Thesis no:
URN:urn:nbn:se:bth-14584
DiVA: diva2-draft:56543

An experimental study on which anti-reverse engineering technique are the most effective to protect your software from reversers

Jozef Miljak

Faculty of Computing
Blekinge Institute of Technology
Se-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor in Software Engineering.

Contact Information:
Author(s):
Jozef Miljak
E-mail: jozefmiljak2@gmail.com

External advisor:
Non appliciable

University advisor:
Conny Johansson
Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Contents

Disclaimers

All the software/programs used within this document have been used only for the purpose of demonstrating the theories and techniques described. No distribution of patched/modified files (software/programs/applications) has been done under any media or host. The file used has already been patched/cracked and uploaded elsewhere by others a long time ago. The author (Jozef Miljak) of this paper cannot be considered responsible for the damages the companies holding rights on those files. The purpose of this paper is educational, sharing knowledge around anti-reversing techniques used and how to bypass these protections. Incase software that has been developed by you or your organization has been studied in this paper and wish it to be removed, contact the author. The author of this paper is not affiliated with any group(s)/person(s) mentioned. Under no circumstances will any patched/cracked/modified files be distributed by the author.

Abstract

With software that handles private data such as; passwords, health and other personal information that should not be shared publicly, it is important to protect the software as good as possible. A way of protecting your software is with legal rights, but as some might disregard that, it is not considered to be enough. In this paper, techniques that can protect your software will be experimented on with the help of a debugger and other tools. The techniques are discussed and compared by its effectiveness and difficulty. Among all of the techniques experimented on, API Redirection was shown to be most effective one. API redirection was also shown to be flexible such a fashion where it could be tailored towards targeted software.

Keywords; anti-reversing techniques, reverse engineering, disassembly

1. Introduction

Many software/applications (digital products) that we use today require some sort of authentication before you gain access to its features, take the most popular ones Spotify, Microsoft Office products, anti-virus software, the list goes on.
All of these software share the same ideology which is "limit the functionality until authentication is made". Examples of such authentication methods are:

- Validate serial number
- Online activation
    - Checks whether the user is allowed or not
    - Often performs the check each time it connects to the internet
- Account validation
    - Validates whether the account has a subscription or not

There are many more methods out there but for simplicity only the basic methods are listed.
These methods are there to ensure that the user only gains proper access after the authentication.

But what if one could learn how the software validates the license? What if someone actually learned how to generate a valid serial number or better yet: manage to bypass the whole authentication process. That is what a reverse engineer does, they analyze software with the help of debuggers and disassemblers and all of their favorite tools to get to know how your software actually works. Most developers which develop software with some kind of licensing do not want their code to be analyzed outside their organization. So what can the developer do in order to make it harder for the reverse engineer to read their code?

That is the question this thesis will focus on, but more specifically, which anti-reversing techniques could be used and how much they slow the reverse engineer down, in other words, how secure they are. Why this is important has already indirectly been said but to make it even clearer, developers don't want their code analyzed outside their organization and possibly their company secrets revealed, unless they open source their software, in that case it would be free for anyone to analyze the code. Another reason is that this could lead to a larger user base because of piracy not being an option, which is reason enough to protect your software as well as possible.

Experiments on these kinds of techniques will be made personally by me. As one can imagine there are many techniques to make it hard for the reverse engineer which also unfortunately means that not all can be covered in this thesis, but the most interesting in my opinion and the ones which are still mentioned when talking about anti-reversing techniques nowadays will be picked. Please note that "make it harder for the reverse engineer" is used above, this is because, just as Oleg writes in his article "I want to mention right at the beginning that it is impossible to completely protect from reversing. "[1] This is because an executable is always prone to reverse engineering due to the fact that the machine can run it; hence it must also be possible for the human to read it.

The experiments will deliver important data that will support this study by collecting data such as:

- Time consumed to get past the protection
- The amount of technique-specific research needed
- Difficulty level and the effectivity of the technique

Oleg Kulchytskyy, a Software Architect, has written an article called “Anti Reverse Engineering Protection Techniques to Use Before Releasing Software” as both the title and the article suggests, protecting your software with something more than just stating that the software may not be reverse engineered in the EULA (End user license Agreement)[1] should be done.

The experimental test results on each anti-reversing technique category will provide important information for developers who work with software that require high security such as:
is this techniques worth implementing and if so, does it have a high cost in terms of time? Do these techniques hold any advantage over other specific cases? Some information about the costs and benefits compared to the other techniques will also be mentioned.
A comparison and discussion will be made over the techniques studied to get a summarized view over the pros and cons of each technique.

2. Terms

Many terms will be used and in order to make it simple, most of them will be explained here.

[Packer] A packer, sometimes even called a compressor, is the process of compressing an executable file and prepending a decompression stub which is responsible for decompressing the executable and initializing its execution. The pros with compressing executable is that they require less storage on the file system [2], a good example of when packing was used for this is during the DOS time when storage was expensive, less time to transfer data from the file system into memory but at the price of more time to decompress the data before the actual execution begins. Another big con is that antivirus software often flags packed software as potentially dangerous executable [3].
[Protector] A protector emphasizes more on protecting the executable against reverse engineering [5] compared to a packer whose goal is to reduce the size. Because a protector adds code to the executable, it adds size to the executable.
[API] stands for Application Programming Interface. An API is just like a function that can be called upon when for example; you want to access the file system. An API does not have to be defined in your software; calling an API from an external source (a DLL for example) is fine.
[DLL] is a library that contains code and data that can be used by more than one program at the time. It is used in Microsoft's products, for example in their operating systems; the Comdlg32 DLL performs common dialog box related functions [4]. DLL is short for Dynamic-Link library.
[VA/RVA] RVA is short for Relative Virtual Address. Inside an image (exe or DLL) file, an RVA is always the address of an item once it has been loaded into memory with the base address of the image file subtracted from it. VA is the same as RVA except that the base address of the image file is not subtracted [24].
[UnpackMe] is software that is packed and then released for reverse engineers to try and unpack it.
[OEP] is short for Original Entry Point. As a result of packing software, the EP (entry point) will most likely change when protected with anti-reversing techniques hence OEP is used to refer to the EP that was originally set, before the added protection.
[IAT] Every win32 executable has an import address table, referred to as IAT, inside the program [5]. The IAT is used as a lookup table when the application is calling a windows API function, this means that the windows loader has to find each address of each API that the program wants to call and construct an IAT containing these APIs.
[ESP] is an x86 register which is short for stack pointer.
[Code cave] is the code that is written inside the executable memory where there is free memory. An example of such is a script's memory allows for 5 bytes but only 2 bytes are used, we can use these 3 bytes to inject our custom instructions.
[Opcode] is the same as operation code, a portion of machine language instruction that specifies the operation to be performed.
[Bit/Byte/Word] - The fundamental unit of computer storage is a bit. Eight (8) bits equals one (1) byte. Four (4) bytes equals one (1) word [6].
[Reverse engineering] is the process of taking a system, or parts of a system and analyzes their functionality and design [7]. For software engineers, we are familiar with analyzing and debugging, but reverse engineering takes it one step further. One could debug software without the need of the source code, this way one could learn how your software works and integrate it with other software. Malware analysts rely on reverse engineering because malicious software is usually not open-sourced, unless it is uploaded for educational purposes.

3. Literature study

3.1. X86 Assembly

Assembly language is a low-level programming language [8], the closest form of communication that one can have with a computer [9], the programmer can as a result track the flow of the data end execution in a program precisely in a mostly human-readable form [9]. Because we will use a disassembler (OllyDbg is a debugger which disassembles) and disassemblers translates the machine language into assembly language [10], we need to understand assembly language in order to understand the program. Because knowing assembly language is fundamental to perform experiments like these the x86 Assembly wikibook will be used to gain required knowledge and incase of stumbling upon instructions which I do not understand. X86 Assembly contains information such as Instruction Sets, Syntaxes and Assemblers, Instruction Extensions, more advanced x86 and more, with sub categories making it easy to find what you are looking for.

3.2. X86 Disassembly

X86 Disassembly is an open source book that goes in-depth in many specific parts such as things to know about different platforms for example Windows, GNU/Linux and OSX, different tools, code patterns, data patterns, and difficulties. Disassembly is the process of translating an already compiled program into assembly [10], language; it is good to have a basic level of understanding how assemblies, computers and operating systems work.

In the x86 Disassembly book you can find information about different compilers, different assembly languages and comments on different tools for disassembling and analyzing as well as techniques used to make reverse-engineering harder.

The reliability of an open source book is not the best because anyone can modify it with an approval but thinking twice about this, anyone can also correct the errors. If parts of the book are found doubtful or unclear, other sources will be used to confirm said doubts or unclarities.

3.3. The Art of Unpacking by Mark Vincent Yason

The art of Unpacking is a paper that has great and valuable information regarding many of the anti-reversing techniques that has been used on software. The techniques are mostly explained briefly, how they are used and what can be done in order to bypass the anti-reversing technique applied. Even though Mark's paper has solutions to each technique, it is not described in a way so that one can apply it and bypass the protection for any software, because every software differs and so does the technique. An example of what I mean would be to look at two different packers, both adding protection with the use of the same technique but in a different way.

Mark's paper is written towards an audience that already has some knowledge around reverse engineering and therefore may not be suitable to start reading until knowledge around assembly and different terms used in reverse engineering is acquired.

3.4. Reversing: Secrets of Reverse Engineering by Eldad Eliam

This book contains good information regarding many spots that a reverse engineer will touch for example when and why reverse engineering is useful, what can be achieved and how to protect your software against reversers. From the back cover:

"Sometimes, the best way to advance is in reverse
If you want to know how something works, you take it apart very carefully. That's exactly what this book shows you—how to deconstruct software in a way that reveals design and implementation details, sometimes even source code. Why? Because reversing reveals weak spots, so you can target your security efforts. Because you can reverse-engineer malicious code in order to neutralize it. Because understanding what makes a program work lets you build a better one..."

