Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Aleksander Bjørkhaug

Finding Educationally Friendly Malware

Defnining indicators and creating a framework

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2021

NTNU
Norwegian University of
Science and Technology

Aleksander Bjørkhaug

Finding Educationally Friendly Malware

Defnining indicators and creating a framework

Master’s thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Kunnskap for en bedre verden

Abstract

This report is a master thesis whose purpose is to understand what educationally friendly malware is, and help educators find them. We want to find what indicate that it is educationally friendly and to create an automated framework that can help educators spend less time doing manual analysis to find them.

With the increasing number of systems connecting to the Internet, the attack surface of malicious software, known as malware, grows. To combat malwares, we imply antimalware systems for detection and removal of them, but before the antimalware system knows what it is looking for a human must have done an analysis. Malware analysis is a difficult subject that requires deep understanding on subject such as operating systems, assembly language and malware techniques. [1] In later years there has been an increase in educational resources, such as university courses or online classes, that makes it easier to become a malware analyst. We want to help the educational process and give the educators a tool that can improves their efficiency in finding malware samples suitable for new students to novice reversers. By shortening down the time used to analyze malware the educators can focus more on other aspects, such as creating better course content that motivates and makes better malware analysts for tomorrow. With better analysts we can create better detection systems and have a sharper edge against malicious software in the future.

From the autumn of 2020 to the spring of 2021, we interviewed experts and researched educational resources to understand what educational malware was. We discovered what was undesired in a malware sample, and indicators that could be used to locate potential samples. We then analyzed different workflows and how you could do malware analysis, to discover tools that could help us find educationally friendly malware. After we knew what to look for, and which tools that could give us that information, we created a program that could take an arbitrarily set of malwares. The program creates a subset of potential educationally friendly malwares with an accompanying report that explains what is found and where.

To verify that the program created a usable output for finding samples, we created scenarios and tested them from a "blind" perspective. We found by using the scenarios and working through the defined process, that we were able to find suitable malware from the sample sets. The process from having no sample and no previous knowledge about the samples, to having a sample that met our criteria was significantly more streamlined and gave results quickly. We spent minimum time in a disassembler, only to verify findings and to assess if the code sections could be used for educational purposes.

Sammendrag

Denne rapporten er en masteroppgave som har mål om å undersøke hva utdanningsvennlig skadevare kan være, og hjelpe lærere med å finne dem. Vi ønsker å finne hva som identifiserer skadevaren som utdanningsvennlig, og lage ett automatisk rammeverk som kan hjelpe lærere finne disse filene ved mindre tidsbruk rundt manuell analyse.

Med et økende nummer av systemer som blir sammenkoblet gjennom internettet, så blir angrepsoverflaten til ondsinnet programvare større. For å bekjempe ondsinnet programvare, også kjent som skadevare, så installerer man anti-skadevaresystemer for å oppdage og fjerne dem, men for at systemet skal vite hva den ser etter må ett menneske ha gjort analyse først. Skadevareanalyse er et vanskelig fagfelt som krever dyp forståelse for blant annet operativsystem, programmeringsspråket Assembly, og skadevareteknikker. [1] I senere år så har det vært en økning i utdanningsmuligheter, gjennom universiteter eller kurs på internett, som gjør det lettere å bli skadevareanalytiker. Vi ønsker å hjelpe utdanningsprosessen ved å gi lærere ett verktøy som kan forbedre tidsbruk og effektivisere arbeidet som gjøres for å finne skadevare som kan brukes for utdanningsformål fra nye studenter til viderekommende analytikere. Ved å kutte ned tiden brukt på å analyse skadevare så kan lærere fokusere mer på å lage godt og motiverende undervisningsopplegg som skaper bedre skadevareanalytikere for fremtiden. Med flinkere analytikere så kan vi lage bedre deteksjonssystemer slik at vi bedre kan bekjempe ondsinnet skadevare i fremtiden.

Fra høsten 2020 til våren 2021 så har vi intervjuet eksperter og gjennomgått forskjellig utdanningsressurser for å forstå hva utdanningsvennlig skadevare kan være. Vi fant ut hva som er uønsket ved ett skadevareeksempel, og indikatorer som kunne identifisere mulige filer. Vi analyserte så forskjellige arbeidsflyter og hvordan man kunne gjøre skadevareanalyse for å finne verktøy som kunne hjelpe oss finne utdanningsvennlig skadevare. Etter at vi visste hva vi skulle se etter, og hvilke verktøy som kunne gi oss den informasjonen så lagde vi ett program som kunne ta ett tilfeldig sett med skadevare. Programmet produsere så en delmengde med mulige utdanningsvennlige skadevarefiler med en tilhørende analyserapport som forklarte hva som ble funnet og hvor.

