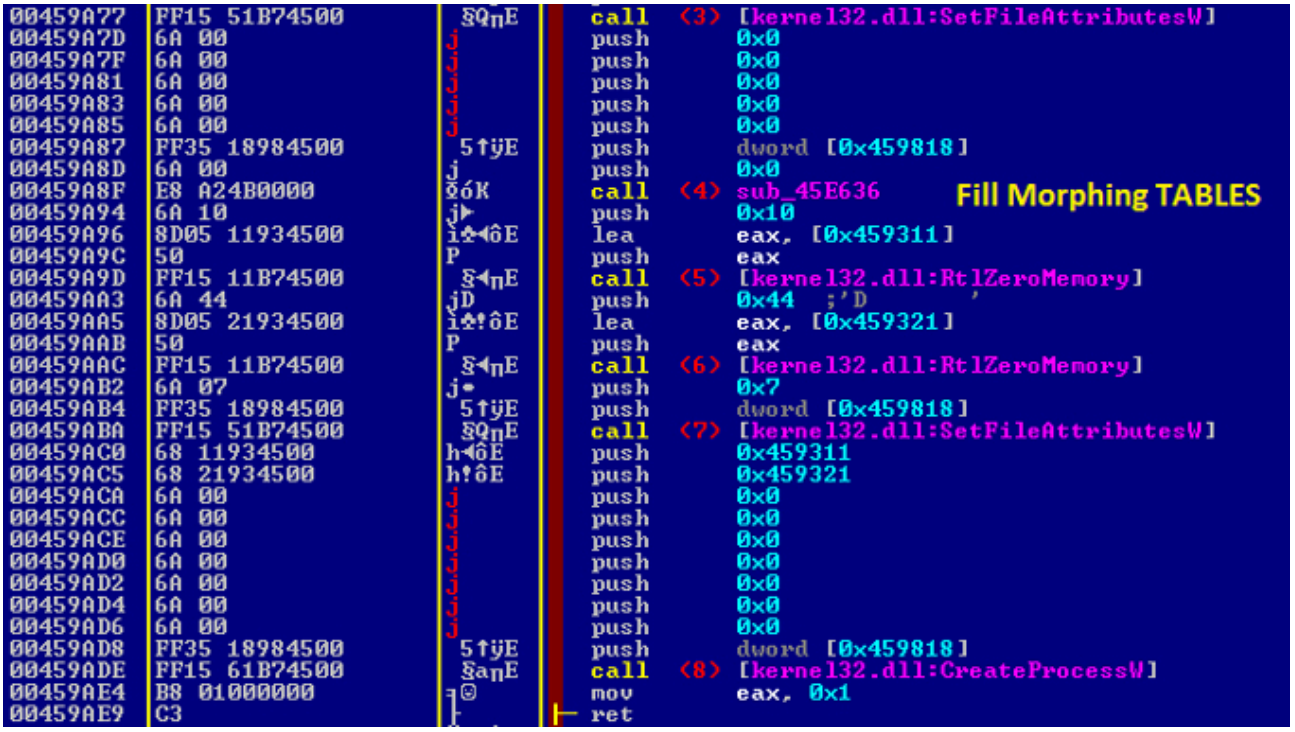

Figure 18: Preparing the reshape of a self-copy.

The first stage consists of preparing random file names, some random seeds, and the buffers involved in the morphing procedure (see Table 2).

Buffer alias	Buffer size	Buffer ptr	Description
TAB1	0x200	0x970000	Randomization table 1
TAB2	0x2300000	0x1100000	Working buffer for reshaping procedure
TAB3	0x10000	0x9A0000	Intermediate table 1
TAB4	0x10000	0xAA0000	Intermediate table 2
TAB5	0x200	0x980000	Randomization table 2

Table 2: Buffers involved in the morphing procedure.

We also see at this step the creation of two different MZPE file headers, originally packed inside the malware (see Figure 19). Their purpose is to fulfil the creation of the processes which will actually carry out the infection.