The quote above is not only informational but also further strengthens the purpose of this paper, techniques that slow down the reverser are very important for businesses that require high security for their software.

3.5. Tuts 4 you

Tuts 4 You is a website with a forum where people can share their findings and also discuss on their forums. Tuts 4 You also hosts a wide range of plugins, eBooks, tutorials and tools which can be downloaded for free. A brief description that I quote from their website
"Setup in 2003 Tuts 4 You is a non-commercial, independent community dedicated to the sharing of knowledge and information on reverse code engineering in many of the subject areas it spans, across the many different operating systems, platforms, hardware and devices that exist today.

Tuts 4 You takes pride in knowing that it has been able, for over a decade, in uniting talented people from all corners of the reverse engineering community and enabling them to develop their skills, share ideas, promote and kick start projects that have helped to shape the reverse engineering community. It's with this knowledge that Tuts 4 You continually strives to be used as a conduit to push the reverse engineering community to new heights and new goals..."
Problems that are stumbled upon during research will be discussed with the members of the forums, many of which are very skilled.

3.6. Lenas Reversing Series

Lena's reversing is a series with a collection of tutorials containing 40 episodes. These series aim to take a beginner in reverse engineering. Just to give a hint, the title of the first episode is "Olly + assembler + patching a basic reverse me" and episode 39 is titled "Inlining a blowfish scheme in a packed & CRC protected dll + unpacking Asprotect SKE 2.2" [25]. Indeed, the title of episode 39 suggests that there is some advanced reversing going, which is the case.

In each episode of Lena's series there is a shockwave flash video to guide you through the episode, step by step with explanations where needed. The problem is that a "Newbie" will not be able to learn much from just following a video and being spoon fed with the solutions if he/she does not think for themselves and trying to understand why the solutions actually work. To be able to think for yourself and gain proper understanding of why the solution is working, one has to know at least the basics of

assembly language. Lena will often try her best to explain why the solutions are working but if you do not understand it well enough, you will not be able to reverse software by yourself because you will be dependent on a video guiding.

Before following Lena's reversing series I had already had courses in assembly and also operating systems. Knowledge around these areas aided in understanding Lena's explanations better especially when things became more complicated, for example when explaining how the memory works, stack, the registers and so on. It is therefore my recommendation that one should acquire basic knowledge on assembly and operating systems in general before falling for the, in my opinion, somewhat misleading title Lena has given her series.

For this thesis and for my own good I followed Lena's series up to episode 23. In the beginning, when learning, I always watched the videos because a newbie expected to be able to learn so fast that the video can be skipped after only a few episodes.
When I reached episode 10 I decided to skip the video and try to reverse on my own, mostly because being spoon-fed with the solutions is not a good way to learn in my experience, but by trial and error, and patience of course, one can learn much better. After trying on my own, I would watch the video for the solution if I feel like I do not have a clue on how to proceed, but even if I succeed I would watch the video because maybe a different approach on how to solve the problem was used. To cover what was learned from episode 1 until 23 would require another 50 pages but to put it simply I learned how OllyDbg works, how to think, what to look for, how to study and observe a software, that there are many solutions to a problem, more about how image files interact with windows and how windows responds and more. Following Lena's series gave me the fundamental knowledge I needed to do the experiments on my own along with specific research about the anti-reversing technique I was tackling at the moment.

4. Tools and Environment

4.1. OllyDbg v1.10

In my search for a debugger and an analyzer, OllyDbg is warmly recommended by many forums and articles. OllyDbg probably is the best user-mode debugger out there and that OllyDbg actually offers more than enough [11]. A recommendation from someone that has devoted several years to develop advanced reverse engineering techniques is a good recommendation.

OllyDbg is a debugger with many plugins available that has been developed by the community. Modifications of the assembly code and memory editing can be done during run time, which is perfect to test whether your goals are achieved before saving the executable. It is available for free at www.ollydbg.de, the latest version is 2.01 but in this study 1.10 is used because this version still has most support and is still used by many when making reversing-tutorials.

4.1.1. OllyDbg Plugins

There are many useful plugins that makes OllyDbg more powerful than it already is, the plugins that will be used in this study are:

- StrongOD
- PhantOm
- OllyDump
- IDAFicator

StrongOD adds keyboard shortcuts, making your life easier when working with OllyDbg for example, pressing “DELETE” will fill the selected data with NOP, pressing “ESC” in the stack window will sync the stack window with the ESP. Apart from keyboard shortcuts,
StrongOD includes a collection of anti-anti-reversing tricks such as HidePEB, Anti-attach, Break on TLS and a few more.

PhantOm is a plugin that helps conceal OllyDbg, it should not be needed for these experiments but it can be necessary when reversing other software.

OllyDump is a must for these experiments. It allows the reverser to dump the debugged process after it has been modified and also has two more advanced options, find OEP, trace and trace into. This plugin will be used in this study for every experiment where dumping is needed.
IDAFicator is a plugin with a collection of utilities. Adds a new toolbar with features such as go to next/previous line you were on, go to beginning/end of current routine, a hardware breakpoint window and more. This plugin will be used to make the analyzing part easier.
These plugins were recommended either from threads in Tuts 4 You or by tutorials for a specific experiment.

4.1.2. OllyDbg Quick Start Guide

Because OllyDbg is filled with features and sometimes it can be hard to find what you are looking for, a quick start with all the shortcuts OllyDbg has will be to great help.

It can save a lot of time knowing shortcuts to your favorite windows/features and after a while they will feel natural and you will not even have to check the quick start guide [12].

4.2. PEiD v0.95

PEiD is a small handy tool which can be used to analyze which packers, crypters and compilers have been used by simply opening the program with PEiD.
Other functions such as viewing sections, details of the executable and also plugins are available but since PEiD is discontinued, the plugins to find the OEP for example will in most cases not work (if the executable is protected).

4.3. LordPE by yoda

LordPE offers tools for manipulating various parts of PE files. One of the main features is the PE editor, PE rebuilder, unsplitter and dumper server. The PE rebuilder is the feature that will be used the most in this study, because after changing parts of a PE it must be rebuilt to run properly.
The version of LordPE can not be found inside the program but it is advertised as version 1.41.

4.4. Import Reconstructor

Because the import table can be corrupt when making changes to an executable, a tool to fix it is needed. Import Reconstructor is designed to rebuild the imports for win32 executable.
Import Reconstructor (often called ImpRec) will reconstruct a new Image Import Descriptor (IID), Import Array Table (IAT) and all ASCII modules and function names.
It also offers more advanced features such as injecting and auto-searching for the API's.

4.5. Environment

Since many of the tools of my choosing are old, I have chosen to work on a Virtual Machine by using VM VirtualBox with these specs:

- Windows XP with Service Pack 2 installed
- 2048 MB RAM
- 4770K @ 3.50GHz (Host computer)
- GTX 770 (Host computer)
- 20 GB HDD

These specs including the guest image from VM VirtualBox will allow me to work efficiently without any lag or breaking any of the tools recommended specs.
It is worth mentioning that I also have the tools installed on another machine, so some of the screenshots may be taken on a Windows 10 machine, but no experiments were conducted on this machine.

5. Research questions

This thesis will mainly answer three very important questions. They are:

1. Which anti-reverse engineering technique was shown to be most effective?
2. Can the most effective technique found in this study be further customized towards the software?
3. Can the techniques be combined in order to further strengthen the protection?

These questions may not be straightforward at the very first glance so let us elaborate starting with the first question.
We have seen that there are many anti-reversing techniques out there and therefore it is important to analyze them in a manner where comparisons can be made between the different techniques in order to prove their effectiveness. How effective the technique is will primarily lie on the fact of how much harder it makes it for the reverse engineer, but other factors will also play a role. In order to compare different techniques, research must be made to better understand them but also to find an UnpackMe which has this technique built in so that experiments can be performed.

The second question is if the found effective technique can be further customized, or in other words, tailored towards the software. This completely depends on the flexibility of the technique and can first be researched about after the experimental results. If the technique is proven to be flexible and possible to modify in a fashion where as it can be tailored towards the software, it can result in a much more effective technique compared to being a static one.

The third question will be based upon my perceiving after conducting the experiments. After the experiments have been made I should have learned if it is possible to combine different techniques and if not, why? Does some of the techniques conflict with others, resulting in incompatibility to combine them?

6. Method

To achieve successful results, a plan of how to go to is required. After extensive research about reverse engineering software, they all share one important requirement - being analyzed.
To analyze an executable file, various tools are needed for different purposes, for example; a debugger is needed to step the assembly output, but a different tool may be needed to dump the code. Apart from the tools, we also need something to analyze, software in this case and also a method of how to study the software to get good results.

6.1. Picking the anti-reversing techniques

The anti-reversing techniques listed below are the ones that will be in focus and experimented on:

- Self-modifying code
- Packers & Protectors
- Checker for debuggers presence
- API Redirection
- Stolen Bytes

Above are techniques that have been around for quite some time and is recognized by a skilled reverse engineer but also very important to examine because the result provides fundamental knowledge for anti-reverse engineering [13]. Because I do not have previous reverse engineering experience, I could not pick the most advanced and modern techniques, but rather the ones that I believe will not stretch outside of my limits after researching about them.
The techniques listed are merely categories which means that one test on one specific technique may not be enough. Let us take the checking for a debuggers presence technique as an example. It can be achieved in many ways, checking for the most common debuggers names (OllyDbg, IDA, Visual Studio) but it can also be achieved by calling a windows API isDebuggerPresent hence the reason why one test may not be enough.

6.2. Picking the tools

There are many tools to pick between but by filtering them with my requirements I was able to compare less tools hence the picking became easier. The requirements I have are popular, able to run on the environment I have set up and lastly no price tag attached freeware in other words. One may wonder why I would look for popular tool and the reason behind that is if I were to run into problems, a Google search on a popular tool yields more information than an infamous one.