For å verifisere at programmet lagde en delmengde som inneholdt utdanningsvennlige filer så lagde vi senarioer og testet analysene «blindt». Ved bruk av senarioer og den nye arbeidsflyten som rammeverket gir, så fant vi skadevare som var passende for senarioet. Vi så at prosessen fra å ikke ha noen skadevareeksempler og ingen forkunnskaper om filene, til vi hadde filer som møtte kriteriene våre var signifikant mer forenklet og ga raske resultater. Vi brukte minimum med tid i en disassembler, og brukte den bare for å verifisere funn og undersøke om kodeseksjonene kunne brukes i en utdanningssetting.

Acknowledgements

This project was carried out at the Norwegian University of Science and Technology (NTNU) through the Faculty of Information Technology and Electrical Engineering. It is the result of my master thesis to receive master's diploma in information security.

I would like to thank my supervisor Geir Olav Dyrkolbotn for being patient and accommodating with my working process. Without him I would not have been able to reach out to experts within this field.

My workplace, The Norwegian Armed Forces Cyber Security Center, has been instrumental in this thesis by giving me the experience and opportunity to work with malware analysis. They have sent be on courses and given me room to further my knowledge, and I would like to thank them for their support.

I would also like to thank my future wife that has help me through this master thesis with support and motivation.

Aleksander Bjørkhaug
Senior Sargent, Norwegian Armed Forces
Lillehammer, June 2021

Content

List of figures

List of tables

Glossary

Assembly code: Machine code converted to human readable Assembly language.

C and C++: A programming language.

Python: A programming language.

Disassembly: The conversion of machine code to Assembly code.

Malware: A program with malicious intent and capability.

Malware analysis: Process of gathering information from a malware file.

Static malware analysis: Analysis process that never executes the malware sample.

Portable Executable: Standalone binary file that can execute machine code on the system.

Exe-file: Runnable portable executable file.

Dll-file: Library portable executable that gives functionality.

Execute, to execute a file: To run or to start a program.

1 Introduction

Malicious software, known as malware, has been a problem for people and corporations since before the internet was widely adopted. [2] In later years after the internet has become an integrated part of our daily life, almost anything technical are interconnected through the web. Malware now has an exponentially larger attack surface with potential victims connecting to the internet at every hour of the day. The purpose of the malware varies from sample to sample, and it could be anything from keyloggers, to full system takeover and destructive behaviors. One thing most malware has in common, is that they want to remain undetected from antivirus and antimalware systems for as long as possible. [3] Antimalware systems and -software base their detection technologies on different mechanisms such as signatures or patterns. At some point, a human must have done an analysis of the malware or a related sample to create a detection indicator. Malware analysis can be a difficult and tedious task, that require both broad and deep knowledge and understanding of how to write code, different programming languages, operating systems, and what an attacker is after and how they can achieve their goals. There are different roads to becoming a malware analyst, or malware reversers as they are also known as, but the easiest way is through educational courses or training programs.

Malware analysis courses are becoming available through many different universities globally. There are also many online resources, that either are free or has paid tuitions, that teach the same subjects with their own variations. A corporation can also have an internal training regime that takes untrained cyber security personnel and turning them into a malware analyst. Whichever way is taken, someone must create a plan and content for the educational resource. Malware analysis and reversing is a practical craft, and a course would therefore require practical examples and training exercises. Finding good examples and doing the analysis yourself as an educator can be difficult. It is a time-consuming task and often requires a deeper understanding of the sample, time to prepare notes and references for the course and creating suitable learning exercises.

The educator might have to start from scratch and find new and undocumented malware to work with. Analyzing new unknow samples could waste the educators time as the file might be too advanced or too complex to be suitable for use in novice courses. It could even miss the functionality needed for the course making the analysis useless. Given that educators should create a compelling and interesting course, as well as finding examples, time would be running short if they also have other responsibilities. If the educator has done the course before but does not work with reversing new samples at a regular basis, they might just reuse older sample. The samples might illustrate to the student quite well what the educator has been trying to teach, but the examples given will most likely not include new techniques used by malware authors. In a worst-case scenario, the course would teach techniques that are completely remediated in modern operating systems.