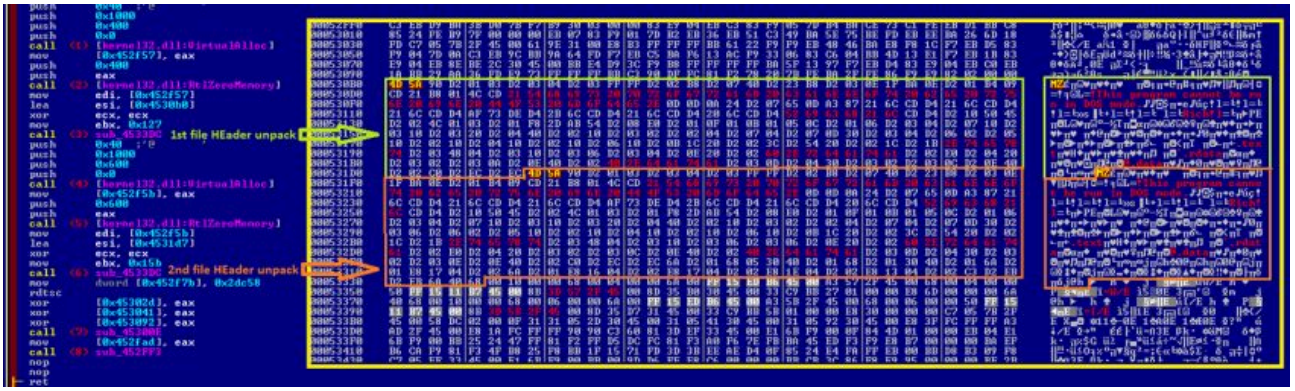

Figure 19: Preparing headers for the files to be constructed.

In the beginning of the second stage, the malware creates a custom import table, also based on time seeds (see Figure 20). The RDTSC instruction, which provides those time-seeds, is called very frequently, not only to randomize stuff, but also for choosing random locations in the target application, where relevant data regarding decryption keys, buffer pointers, etc., will be placed.

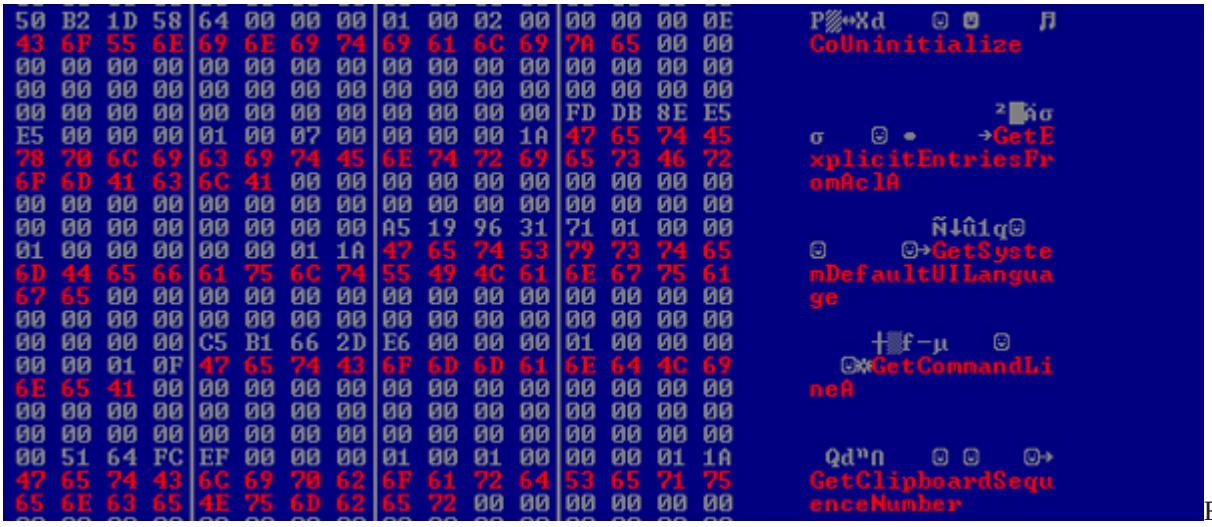

Figure 20: Building a customized import table.

In Figure 21, we can see a sequence of instructions which progressively builds the decoration of the new infection.

Figure 21: Reshaping a new infection.

All the steps required for a full file creations are called in a sequence of three consecutive calls, as shown in Figure 22 {reshape / append / recrypt}.