In my search for the necessary tools, I found:

- OllyDbg - Disassembler and Debugger
- LordPE - PE (executable files) editor
- PEiD - detects common packers, crypters and compilers
- Import Reconstructor

These tools were chosen based upon reviews and recommendations found across internet and they all fit the requirements I have - no price tag, popular and runs on my environment. As a bonus most of the tutorials are using the tools above which made the tool picking even an easier task.
Just using a debugger and stepping through code can be very time consuming and hard itself therefore having more tools at hand can save time and make the process more smooth. As an example, PEiD will be used to check whether the software has been packed or protected, in that case, we would have to be more careful when stepping the code so that we can follow the process of the unpacking more smoothly by following the typical behavior of said packer/protector.

6.3. Picking the software

Software with the applied techniques has to be acquired and this will be done through tuts4you.com. The software uploaded is often very simple programs with protection added on top of them. The software with the added anti-reversing technique will be downloaded from tuts4you preferably with a tutorial of some sort, text guide or video.
Software that was developed for the community as a challenge to be reversed was preferred because of legal issues one may encounter if trying to reverse commercial software without permission. Getting permission, let alone information on which anti-reversing techniques used, from a company's product can take quite some time, but most likely the permission request to will be denied. Thus, software developed and uploaded by members of tuts4you is preferred.

6.4. Steps to perform the experiments

Each of the anti-reversing techniques that will be experimented with will share a similar study-method which is as follows:

1. Observing the software - Run the software, explore it, press buttons, see what happens, look for limitations
2. Analyzing the software
    a. Open the software with PEiD - Look if known packers has been used and which compiler was used
    b. Analyze the software in a debugger - See how it behaves inside the debugger by stepping the code. If debugger fails to attach, use plugins/other tools to bypass debugging detection
    c. Study the behavior of the anti-reversing technique applied to the software
3. Patching the software
    a. Find the patch that unprotects the software
    b. Save modified binary code and rebuild if necessary
    c. Run the modified executable to verify that it runs properly
4. Document process and results

When reverse engineering software it is important to observe it, get to know it, what happens if you press this or that [14]. Knowing the software before debugging it can save time because you will most likely already know what you are looking for and where the interesting parts are going to happen, this is why the first step of performing the experiments is observing it. Because I will be dealing with anti-reversing techniques I will use PEiD to my assistance to get more information about the software

beforehand, then I will start debugging to find the location of the interesting part(s), which corresponds to step two. When step two is completed and a vulnerability is found, some modification to the executable has to be made so that the protection is no more. Most modifications will be made by using the debugger, OllyDbg, by modifying the instructions. This is often called "patching", hence step number 3 is named "Patching the software".
These steps will be used in the Practical Results chapter in order to get the data needed such as time, difficulty and how effective it was proven to be. The results and notes along with the process will be documented so that one can re-create the experiments with as much help as possible. The process is mostly documented so that the reader gets a better understanding of why the results are as they are but also so that the experiment can be re-created to verify the results.

6.5. Rating scale

The data will be documented in the section "Difficulty, Effectivity and Conclusion" at the end of each experiment. In this section I will rate the difficulty and effectivity on a scale from 1 to 10 where 1 indicates that the difficulty/effectivity is on a very low level (easy/not worthwhile) and 10 would indicate the opposite, a high level (hard/worthwhile) that is of difficulty/effectivity. The difficulty will be easier to rate because it all depends on my perceiving of how hard it was to find the patch to remove the protection and how hard it was to understand how the technique works.
Because my perceiving most likely differs from someone else I feel that sharing relevant background of me is required. I am a last year student on a 3 years program for a Software Engineering title.
My programming skills are good and I have had courses that are very beneficial to this study, assembly language and operating systems. As mentioned before, I do not have previous experience regarding reverse engineering, so many things is new to me, but I believe myself to be a fast learner on these topics because of my strong technical background.
The effectivity will be harder to rate because two things must be considered, how difficult it is to implement it and the time taken. For example, if a technique is easily applied and causing the reverser to spend much time, the effectivity of said technique is high.
When later comparing them, to my help a graph with the time taken on the x-axis and the difficulty level on the y-axis will be used to gain an overall overview.

Since more than one experiment may be conducted on techniques which belong in different categories, example of such is checking for a debuggers presence and the use of packers/protectors, only the first experiment in each category will contain detailed information such as pictures and deeper explanation if needed.
This is to keep the experiments short with enough information for them to be recreated.

7. Experiments - Practical Results

Each category of techniques mentioned will contain at least one experiment. The amount of experiments per category will be based upon the difficulty, effectivity and time taken. The aim is to have at least two experiments per category so that the results are more accurate, however, if an experiment requires too much, I will be forced to limit myself to only one experiment, because it is most likely going to take a long time on the second experiment hence the effectivity would be close, if not the same, as the previous experiment.
Each technique will also contain a reason for why the technique was chosen and the introduction needed to understand the technique.

Each experiment will be divided into 4 sections. The first one will be observation of the software then comes the analyzing part, patching the software and lastly the data collected during the experiment to determine the difficulty, effectivity and to conclude the experiment.
The layout was chosen based upon the steps described in Method chapter.

7.1. Checking for a debuggers presence

7.1.1. Technique description

One of the first things that comes to mind when trying to stop a reverse engineer is to hinder him/her from using his tools on your software, which is one of the reasons that checking for a debuggers presence is one of the oldest but yet powerful technique, depending on how good the reverse engineer is.
This technique is easiest implemented by checking the BeingDebugged flag in the Process Environment Block (PEB) [15]. It is the kernel32.IsDebuggerPresent API function that checks this flag to identify if the process is being debugged by a user-mode debugger. However, these sorts of implementations are not worthwhile due to the fact that even plugins can patch these by setting the flag value to 0 itself.
There are many ways to check for a debuggers presence such as scanning the processes to see if popular debuggers are being run by running string searches.
Two cases in this will be studied, one where the software calls the famous isDebuggerPresent API and one where the software snapshots all running processes and searches for “OLLYDBG.EXE”.

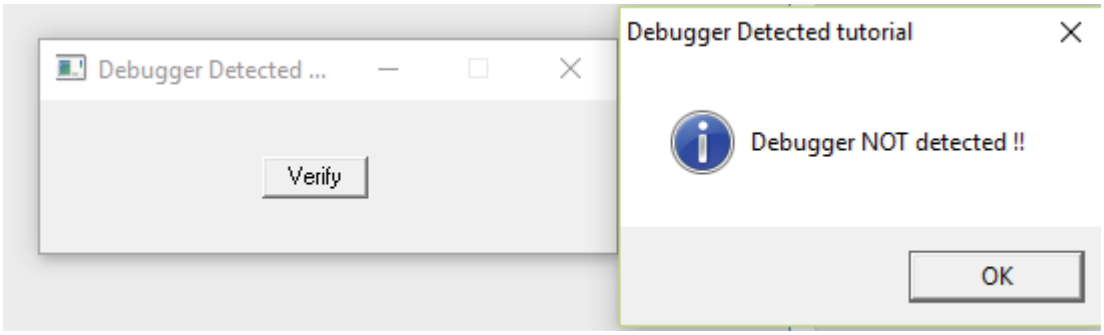
7.1.2. Experiment(s)

7.1.2.1. Debugger Detected.exe

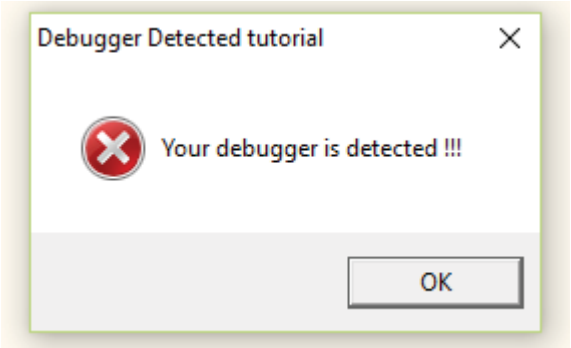
This software can be found in episode 19 of Lena’s Reversing series.

7.1.2.1.1. Observing the program

This program is simple and can therefore be studied within minutes, after executing it, there is only one button called “Verify”. When this button is pressed, a message box pops up confirming that no debugger is detected.



Let's attach OllyDbg 1.10 and see what happens. Indeed, the debugger is detected.

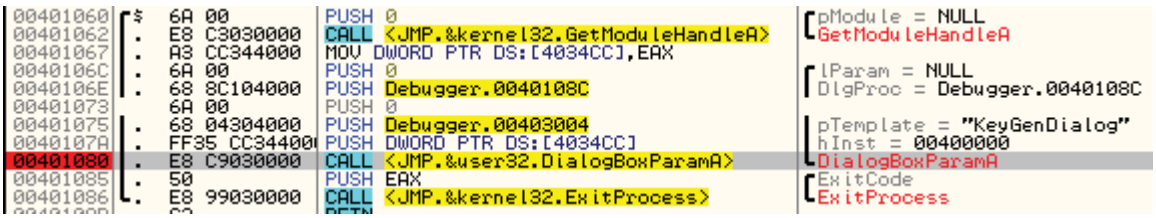


After pressing "OK", the program terminates. It is now time to analyze the software inside OllyDbg to see why we get this error and what can be done to bypass this error message.

7.1.2.1.2. Analyzing the program

First step of analyzing is looking in PEiD for any known packer/protectors used, which indeed was not used in this case, since it is an anti-debugging technique but it could have very well been one added to make it even harder for us.

After opening the executable in OllyDbg, I start stepping the program (running instruction per instruction).
After only stepping 8 instructions, to the selected instruction, we get the error above.