Doing malware analysis and reversing is something the author of this master thesis is proficient with. He has been in the field for 5 years and taken courses and certifications in both general analysis and specifically binary reversing. He has also been in the educational sphere through creating in-house training program and holding workshops for new aspiring analysts. He holds, annually, a longer lecture at the Norwegian University of Science and

Technology (NTNU) in their “Reverse Engineering and Malware Analysis”-course. [4] Through these experiences, the author believes that he can be seen as a professional and an expert in the field of malware analysis and reversing, and knowledgeable in what is needed for malware education.

In this thesis we will look what makes a malware sample educationally friendly, and how we can help the educational process and reduce time spent with preparations. We want to make it more efficient for both educators and analysts that want to learn and practice their craft. We want to make it easier for people to find samples and shorten the time from having unknown samples to knowing how to use the sample in an educational setting.

In this master thesis we will first define what we want to solve through specific research question, what delimitations we are setting for ourselves and what other has done withing the topics described. To enable the reader to be able to follow along with the work done in this report we will explain some necessary theory. After we have the necessary basic understanding, we will go through our method of solving the research questions. Here we will highlight interview quotes and our findings from our own research. Here we will also explain our creation process of the framework and what we thought with the different parts. After that we will present our results and findings and use those results in life-like scenarios to test the usefulness. The results of tests will then be used to analyze if we have improved on the points presented from the interviews. When we are done analyzing our results, we will discuss our findings against the research questions we defined to see how we have answered them with our work. After that we are going to discuss weaknesses with this project and potential future work to improve it. At the end we will conclude our project with what we have done, what we found out and how successful we were. After that will add the references and attachments; interview guide, transcribed interviews and interview consent form.

1.1 Topics covered

As we have stated earlier; malware has always been a problem for computers and users alike. There are many different kinds of malware, but they all have a malicious purpose and can be quite advanced in obtaining their goals. Malware has from the beginning evolved to have new features and countermeasures against analysts and detection systems, and they will keep evolving in the future. This is a problem for security experts that tries to protect computer systems. Cyber security could be said to be an arms-race between the malware authors and the security experts to develop the best attack vs. the best defenses.

Malware analysts are security experts specialized in taking apart malware and analyzing its components to understand its capabilities. Becoming a malware analyst, or malware reverser, is often a long and difficult path. There are books and courses that gives introduction to the field, but the examples given are often outdated or are too simple for real life situations. There is a void of real-life samples between entry level malware reversing from books or courses to experts capable of reversing any sample given. Reversing a large set of malwares manually to locate acceptable level of difficulties would be a too time-consuming exercise, so we should look to automated analysis.

In this project we are going to answer the question:

"Can we with the help of automated analysis shorten the time used to find malware samples suitable for educational purposes by categorizing and capability tagging malware samples from an arbitrarily large set of unknown malwares?"

To be able to answer this we must find out how much time different experts use to verify that we have improved the process. We then must specify what must be included in the malware analysis and define what educationally friendly malware is. Lastly, we will create a proof-of-concept and test our method of automated analysis and test that the output given can be used to shorten the time spent finding new samples to use in an educational setting.

1.2 Keywords

Malware analysis, static analysis, automation, education, malware functionality, tagging, proof-of-concept, categorization.

1.3 Problem description

There are many tools that help with analysis of malware. Most rely on running the malware, i.e., dynamic analysis, and some rely on analyzing the file statically without running it. As it is a time-consuming task to manually reverse engineer samples to find relevant code sections and examples for education, we want to find tools that help us find the relevant sample for us. We will only use static tools and techniques on the malware samples in this thesis. This is because we want to reduce the resource intensity and we would not want to infect our system with countless unknown malware samples. Through the usage of tools, we will generate report files that use the results of the tools and highlight different findings that makes it easier for an educator locating promising samples. We will through the use of indicators of educationally friendly malware, try to sort or score the samples so it would be even quicker to locate possible usable samples.

1.4 Justification, motivation and benefits

When looking for good educational malware samples, a large set of malwares to look through is preferred. By having a large set of malwares, the educator would use most of their preparation time on finding suitable samples instead of preparing the best possible lessons. By using automated tools in a framework, the educator could minimalize their time of analyzing malware and instead focus on creating a better lesson. Or in a work environment, the tool framework can help the new malware analyst find the sections of the code relevant for their work or help them find samples for furthering their understandings on malware. By testing out the concept on our own set of samples, and verifying that we get usable output, we will have created a tool usable for other educators or malware analysts to find and automatically analyze malware.