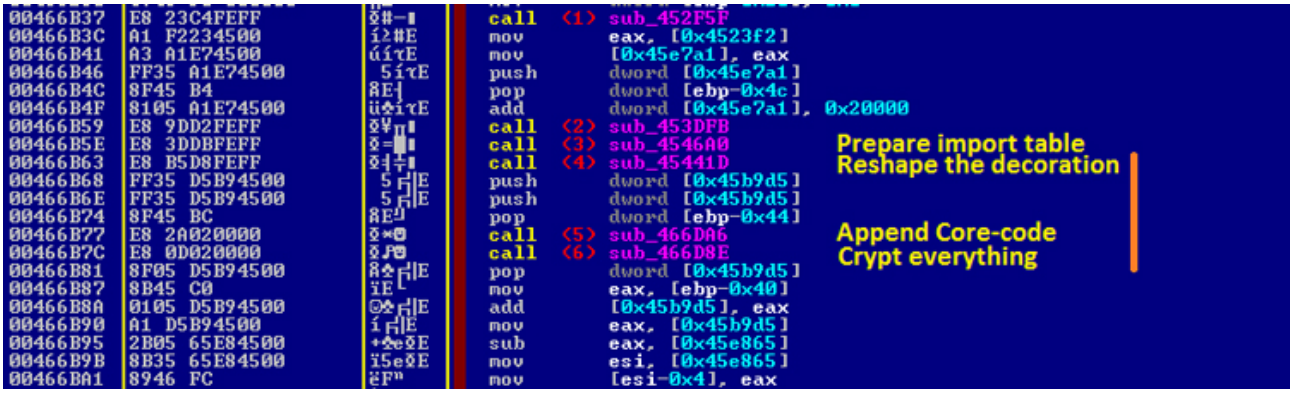

Figure 22: Main reshape steps for self-copies.

2.2 Analysing Virlock – Combining purpose

We have seen lots of malware categories that combine their powers with other malware categories. The results of those combinations have, most of the time, been some kind of surprise for security products. Not only do malware authors learn from security products how to improve their performance, but we also learn from malware authors that there is always something which we have not taken into account in the first place. This sounds like an evolving loop, where security products try to nullify malware actions, while on the other hand malware authors try to nullify security products' actions. Well, at least the loop is more like a three-dimensional spiral, otherwise we would not exist at this moment in time.

The following is a brief history of combined malware actions including Virlock, which we find as a reference for this case:

- Viking / Jadtre – rootkit and file infector
- CBDoorK – rootkit and backdoor
- Sality – file infector, botnet, worm
- Virlock – ransomware, file infector.

2.2.1 File infector and screen-locker

Until Virlock, no other malware combined these features. Malware authors who write ransomware are doing it for the money – they say as much in their readme files appearing on the infected computers. For example, a piece of ransomware using the Bitlocker feature from *Windows* tells the infected users that 'This is just how business works, pay and you'll get your data back.'

Early versions of ransomware only locked users' accounts, hoping that some of them would fall into their trap – and they succeeded, but there is always room for improvement. Some of the next versions tried to encrypt users' files with symmetric keys and locked the users' accounts, making it more difficult to revert the process. But as the security products improved their strategies and delivered rescue-CDs to users, malware authors improved their methods of cryptography, using asymmetric algorithms, and gave up the screen-locking. When infecting users with those kinds of ransomware, malware creators need a management system in order to bind private-keys with malware versions. Maybe they did not expect their methods to be so fruitful, but they seem to be overwhelmed by the number of infected users and public/private keys. It is not unusual for a user to try to pay, and get a decryptor which attempts to decrypt files from a different infection.

Virlock tries somehow to escape the load produced by the key-infection management system while improving the old techniques used in locking files and user accounts by embedding the clean file and packing it safe inside the malware with random and hard-coded keys. It also tries to crack users' account passwords, to lock their account in order to make it as difficult as possible for the users to recover their files. Using the presented technique for file infection, security products have to consider an entire arsenal of variables in order to begin a clean method, because it would be very easy to miss a certain hard-coded-key and to damage the file instead of recovering it.

3. Getting to the Core of Virlock

We've seen so far that Virlock uses a template-based reshape, so we can use that template as some kind of regular expression to find some inner pylons / code-blocks to start with. Studying the five different versions until now, there are certain similarities between them, which will lead us to classify a sample as infected.

In this chapter we will try to reveal the malware's weak points and see how those weaknesses may contribute to studying it better in all its present forms.

3.1 Revealing the core, inside different malware versions

First, there is an initial layer of decryption which will end up by continuing the execution somewhere at FirstSectionVA+0x400 or FirstSEctionVA+0x1000 with or without additional obfuscated code and possibly a

short second decryption stage (Figure 23).