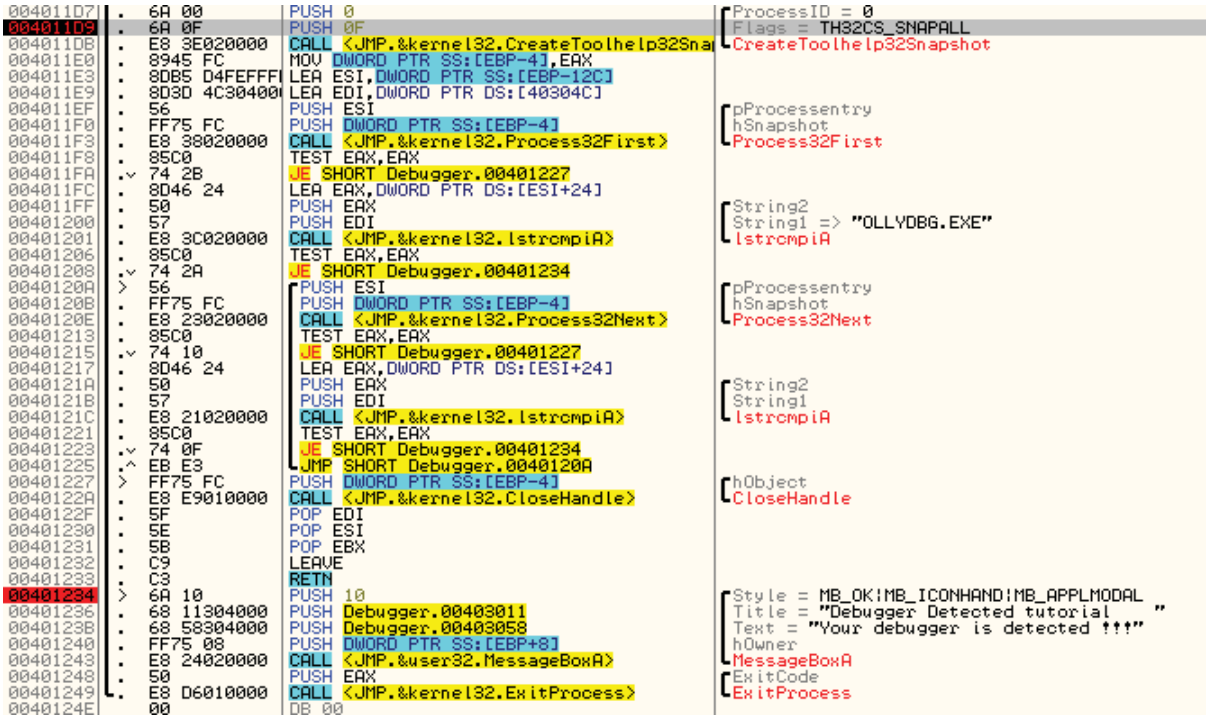
This means that there has to be a check in one of the calls above.

Let us look at the win32.hlp for a better understanding over what the API DialogBoxParamA does.

```
int DialogBoxParam(
  HINSTANCE hInstance,          // handle to application instance
  LPCTSTR lpTemplateName,       // identifies dialog box template
  HWND hWndParent,              // handle to owner window
  DLGPROC lpDialogFunc,         // pointer to dialog box procedure
  LPARAM dwInitParam            // initialization value
);
```

An interesting argument is passed, DLGPROC, a pointer to the dialog box procedure, deeper digging has to be made here.

When analyzing the dialog box procedure, I found that a call was made to another procedure which calls on an API called CreateToolhelp32Snapshot. That API takes a snapshot of the specified processes, heaps, modules and threads. It looks like this procedure searches through every process that is currently running on my machine and then compares it to a string.



If the string matches, it jumps to the second red marked address 00401284 from 00401223 which is the JE instruction after comparing the string.
It looks like we have now found and understood the procedure that gives us the error message box, so how do we go from here?

7.1.2.1.3. Patching the program

The beauty of reverse engineering is that there are many different ways to modify the instructions so that it works exactly how you would want it, for example, we could change the string that the procedure is looking for, example of such is to change it from "OLLYDBG.EXE" to "OLLY.EXE" we could NOP the JE instruction, we could even NOP the whole call to this procedure.
After applying one of the methods mentioned, it appears the program no longer finds our debugger.

7.1.2.1.4. Difficulty, Effectivity and Conclusion

It was not very difficult to patch this program, it took 25 minutes in total and this could have been done much faster by someone more experienced in this field.
I rate the difficulty level to be 3 because of the time taken and the need to look up the different API calls to understand what they actually do and what arguments they get passed.
As for the effectivity, I would rate 3/10 as well because the way this was implemented, snapshotting all processes and then just looping it through to see if any process matches the debuggers name requires a small amount of time but held me off for almost half an hour.

7.1.2.1.5. ReverseMe.A.exe

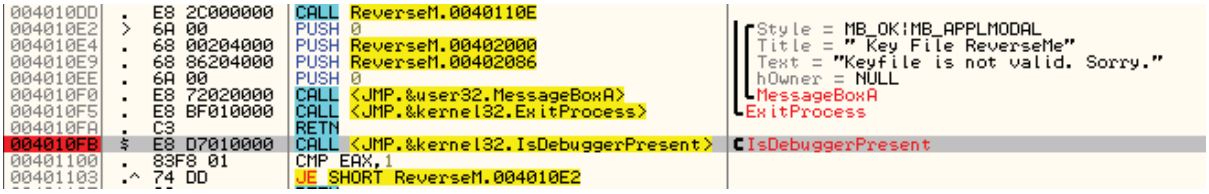
This software can be found in episode 19 of Lena's Reversing series.

7.1.2.1.5.1. Observing the program

Running the program at first just displays a message with the text "You really did it! Congratz!!" So I opened it up in OllyDbg, and whoops, a different message pops up.
This time, the text was "Keyfile not valid. Sorry." This shows that the behavior changed once we started the program with OllyDbg, let's find out why.

7.1.2.1.6. Analyzing the program

A quick check in PEiD shows not packer/protector was used, moving on to OllyDbg.
OllyDbg can be very helpful with its comments after auto-analyzing the instructions. The auto-analyzing option can be enabled in OllyDbg settings under the "Analysis 1" tab.
Because of the comments OllyDbg generated, I quickly saw why we got the change of behavior.



A call to the kernel32 API isDebuggerPresent is made, and jumps to the procedure of creating the message box to display that the key file is invalid. And since we use a debugger, the isDebuggerPresent is going to return "1" (return values are stored in the EAX register) which is then compared in the next instruction, with 1. If the values are equal, a jump to another routine is made, hence the change of behavior that was mentioned.

7.1.2.1.7. Patching the program

Again, there are many ways to go, NOP the JE instruction or replace the call to isDebuggerPresent with MOV EAX, 0 or NOP the whole call to this procedure.

7.1.2.1.8. Difficulty, Effectivity and Conclusion

The difficulty level was very low, 1/10 the reasons for this rating is because OllyDbg helped a lot with its comments and therefore I quickly realized where to look. Took me 5 minutes all together to patch this. The effectivity of this technique also gets a rating of 2/10.

7.2. Use of packers as an anti-reverse engineering technique

7.2.1. Technique Description

What a packer is and how it works has been explained, so the actual technique should be familiar by now. It is said that packers are a good way of stopping the not-so experienced reversers due to the fact that it can be hard to follow inside a debugger [16] and/or make modifications to the executable because of the added protection, which is a good reason this technique should be tested in this study.

7.2.2. Experiment(s)

7.2.2.1. UnPackMe_EZIP1.0.Exe

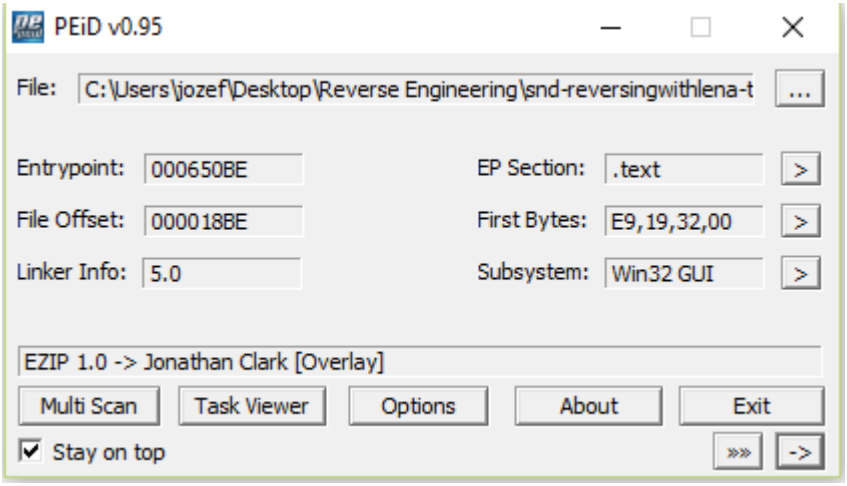
This software can be found in episode 20 of Lena's Reversing series.

7.2.2.1.1. Observing the program

Running the program gives us a hint of what is ahead. A message with the text "If you unpack write a tutorial… :)". Clearly it has been packed with something. Left is a button which you can only press "OK" on and the program terminates.

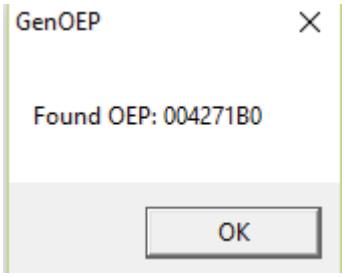
7.2.2.1.2. Analyzing the program

First let us open this program in PEiD.



PEiD suggests that EZIP 1.0 was used, which seems to be true judging by the programs name. Let us see more what PEiD can offer us.

There is a plugin called Generic OEP Finder if you press that "->" in the bottom right corner. Using this plugin gives us this information



PEiD found us an address, called OEP. This means that the original code, the unprotected, starts at this address, let's visit this address in OllyDbg.

7.2.2.1.3. Patching the program

Since we know the OEP of the program, what is left is to dump the real code section, if it works then we know PEiD did indeed find the right OEP (this can be verified in various ways, one other way is to use OllyDbg). There is a built in plugin called "PEiD Generic Unpacker" that automatically does the job for us. Again, there are many ways to dump a program, however this time PEiD was used in order to show how powerful of a tool it can be.