1.5 Research questions

In this project we will investigate the following research questions:

1. What constitute static malware analysis, and which tools are the "state-of-the-art" of statically analyzing malware samples?
2. What indicates that a malware sample is educationally friendly or suitable?
    a. How can we use tools to find and display these indicators or features?
3. How much time do normally educators use to find and understand their malware samples to use in an educational setting?
4. Are we able with minimal manual verification able to list possible educationally friendly malwares from an arbitrarily large set of unknown malwares?
5. Can we through automated static analysis shorten the time an educator uses from having unknown samples until they have samples with the right techniques wanted to use as examples or exercises in an educational setting?"

1.6 Delimitations

In this part we will go through our delimitations and restrictions that we have placed on this master thesis. We have these restrictions so that we can focus our work on specific, but common, types of malwares so that we are more likely to come to a useful conclusion for other educators or analysts.

In this master thesis we will only focus on malware samples that are Portable Executables and written and compiled with a C or C++ compiler. We will however not differentiate between 32-bit or 64-bit samples. We have restricted us to C and C++ because it is the most common form of malware that contain least excess, benign code. [5] There is a flora of different packed malware that rely on less common compiling methods, such as Delphi [6], that is too packed or obfuscated to be analyzed in any meaningful way statically. The end framework should be able to differentiate the different types of binary samples and be able to filter out undesirables.

We will only focus on techniques that can be done statically on the binary sample. We have decided to rely on static analysis to find educational friendly malware because this method is more reliable with a large quantum of binary samples to analyze. We want the framework to be able to bulk analyze so that the user can pick and choose which samples they want and have a better chance of finding a suitable sample for their use-case. We do not wish to run unknow malware samples on our system, even if it is in a virtual environment. Worst case, it could escape out of the analysis environment and infect the host machine. Or more likely; one or more samples would encrypt our analysis environment and render all reports useless.

Next, we have decided that we will try to find tools that are compatible with the python programming language. We have decided to use this language because the writers are most comfortable with this scripting language, and it is the go-to language for many malware analysts and tool writers.

It is worth noting that this master thesis paper is written part time. The writer works fulltime while working on this project, and therefore might be at a disadvantage to students that writes fulltime.

1.7 Related work

In this chapter we will to at previous work that have tried to either categorize, automatically analyze or explain a malware's functionality. There has been done a tremendous amount of work on the subjects of categorization and analysis of malware in academia, private and public sectors. There seems to be a weighting towards finding different categorizations methods of malware in academia, and a more focus of finding capabilities in the private or public sector.

Ferhat Ozgur Catak (et al.) [7] wrote a paper on a method of detecting and categorization of malware based upon dynamic analysis and machine learning or deep learning. The detection was based upon behavior and especially the use of Windows function calls. The goal was to classify malware that were protected by different techniques, such as morphic malware. The classification resolution did not include family name, but were on the level of "adware", "backdoor", "downloader", and so on.

Savan Gadhiya and Kaushal Bahavsar [8] have listed what they believe to be the basic steps to take when doing malware analysis. They talk about both static and dynamic analysis of malware. In static analysis they highlight techniques such as file fingerprinting and extraction of hard coded strings, and in dynamic analysis they present two main methods: Comparing a system state from before a sample is executed with one after it has been running and observing runtime behavior. What they talk about in their paper corelate quite nicely with what Michael Sikorski and Andrew Honing wrote in their comprehensive book "Practical Malware Analysis". [1] Their book on malware analysis is the go-to book for many educators and novices in the field to learn everything from basic static and dynamic analysis to advance techniques.

FireEye with Willi Ballenthin and Moritz Raabe created a tool to automatically identify malware capability statically. [9] The tool, named capa, is created to help malware analysts quickly understand and get an overview of a sample's capabilities and functions. The tool is a binary search engine that takes rules or specialized search queries to find matches within a sample. The team behind the tool has released an extensive rule repository that makes the tool a powerful helping resource for any malware analyst.

2 Theory

In this chapter we will go through and explain important topics and concepts. We will focus our explanation of topics that are directly intertwined with what we will do in this project. We might also brush over some complementary parts that are not directly relevant, but important to understand the subject. We want to give the reader a good basis to understand the terminology used later in this paper. If you have worked with malware analysis before you might already have the knowledge needed to understand the following chapters.