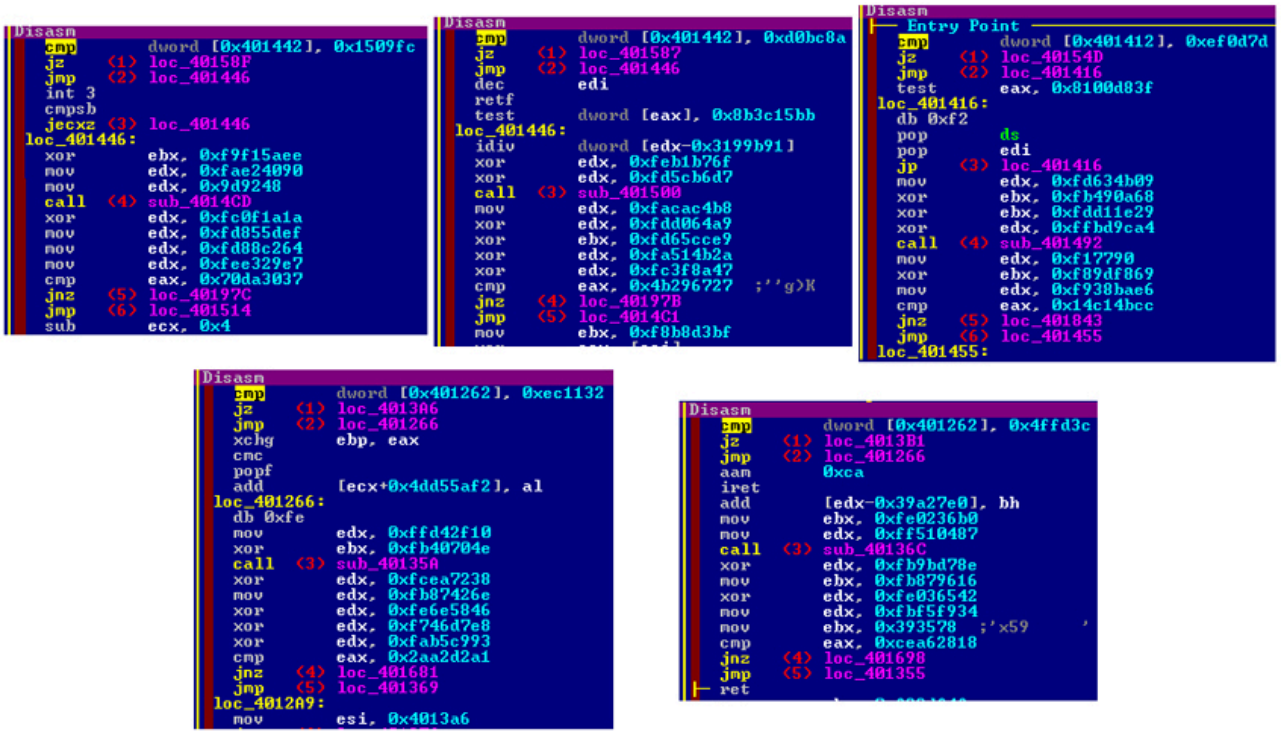

Figure 23: First chunk of relevant code in all five versions.

There are two major switch sections inside the malware which choose a path of execution depending on the hard-coded flag discussed in section 2.1.2.2. We will consider the two sections as the core of the malware, as they are present inside all versions, no matter how obfuscated the code is, and the path to those functionalities is unique if an emulator behaves just like a real operating system.

Not all versions are as compact, as shown in Figure 24. There are some cases where junk-code might appear between relevant instructions in our target code, but ignoring them is not as difficult as one may think.