7.2.2.1.4. Difficulty, Effectivity and Conclusion

All of the unpacking was done by using PEiD so not much reverse engineering was needed. Thus I will rate this 2/10 for both difficulty and effectivity. BUT a very interesting thing was noted. The original file size was 228 kB but after unpacking, it is now 444, almost twice as big! This shows how useful role packers played back when storage was expensive [17].

7.2.2.2. UnPackMe_eXPressor1.3.0.1Pk.Exe

This software can be found in episode 20 of Lena's Reversing series.

7.2.2.2.1. Observing the program

When executing the program, a message box pops up saying that it is packed with a demo version of eXPressor, pressing "OK" brings another message saying if we successfully unpack it, that we should write a tutorial for it. Pressing OK to this message terminates the program, no more to observe, let's analyze it with our tools.

7.2.2.2.2. Analyzing the program

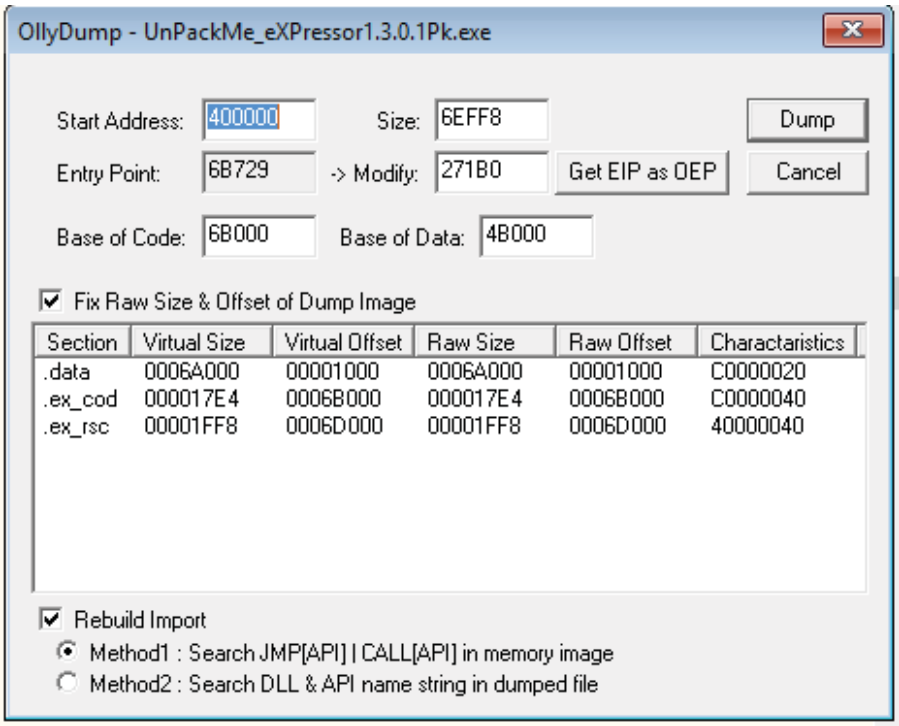
First let us open this program in PEiD.
PEiD suggests that "eXPressor 1.3.0 -> CGSoftLabs" was used, which seems to be true.
Using the plugin called Generic OEP Finder does not help us in this case; it says that no OEP was found, looks like OllyDbg must be used for this one.

There is a method that can be followed for many packers and it is called the "ESP Trick".
The ESP trick suggests that we should set a hardware access breakpoint at the ESP address as soon as it changes. This is due to the fact that once the packer has unpacked itself, it must read its previous values to prepare for the original program. Some packers use the instruction PUSHAD which pushes all 8 general purpose registers onto the stack; this is a big giveaway on when to set the breakpoint.

I found that after only one instruction, the ESP register changed, so I set a hardware breakpoint on this address and then press run once again. As expected, we break on an instruction "JMP EAX" with the address to the OEP.

7.2.2.2.3. Patching the program

Since we now know the OEP by stepping into the instruction JMP EAX, we can now easily dump the program. To do so, I am using a plugin called OllyDump. OllyDump fills in most of the settings for you; usually all you have to do is press dump.



After pressing the "Dump" button you now choose the name of your new unpacked program.
This file grew from 178 kB to 452 kB after unpacking.

7.2.2.2.4. *Difficulty, Effectivity and Conclusion*

This packer was slightly more advanced because PEiD was not able to work its magic on this one, hence research on general unpacking was required. In my quest to find a general method, I came across a paper that described the ESP Trick and other general methods that can be used [18]. This experiment took one hour, mainly because of the research needed and once knowledge was acquired I was only a few steps from finding the solution. Therefore I rate the difficulty level 5/10 and the effectiveness a 6/10 because of how easy one can pack one's software and put a new reverse engineer into a research phase.

7.3. API Redirection

7.3.1. Technique description

API redirection is a technique used by packers/protectors to prevent the reverser from easily rebuilding the import table of the protected executable. When a protector is used, it can intentionally mess with the IAT so that when we dump the unprotected program, it probably won't run because the IAT is broken. On the bright side, this also means that the protector will have to figure out which dlls and functions to load and where to place the pointers so that the original program still operates as intended, a routine for this must be existent.

If the routine of the API redirection is found, we can patch it in a fashion where we skip the routine or if the routine checks whether the API is valid to redirect or not (not all APIs can be redirected), we could patch it so that every check returns “not valid to redirect”.
Let us take a look at software using the API redirection technique.

7.3.2. Experiment(s)

7.3.2.1. API Redirection Tutorial.exe

This software can be found in episode 22 of Lena’s Reversing series.

7.3.2.1.1. *Observing the program*

When running the executable, a window demanding a name and a registration key pops up. Our goal is not to find the executable, but to unpack it. One may wonder how to know whether the executable is packed or not, the best way to do so is to look in a disassembler just like OllyDbg. The easiest way is probably to look in PEiD but please do note that PEiD may not always find if it's packed or not since it is outdated.

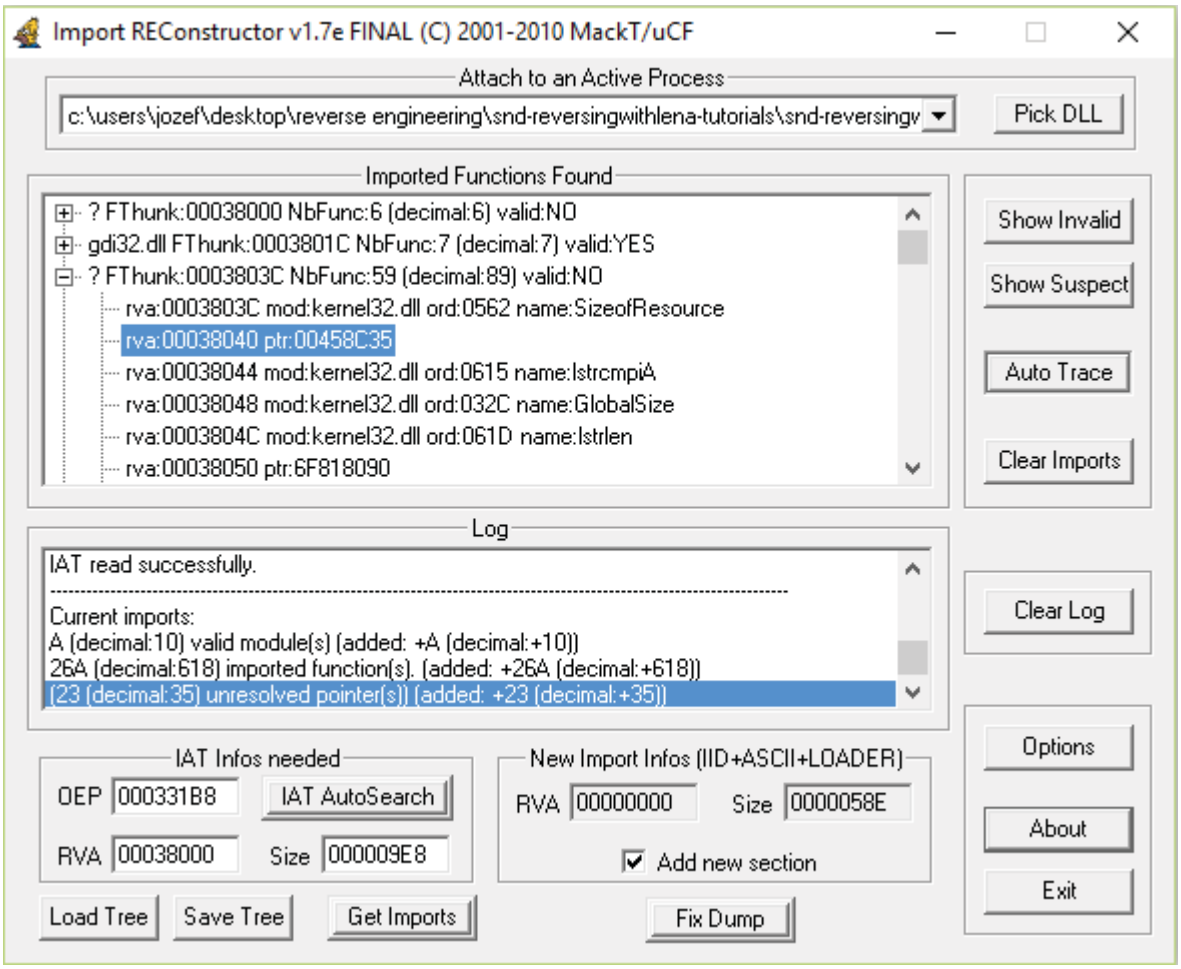
7.3.2.1.2. *Analyzing the program*

PEiD shows some interesting information, looks like a packer called MoleBox 2.x.x -> Mole Studio was used. Let’s debug the executable in OllyDbg. In OllyDbg it is clear that a packer has been used,

mostly because of the first instruction being PUSHAD. After placing a hardware breakpoint (steps have been described earlier) I found the OEP.