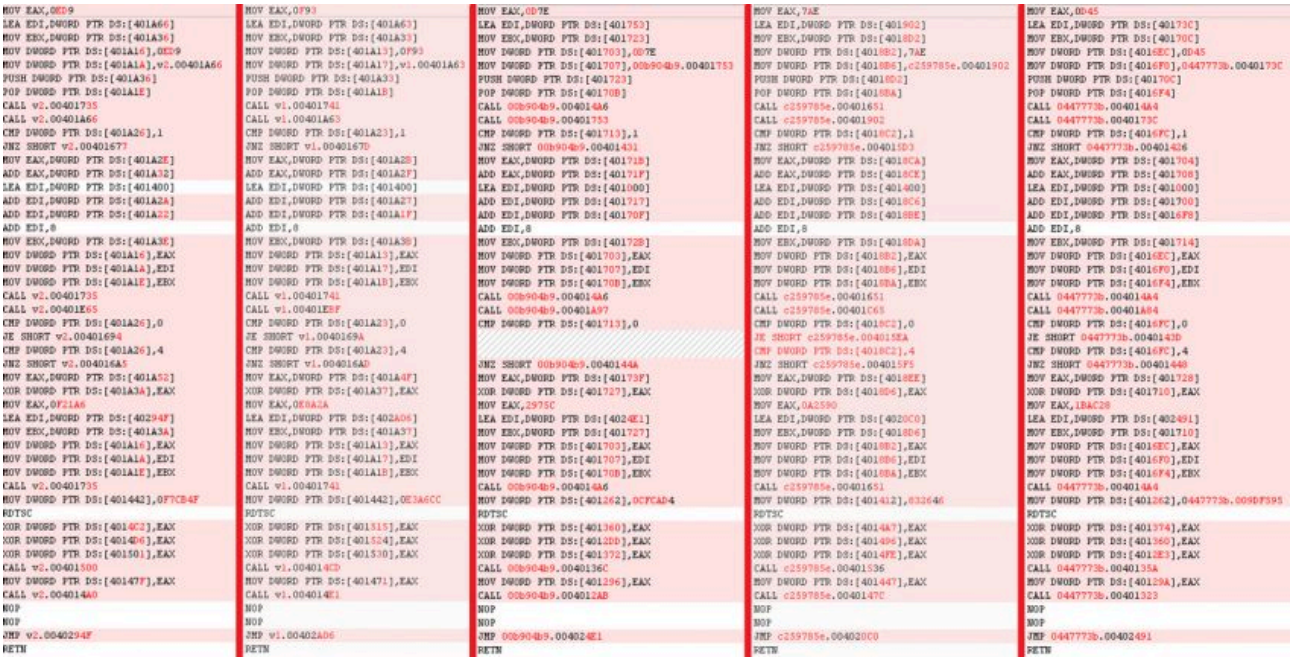

Figure 24: A comparison between all five versions inside context-switch sections.

3.2 Searching for a match

Most detection algorithms will just try to find a relevant piece of code inside a piece of malware. Looking at the code shown in Figure 25, we might be tempted to say that we found something relevant for our malware (a branching point where it chooses to execute as installed or as a fresh infection). However, in other malware versions we found other such pieces of code, doing the same thing but with modified instructions. Considering this, the detection cannot choose that sequence of instructions to follow, but we need some rules depending mostly on the constant addresses given in the piece of code and the instruction types, which are not so different across different malware versions. This kind of matching seems to be as powerful as a regular expression-matching algorithm, but additional changes have to be considered.

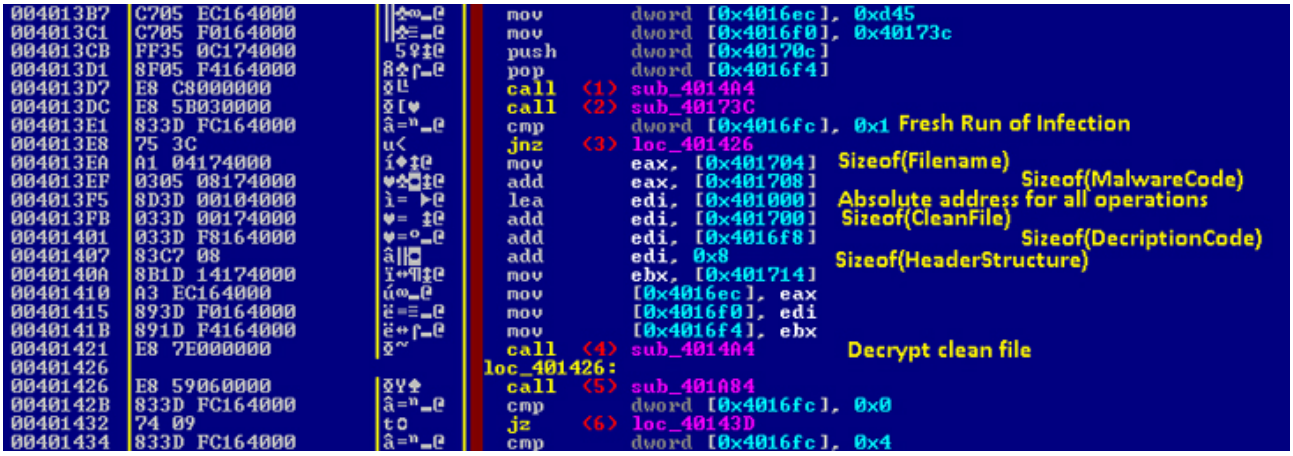

Figure 25: Piece of malware code to decrypt clean file.

3.3 Cleaning infected files

To recover the clean file from the malware, we need to follow the code until a point at which we can check whether the infection contains a clean file (switch-flag == 1) or not (switch-flag != 1). If we do have a clean file,

we need to grab the hard-coded values inside the malware (different with each infected file) and to force the emulation of decryption functions.

A simple clean procedure is to use the emulation to execute the decryption function. After that, we can grab from memory, using the specified variables, the actual clean file. The starting point of a particular clean file inside the malware is shown in Figure 26.

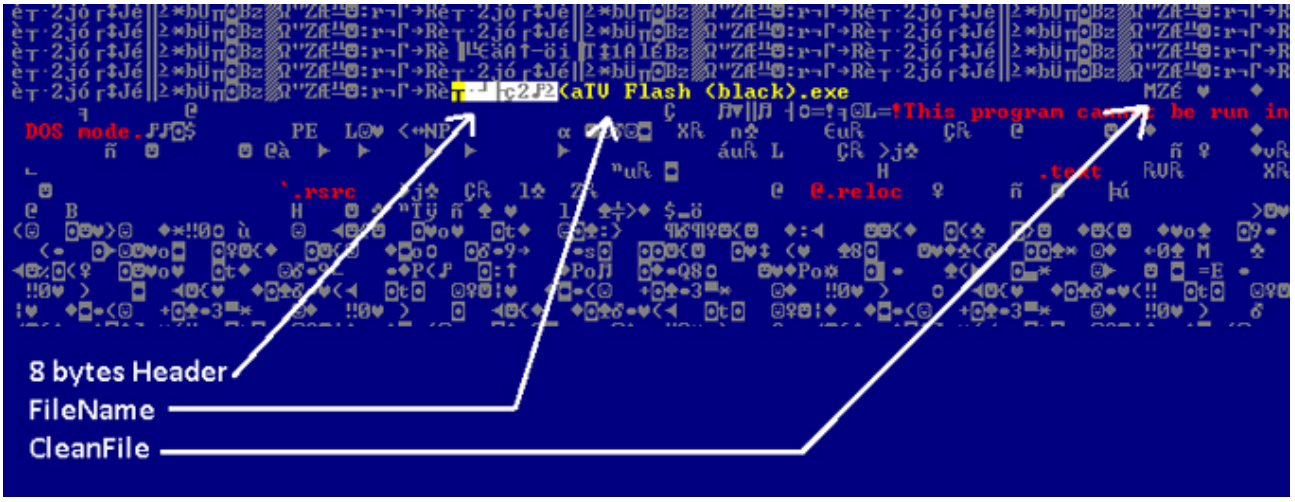

Figure 26: Finding clean file using hard-coded variables.

4. Statistics

Figure 27 shows a graphic for the timeline of Win32.Virlock.Gen.1, which is the most widespread version at the moment.