7.3.2.1.3. Patching the program

After unpacking the executable, it does not run. This is due to the API redirection technique. Some of the API's has been broken. Attaching ImpRec to the debugged process confirms this.



35 API's unresolved (decimal: 35), which means something is wrong with them. Let's study this more. A program could get the API's virtual addresses used by only calling two APIs, LoadLibraryA and GetProcAddress [19].

> The **LoadLibrary** function maps the specified executable module into the address space of the calling process.
>
> ```
> HINSTANCE LoadLibrary(
>   LPCTSTR lpLibFileName      // address of filename of executable module
> );
> ```

In which lpLibFileName equals the address of filename of executable module and the return value if the function succeeds is a handle to the module.

The **GetProcAddress** function returns the address of the specified exported dynamic-link library (DLL) function.

```
FARPROC GetProcAddress(
  HMODULE hModule,        // handle to DLL module
  LPCSTR lpProcName       // name of function
);
```

The hModule equals a handle to DLL module and lpProcName is the name of the function, the return value if the function succeeds is the address of the DLL's exported function.

To put it simply, LoadLibraryA is called to load a specified dll and then with the handle that this function returns, you can easily retrieve the address of each imported API you want to call with the GetProcAddress.

If we get back to the ImpRec picture, we can see a highlighted API that is invalid, it's at the address 00458C35, let us see where that is with OllyDbg by using the memory map.

00400000	00001000	API_Redi		PE header	Imag	R	RWE
00401000	00037000	API_Redi	0	code	Imag	R	RWE
00438000	0000A000	API_Redi	1	data	Imag	R	RWE
00442000	00005000	API_Redi	2		Imag	R	RWE
00447000	00004000	API_Redi	3	resources	Imag	R	RWE
0044B000	00011000	API_Redi	4	SFX	Imag	R	RWE
0045C000	00001000	API_Redi	5	imports	Imag	R	RWE
0045D000	00008000	API_Redi	6		Imag	R	RWE
00470000	00009000				Map	R	R

It is inside the packer, this suggests that a API redirection indeed has been made.
By placing a hardware breakpoint at 00438040 we can trace where in the packer the redirection has been made so that we can patch it. The current value (when at the OEP) of the address above is 35 8C 45 00. Restart the debugging and trace. The value E0 68 ED 76 is written in the IAT just before jumping to kernel32.FindClose (look at MSDN for what this API does) which means this is probably the real API address. This implies that the address is changed by the packer sometimes later in the code, continue stepping. A procedure (at address 00453E90) which checks whether an API can be redirected or not is found not so far away from where the hardware breakpoint and if possible, use kernel32.VirtualProtect. We have found our patching point; change the conditional jump to always be true so that all the API's show up as non-redirectable.
After patching, we re-do the ImpRec process and the result will be a success, all API's found! Save the dump and use LordPE to clean and rebuild it if so desired, to save more size.

7.3.2.1.4. Difficulty, Effectivity and Conclusion

This technique is considered to be advanced and has by no means been easy to reverse engineer. To patch and understand a technique like this, I had to research about the IAT in general and how windows load the APIs. This caused me to spend more time than expected.
I rate this experiment to be 8/10 for both difficulty and effectivity mainly because of how much knowledge is needed to reverse such technique and of how effective it can be. It took me 2hrs 35 minutes all in all but I expect the next experiment to take less time due to the knowledge gathered from this experiment.

7.3.2.2. UnPackMe_PeSpin 1.32.b

This software can be found in episode 22 of Lena's Reversing series.

7.3.2.2.1. Observing the program

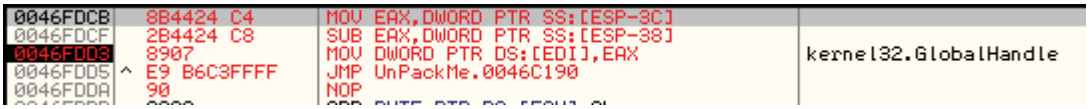
Not much to see, a message box saying it uses the API Redirection technique and that's about it. Press "OK" and the program terminate. This software is written by Teddy Rogers from Tuts 4 You.

7.3.2.2.2. Analyzing the program

PEiD found a packer named PESpin, great, now we know the packers name, we can now search the internet to see exactly what PESpin does and known ways around it! But we are going to do the manual way, which is find the OEP, check for invalid API's and if found, redirect them and finally dump and rebuild.
By performing the ESP trick I found the OEP to be at address 004271B0. When landing at OEP, a call is made to a suspicious memory location at 46FFE1. Step into that call.
Following that call will lead to kernel32.dll module. Let's restart and set a hardware breakpoint at GetVersion. After breaking and a long road of tracing and entering useless calls, the interesting instruction is finally found at 46C18A. Copy the instruction at that address but also the one below it and find a code cave. Looking at the stack, we can see something interesting at address 0x03C and 0x038.
0x03c holds the address the packer should return to after executing the first instructions of the redirected API in that allocated memory location and the 0x038 address holds the size of the emulated instructions.

7.3.2.2.3. Patching the program

Now that we have all the information needed, what is left to do is the actual patching.
There is a code cave at 46FDCB, so why not use it. Assembly these two lines of code and paste the two instructions from 46C18A, this is what is should look like.



Now assembly the instructions at 46C18A + next instruction (since we copied them both to the code cave) to jump to this code cave. Remember to change the last jump instruction so that it really jumps to the correct address (simply pasting the binary code will be enough).
Now trace to the OEP, attach ImpRec, however this time when importing the instructions, we are going to do it slightly differently. Because the packer has spread our APIs across the whole memory, right click in the white space in ImpRec -> Advanced Commands -> Get API calls and press okay (if default settings are activated). Now because of this command we just executed, we have to press the "Show invalid" button and cut all thunks that are invalid. Finally, use the fix dump button (remember to fill the OEP first) and your unprotected executable should now run just fine!

7.3.2.2.4. Difficulty, Effectivity and Conclusion

2 hours and 20 minutes were spent on this experiment; it is safe to say that API redirection can be a very powerful technique. The difficulty of this one was higher mostly due to the amount of tracing required and entering calls which are then found to be nothing interesting. I rate both the difficulty and the effectivity to be a strong 7/10, the score would maybe have been higher if this was the first

experiment for this technique, and since this time I was able to skip the research part and had more knowledge around this technique.

7.4. Stolen bytes

7.4.1. Technique description

Protectors can copy bytes from the original executable and place them inside the protected executable. This means that just simply unpacking and dumping the executable from the OEP will not be enough for the unprotected executable to run properly, due to missing bytes, hence the name “Stolen Bytes”. The stolen bytes are replaced with a jump instruction that jumps to the relocated code. To make it work, a jump instruction back to the instruction that comes after the stolen bytes, from the relocated code. One may think that recovering the stolen bytes is easy, but it can be made even more difficult by filling the relocated code with garbage instructions to make it harder to distinguish the real instructions from the fake ones [20].

7.4.2. Experiment(s)

7.4.2.1. XorIt.protected

This software can be found in episode 23 of Lena’s Reversing series

7.4.2.1.1. Observing the program

When running the executable, a window with the text “By Trial ACProtect” is displayed, pressing “OK” brings us the main program. By the looks of it, the program is a XOR calculator written in Spanish. Since my Spanish is limited, I will move on to analyzing.

7.4.2.1.2. Analyzing the program

PEiD can not find the compiler/packer used so by default it says “Nothing Found *”. Not much of a help, but since we know this program was packed with ASProtect 2.0 (says so in the download description), we are going to trust the uploader that it indeed is protected.
Once we are in OllyDbg, step the instructions until a change to the ESP register has been made. We are doing the ESP trick again, but this time however, we will have to copy the stolen instructions. When breaking from the hardware breakpoint after a POPAD, copy the instructions until the PUSHAD instruction and save them in a text file. These are the stolen “bytes” but obfuscated in a way so that we can't really tell what it is just yet, until analyzed.
Place a software breakpoint after the PUSHAD and see if the ESP register changed, if yes, then place a hardware breakpoint and remove the last one. Now remove the software breakpoint and run (F9 shortcut for OllyDbg). Redo this process until the OEP is reached. OEP in this case is at the address 00401FFC.

7.4.2.1.3. Patching the program

Now that we are at the OEP, attach ImpRec, fill in the OEP, and press the IAT AutoSearch button. ImpRec did indeed find the IAT but failed to recognize the true size of the IAT, going to back to OllyDbg, we can find the correct size by going to the start of the IAT and scrolling down until the end. After this, press the Get Imports button and it should show that all the APIs were imported (*). Make a full dump and open it in OllyDbg, it is time to restore the stolen instructions. Find a code cave and paste the code we retrieved here and do not forget to assembly an instruction that jumps to the OEP after the pasted code.
Open LordPE, select the PE Editor and modify the OEP to the address of the newly pasted code. Save and rebuild. It should now run as intended!

* Since some of the software these experiments are conducted on is old and because of the windows APIs changing with updates and new windows releases [21], ImpRec may say that some of the APIs are invalid. This is because a 0x00000000 separator is needed in between different DLLs for ImpRec to grab all the APIs successfully. If this issue is encountered, one can simply double click on the invalid API and choose the correct dll and function, and ImpRec will do rest of the job to fix it.

7.4.2.1.4. Difficulty, Effectivity and Conclusion

Quite some time spent on this experiment, 4 hours to be exact. This is due to the fact that the unpacked program did not run properly after unpacking. The reason for this is that ImpRec showed invalid APIs even though they actually were valid, just that they belonged to a different DLL but did not have a 0x0000000 separator, a lot of time was spent on realizing this and searching for the fix. When it comes to actually patching this, it was also quite hard. The ESP trick was easy, but I had to break 11 times before I reached the OEP and this could make a new reverse uncertain if he/she is on the right path. Although, once OEP was reached, reconstructing the IAT, finding a code cave, rebuilding and testing were required. I rate the difficulty 7/10 and the effectivity also a 7 because the amount of breaks one must go through can be scaled and thus increasing the time taken.