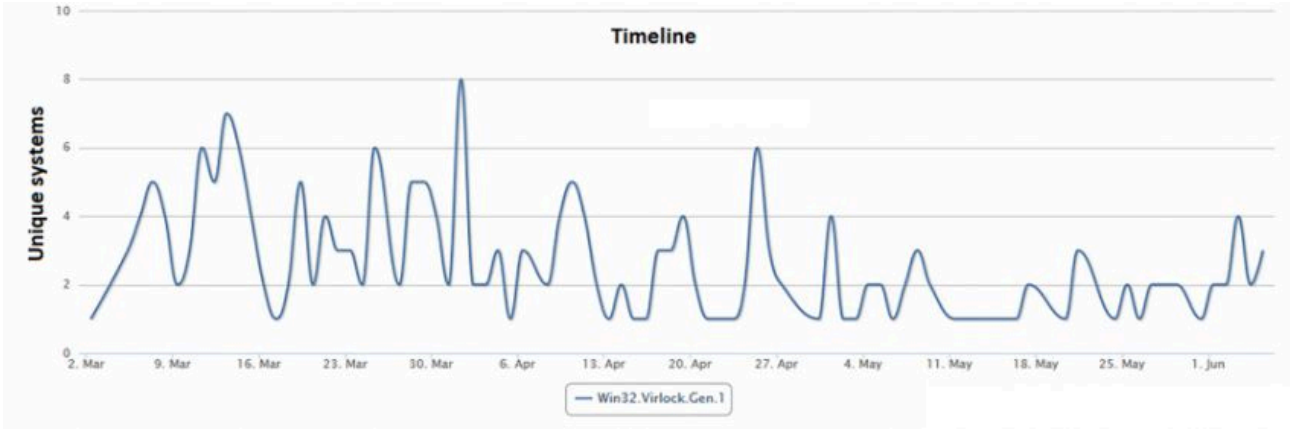

Figure 27: Infection timeline for Win32.Virlock.Gen.1.

In Figure 28, we see how many systems have been infected since March 2015 for the three most common detections. Almost 39,700 unique files were detected by *Bitdefender* on 148 systems in less than five months. The highest number of infections were detected in Canada – almost 30,000, representing 75% of all infections. We expect a small increase in the next few months as the authors of the malware seem to still be working on it, and a total decrease by the middle of next year, by which time many security products will have solutions for it.

Country name	Inf. systems	Inf. files
China	22	192
United States	11	663
Germany	10	106
Russian Federation	9	82
Canada	8	10101
Australia	7	3708
France	5	172
Switzerland	5	909
Romania	5	219
United Kingdom	4	544
Iran, Islamic Republic of	3	77
Sweden	3	42
India	2	15
Bosnia and Herzegovina	2	47
Ukraine	2	9
Netherlands	2	1178
Poland	2	4
Indonesia	2	74
Vietnam	2	1388

Country name	Inf. systems	Inf. files
Canada	3	19124
Australia	2	6
United States	2	281
Indonesia	1	4
Russian Federation	1	1
Vietnam	1	115
Namibia	1	23
Switzerland	1	29

Country name	Inf. systems	Inf. files
France	2	205
Canada	1	1
Indonesia	1	13
United States	1	6
Russian Federation	1	2
Vietnam	1	33
Namibia	1	2
Switzerland	1	5

Figure 28: Left: Win32.Virlock.Gen.1, Top-right: Win32.Virlock.Gen.3, Bottom-right: Win32.Virlock.Gen.4.

5. Conclusions

It seems that malware creators are constantly learning from their mistakes and they always find new ways to bypass security products, be it with a small improvement such that their sample will not be detected for a few days, combining technologies that could force certain security products to redesign their engines (due to performance-hits) in order to come up with a feature to successfully detect and clean the malicious application, or forcing security companies to search for better solutions or to give-up by not being able to keep up with damages done by specific malware infections.

Virlock is among the few malware applications which combines different technologies to harden the reverse engineering process and at the same time to make the creators of security products question their technologies. The redesign process of certain engines is not always an easy step, and most of the time this is not a solution. For example, to add some features to emulators, in order to execute unimplemented APIs, to track a certain sequence of generic assembly instructions, or to increase the complexity of search algorithms near to the complexity of strstr(), might result in performance hits which will impact the overall functionalities of the security product. Some designers being inspired in the first place might laugh at the idea that an improvement could be made as a next step inside an already evolved tool, but that is not always the case.

With the advance of malware technologies in the last few years, we find it even harder to revert malware, or to revert the infection process and to restore the system to a clean state. Ransomware using asymmetric encryption algorithms is constantly destroying user-data requiring money to get data back. More than ever, we need methods to automate dynamic analysis and at the same time to extract relevant features from different infections along with improving the prevention techniques. Model-checking and symbolic simulation may be a solution from that point of view, and maybe combining that with time-line analysis and control of a running operating system environment, we might prevent, learn and successfully revert much more complex infections.

There is also a small chance that by using classifiers to extract common vector-features from traces obtained from emulation of such malware, and then dynamically observing the modifications which take place during the infection, one could generate the detection process (which resumes to a search problem in the space of files to be scanned), along with the disinfection process, in just one click.

Acknowledgements

This work was co-funded by the European Social Fund through Sectoral Operational Programme Human Resources Development 2007 – 2013, project number POSDRU/187/1.5/S/155397, project title ‘Towards a New Generation of Elite Researchers through Doctoral Scolarships.’