7.5. Self-modifying code (and decryption)

7.5.1. Technique description

Self-modifying code is a way to trick static analyzing from succeeding since the code changes during runtime. This is an excellent technique to force the reverser out of the static analyzation method, because the code must be stepped to reveal the real code.
Self-modifying code was used to hide copy protection instructions in the 1980s DOS based games [22]. The floppy disk drive access instruction “int 0x13” would not appear in the executable program’s image but it would be written into the executable memory image after the program was run. This is due to this technique.

In software where this technique is applied, it is not to save memory but to trick the reverser. Code is modified on purpose. This technique has been used many times by programs which do not want to reveal what they actually are doing; a famous example of this would be virii.
Knowing how to spot and handle self-modifying code is crucial for malware analysts but also important for the reverser

7.5.2. Experiment(s)

7.5.2.1. ReverseMe Tutorial.exe

This software can be found in episode 18 of Lena's Reversing series.

7.5.2.1.1. Observing the program

A window pops up; with the text "You need to remove the nag…" pressing the OK button will then open the "main program" which contains some text which is unusefull for this experiment. I hope that the size, which is 5 kB, indicates that there is not quite much to observe and play around with, so let's move on to the analyzing part.

7.5.2.1.2. Analyzing the program

PEiD says that MASM32/TASM32 (assembly language) was used. Trying to find the OEP with PEiD will yield the correct result but it will not help much since the self-modifying code technique is in use (so just dumping from the OEP will not be enough).
In OllyDbg, one can search for strings so let's search for the string we saw in the beginning to remove the nag. Searching for the string "You need" will point us to an address at 004012B6. Let's breakpoint at the call to the user32.MessageBoxA and run. The result, OllyDbg never breaks because the instructions are never executed. This is code meant to divert a low skilled reverser hence the name decoy code.

Analyzing the code to see what happens is very important, after all, we are using a debugger so that we can see how it behaves. The fourth instruction, moving something to the register EDI is interesting. Enter the call that comes after this instruction. We find ourselves in a loop which XORs the byte codes with 5A starting from 401000 until 401218. This seems like some decryption going on (the code has previously been encrypted). The process to encrypt is the same as decrypting, just XOR with the same value. This makes it a perfect encryption/decryption technique and has been used widely which is also why it has its own name, enxor.

Placing a breakpoint after this loop will decrypt the code we are looking for and can be confirmed by going to 401000 after breaking. After the loop comes a call to 401011, which is indeed to the decrypted code. Just looking at the instructions that follow it is clear that this code will be modified, by itself. Take a look at the instruction "MOV WORD PTR DS:[EDI], 6A". This instruction basically replaces the two first bytes at the address EDI (401011) is pointing at by 6A. After stepping this instruction, we can see that the instruction at 401011 is no longer XOR EAX, EAX but rather PUSH 0.

Continuing to step, more code will be modified but remember that it has to be executed at some point. A call to 401000 is found, I hope that it is clear that this will indeed run the self-modified code. After the call to 401000 another enxor is found, decrypting the "remove nag" message we saw earlier when observing the program.

To remove the nag, we can assemble PUSH 1 at 40101D, see win32.hlp for this API why this works or we can simply jump past it. After jumping past this, another piece of self-modifying code follows, then jumping right back up again. By continuing the stepping, the main window will be displayed, no nag.

7.5.2.1.3. Patching the program

If the jumping past the nag was chosen, then remember the OP codes this instruction assembled (EB 57) at the address 401016 and 401017, but these bytes are currently encrypted, so different sort of patching is needed here. Scrolling up to 401016 we can see that the current OP Codes are 30 5A and these bytes later get XORed with 5A. XOR EB, 5A = B1 and 57, 5A = 0D. So by assembling B1 0D at 401016 and 401017, it should later during the decryption turn into EB 57.

7.5.2.1.4. Difficulty, Effectivity and Conclusion

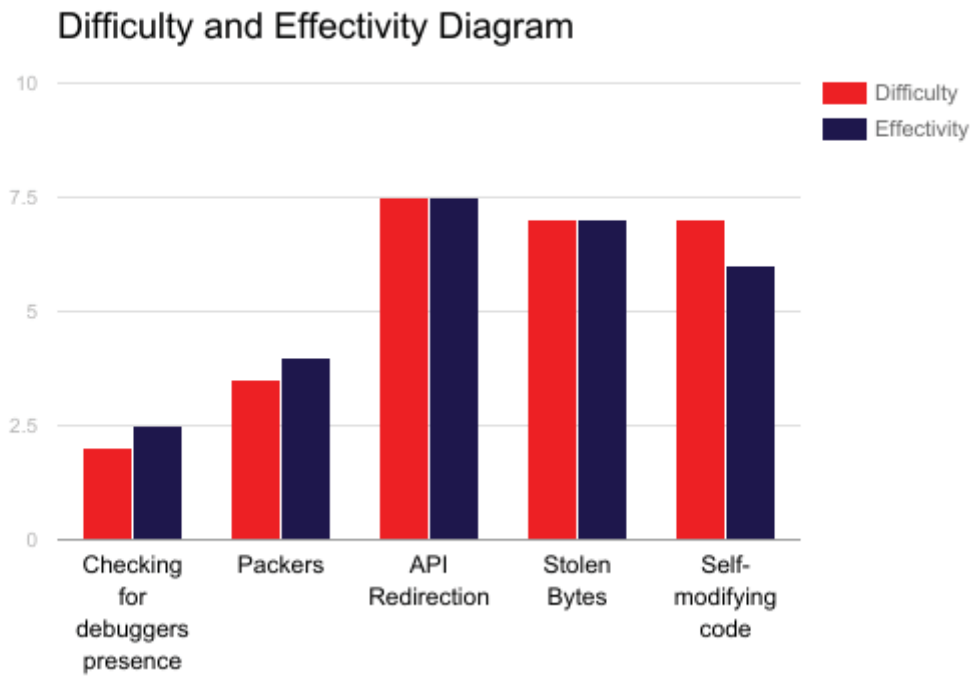
The difficulty of this experiment was high, 7/10. The reason is simply because of the amount of analyzing on such a small file one must perform before gaining proper understanding of the program. Imagine stumbling upon files that are larger than 1mb, a lot of instructions to keep an eye one since the self-modifying code can modify quite a big portion of the code.
It took me 4 hours to patch the software therefore the effectivity falls under a score of 6/10, mostly because a technique like this can take time to implement and is probably not so easy.

8. Analysis and Discussion

My first research question was "Which anti-reverse engineering technique was shown to be effective in this study" required me to conduct experiments before being able to answer it. But before conducting the experiments, knowledge around reverse engineering had to be acquired. After knowledge was acquired and my eagerness peaking, the experiments were started. The techniques were the studied and analyzed by debugging real software, with the added protection added, inside OllyDbg.

8.1. Which anti-reverse engineering technique was shown to be most effective?

With all experiments done and documented, the first research question can now be answered, "Which of the anti-reverse engineer technique was shown to be most effective?" In each of the experiments, the difficulty, effectivity and time taken has been documented in the section "Difficulty, Effectivity and Conclusion". To get a better overview, a diagram with the value 10 in the Y-axis and the techniques in the X-axis has been made. The red color indicates the difficulty to get past the protection added. The dark blue color indicates the effectivity of this technique, which is based on a few factors such as; time taken and an approximate of implementation time.



As shown in the graph, just checking for a debuggers presence is not enough to protect your software, but please keep in mind that there are more advanced ways of implementing this technique, two different implementations were experimented on.

Packers alone are not enough because there are general guidelines one can follow, such as performing the ESP trick to easily unpack the software. The reason for the low effectivity and difficulty for these techniques is that they are famous and not advanced, and a packer alone will not do much more than just unpacking the software, if not modified with extra layers of protection, such as API Redirection or Stolen Byte for an example.

The API Redirection technique was shown to be most effective with a score of 7.5 for both effectivity and difficulty, with the Stolen Bytes and Self modifying code very close to it.
When experimenting with the API Redirection protected software, thoughts of how powerful this technique actually can be came to mind.
I started experimenting with the technique I thought was easiest. Right to left in the graph was supposed just jump in effectivity and difficulty but as one can see, API redirection, in the middle, came as a shock. It is important to keep in mind that a technique like API redirection can be hard to implement if done manually, but there are protectors which can do it in a small time but the difficulty of the reversing needed probably will be less than if done manually because commercial protectors is discussed in forums and therefor may not as secure as one would want them to be.

8.2. Can the most effective technique found in this study be further customized towards the software?

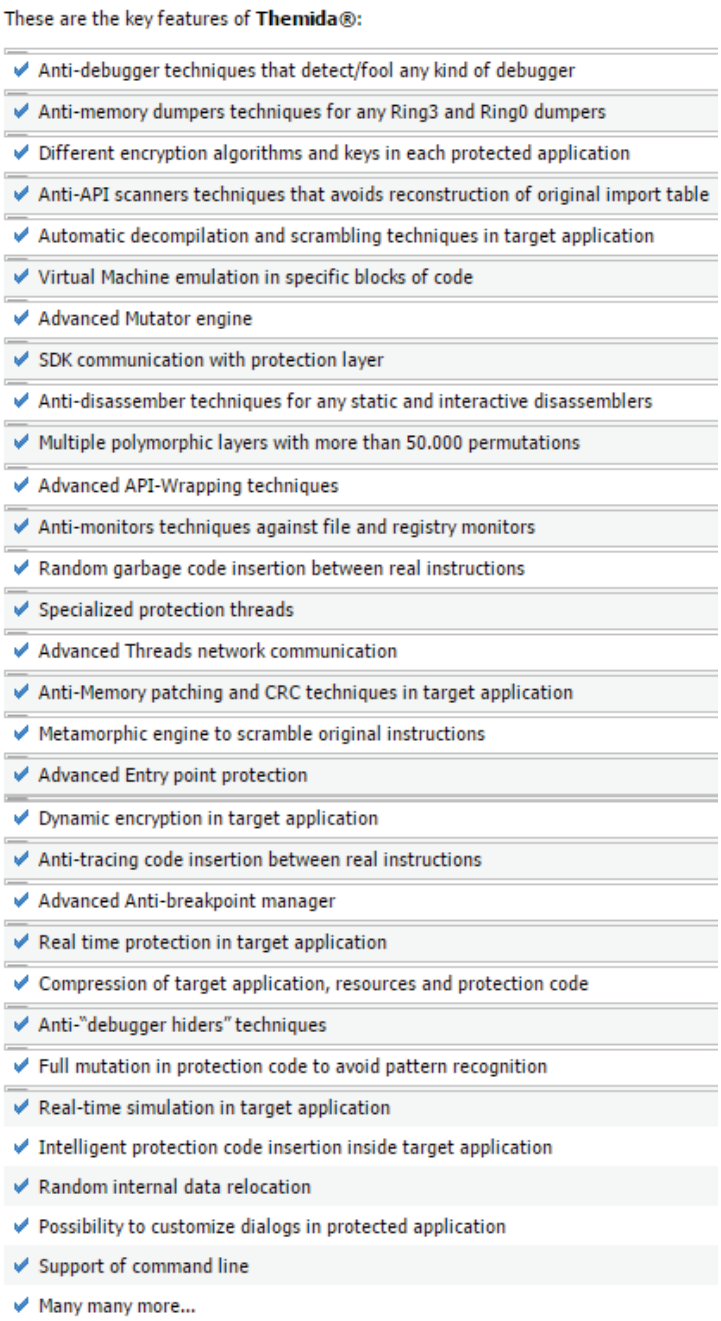
To answer this question directly, yes it can be customized towards desired software.
Both of the experiments that were done on the API Redirection technique required much knowledge around what was going on, how an executable works, how the API's addresses are acquired, how the packer/protector does and how an IAT works.
It is safe to say that a reverser with not so much experience will either be lost or give up when challenged by such a technique or if determined enough, he/she will have to do research to gain better knowledge.
As for the question, how can it be customized? Well, that would depend on if one is using a commercial packer/protector or if using a private one. When using a commercial packer/protector you can not specifically tell it (of which I am aware of) where to store the redirected addresses for example or which API's it should redirect.

If one were to use a private packer/protector, then you have the ability to do all the customizations you want. In that case one could choose the API's to redirect, where to store the addresses and even add more advanced stuff like API hooking. All DLL modules are not prone to be redirected due to different reasons hence one should be careful when choosing the API's or have a routine to check which ones are [19]. To shortly explain API hooking, it can for example be overwriting bytes in one function to jump to another which is some sort of code injection and when later compiled into a DLL file, the customized code will execute because of the overwritten bytes (hooking).

In my opinion, if a developer wants to customize a technique like this for his/hers software then reading about IAT and how they are constructed along with how a simple API Redirection works is recommended to better understand the working of this technique and as a result able to implement it better.

8.3. Can the techniques be combined in order to further strengthen the protection?

Yes, and this is what modern packers/protectors do. They combine known techniques proven to be effective to further strengthen the protection of the software. An example of techniques that Themida from Oreans Technology uses:



Some of the techniques studied in this thesis are key features in Themida's product [23].
After analyzing these techniques and gaining more knowledge about them, I have found that they can be combined, as long as they do not conflict with each other in some way. A conflict could for example be two techniques that modify same sections inside an executable; this could result in errors when trying to run it.

If we were to look at the techniques that I have been experimenting on, a suggestion would be to combine the Self modifying code and API redirection technique, they do not conflict each other; API redirection redirects API's and self-modifying code modifies pieces of codes during run time. Looking at the diagram, combining those two techniques would give a reverse engineer a hard time. Then again, there is nothing that stops the developer from combining three techniques, or even more, but adding too much of protection could result in a larger file size and potentially worse performance. Imagine having code that modifies into the API redirection routine, possibly even limiting the routine to only redirect some of the API's and then have another routine to redirect the rest.
A skilled reverser would probably not be stopped by only these two techniques, but as for the less skilled reversers, they probably would have to spend quite some time in figuring it out.

9. Conclusion

The research questions raised can all be answered with the research methodology chosen, which is by conducting experiments. The first research question which was to find out which anti-reversing technique was the most effective required me to wait until all experiments were finished before I was able to analyze the result. The answer to the question in this study was the API redirection technique as shown in the comparison graph "Difficulty and Effectivity Diagram" in chapter 8.1. The analysis of why this was most effective technique can also be read about there but also under the experiments conducted for this technique.

The second research question was about finding out if was possible to further customize, the found answer to this question is that it can be customized but it would depend on whether you own the protector that adds this technique or not.

Last question which was about finding out if the techniques can be combined to get better protection was shown to be possible. From my experiments I learned that it should not be a problem as long as the techniques are not conflicting with each other in some kind of way, touching the same parts of code etc. Many commercial protectors uses many techniques combined to get a better protection, rather than trusting that one anti-reversing technique is enough. Themida's Advanced Software Protection System is an example of such commercial protector.

10. Future work

There are many techniques that one can implement to protect their software against reverse engineering, but the question still remains, which is proven to be most effective?
In this study five techniques were compared, but having a larger comparison brings a better overview and higher value for someone that would want to implement some kind of protection.

In this study I did not look at performance gain/losses after implementing the techniques, mostly because I wanted to focus on the effectivity of the techniques, but looking at the performance before and after adding protection also brings a lot of value.

Lastly, If more techniques were to be compared by follow up studies then one could focus on the most effective technique and try to tailor it towards a software and then comparing it to a non-tailored (for example an packer/protector), studying the differences in effectivity/difficulty.

References

[1]. https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software - Oleg Kulchytskyy, Anti Reverse Engineering Protection Techniques to Use Before Releasing Software, Apriorit, 2016-09-05.

[2]. http://www.darkreading.com/partner-perspectives/malwarebytes/malware-explained-packer-crypter-and-protector/a/d-id/1328458 - Packers & Protectors, Pieter Arntz, Malware Explained: Packer, Crypter & Protector, DarkReading, 2017-03-22

[3]. https://www.kaspersky.com/resource-center/threats/suspicious-packers - What are Suspicious Packers?, Kaspersky, 2017-04-25

[4]. https://support.microsoft.com/en-us/help/815065/what-is-a-dll - What is a DLL?, Microsoft, 2017-04-26

[5]. https://msdn.microsoft.com/en-us/library/ms809762.aspx - Matt Pietrek, Peering Inside the PE: A Tour of the Win32 Portable Executable File Format, Microsoft System Journal, 1994

[6]. http://www.cs.scranton.edu/~cil102/data_bits.html - Bits, Bytes and Words, University of Scranton, Computing Sciences, Accessed on 2017-05-16

[7]. Teodoro Cipresso, Software reverse engineering education M.S. thesis, Dept. CS, San Jose State Univ. San Jose, CA 2009

[8]. https://en.wikipedia.org/wiki/Assembly_language - Assembly Language, Wikipedia, last edited on 2017-05-17, Accessed on 2017-05-21

[9]. https://en.wikibooks.org/wiki/X86_Assembly/Introduction - X86 Assembly/Introduction, Wikibooks, Last edited 2014-06-22, Accessed 2017-05-14

[10]. https://en.wikipedia.org/wiki/Disassembler - Disassembler, Wikipedia, last edited on 2016-11-7, Accessed on 2017-05-19

[11] Reversing: Secrets of Reverse engineering, Eldad Eliam ISBN: 978-0764574818

[12]. http://www.ollydbg.de/quickst.htm - Oleh Yuschuk, OllyDbg v1.10, OllyDbg Quick Start Guide, Accessed by 2017-05-17

[13]. http://www.geeksgyaan.com/2016/03/anti-debugging-techniques.html - Abhishek Choudhary, 5 Anti Debugging Techniques That Will Protect Your Software, Geeks Gyaan, last edited on 2016-06-20

[14]. https://ethics.csc.ncsu.edu/intellectual/reverse/study.php - Ethics in Computing, Reverse Engineering, NC State University, Accessed on 2017-05-15, last edited 2004

[15]. https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf - Mark Vincent Yason, The Art Of Unpacking, IBM Security Systems, 2007

[16]. http://www.strongbit.com/execryptor.asp - Stop Crackers and Software Pirates, StrongBit Technology, Accessed on 2017-05-13

[17]. http://www.statisticbrain.com/average-cost-of-hard-drive-storage/22 Average Cost of Hard Drive Storage, Statistic Brain, research conducted 2016-09-02 Accessed on 2017-05-14

[18]. https://www.diva-portal.org/smash/get/diva2:812426/FULLTEXT01.pdf - Danut Niculae, General Unpacking: Overview and Techniques 4, 2015-05-18

[19]. http://www.ntcore.com/files/inject2it.htm - Ashkbiz Danehkar, Injective Code Inside Import Table, ntcore, released on CodeProject 2005

[20]. http://pferrie.tripod.com/papers/unpackers.pdf - Peter Ferrie, Anti-Unpacker Tricks chapter 1.d, Microsoft Corporation

[21]. https://msdn.microsoft.com/en-us/library/ff818516(v=vs.85).aspx - Windows API Index, Microsoft

[22]. https://pdfs.semanticscholar.org/a75f/64ec687dd2d969d9c9c67ac41fd8ee6f2fac.pdf - Bertrand Anckaert, Matias Madou and Koen De Bosschere, A Model for Self-Modifying Code, Ghent University

[23]. https://www.oreans.com/themida_features.php - Key Features of Themida, Themida, Oreans Technologies

[24]. http://www.osdever.net/documents/PECOFF.pdf - Microsoft Portable Executable and Common Object File Format Specification Microsoft Corporation, page 7. Accessed on 2017-05-16

[25]. https://tuts4you.com/download.php?list.17 – Lena, Lena's Reversing for Newbies, Accessed 2017-01-19