2.1 Malware analysis

We use the term malware analysis, or just analysis, frequently throughout the paper. By malware analysis we mean that we investigate the malware samples properties and functionality to try and understand what it does. There are multiple of different ways to gather the data and how to interpret the data. We normally split the methods into two major groups: static analysis and dynamic analysis. Both static and dynamic analysis can be split into two subgroups of basic and advanced techniques. [1]

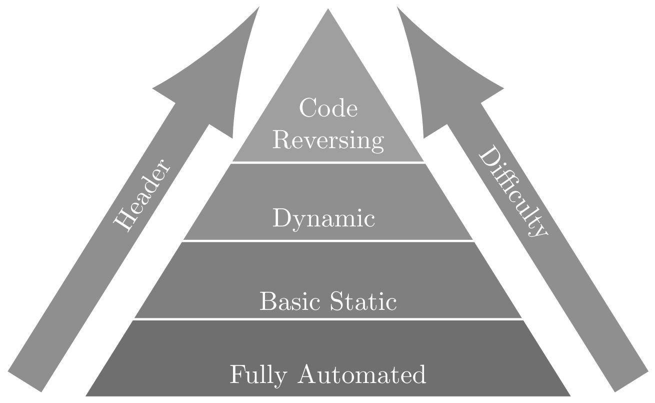


Figure 1: Malware analysis triangle [10]

Figure 1 illustrates the different stages that a malware analysis process could have. It starts at the bottom and become increasingly more difficult and challenging for the analyst. The higher up the “pyramid” an analyst are able to get, the more valuable information they might extract from the sample. The lowest level and the technically easiest to execute is “Fully automated” analysis. This often means that an analyst, or an automated system, send the sample to a “sandbox” that executes several analysis processes on the sample. A sandbox is an automated analysis framework that employ virtual machines and analysis tools. The sandbox automatically starts a new virtual machine and executes the sample inside it, monitors the system and the samples actions, and generate a report based on its findings. The final report often has data about runtime information from the execution of the sample, such as system functions use or network activity, and the sandbox gives the sample a threat score. What is needed from the analyst at this stage is to understand the report and the threat score that it gives, and the report are often so simplified that non-technical personnel can form an opinion on the sample and course of actions.

Basic static analysis is then we gather information from the sample without executing it. We use tools that are able to parse the sample file and extract metadata about the sample. Metadata that a malware analyst is interested in is often; what type of file is it, is it compiled and how, what imports does it have, are there anything unusual indicators in the header data? Another static technique is fingerprinting the file and looking for already done reports on the file. By fingerprinting we meant that we mathematically generate a unique hash of the sample that we can use to identify it regardless of who has done the analysis. We often fingerprint using MD5 hashing [11] or the SHA256 hashing [12] to calculate a uniquely identifiable number. A popular static technique is to iterate bytewise through the sample to gather bytes that can be displayed as a written character. If there are enough bytes that can be displayed as text, then it is gathered and reported. The reported data has a high likelihood of containing meaningless data, but the technique also gather strings that the authors has used in the sample to achieve his actions.

Next on the pyramid is Dynamic analysis. The pyramid means basic dynamic analysis and is almost the same process that the sandbox has on its sample. The analyst manually executes the sample in a safe environment that has tools which gather observations from the system. The analyst then must manually go through the logs and make a up their own opinion of what the sample has done and how much of a threat it is. The process requires some knowledge about creating a safe environment with logging and observations tools, and how to read and analyze the given reports.

At the top at the pyramid code reversing reign supreme. Code reversing is more than just one thing, it is both advanced static and advanced dynamic analysis, in its own way. When doing code reversing you manually read and review the underlying code within the sample. It does not matter what type of file it is; it is just more difficult to understand certain file types. When code reviewing PE files you must first transform the binary data to a human readable format though various tool. This process is called disassembly and gives human readable representation of CPU instructions called Assembly. When the analyst has the static assembly code, they can manually go review it and create an understanding of how the sample function and what it can do. This static process can be supplemented with the dynamic process of debugging. Debugging is when the sample is controlled by a "debugger" that can stop and run the sample at will, and it has full control and visibility into the samples usage of memory and instructions executed. When combining these techniques with a good understanding of assembly and how operating systems work, you can reverse any sample and figure out what it does.

2.2 Portable Executable

In Delimitations we disclosed that we are focusing of Portable Execution (PE) [13] files that either are 32-bit or 64-bit. A PE file is an executable file that execute code on the Windows system, often known as a "exe-file". A Windows PE file can have multiple different file endings, but the most common hare ".exe" and ".dll". What we mean by "32-bit" or "64.bit" is how the file handles memory, and what type of underlying code it has. The Windows operating system also comes in 32-bit or 64-bit mode, and a 32-bit Windows can only handle 32-bit PE files while 64-bit Windows can handle both.

The PE file's purpose is to do actions on the system. This could be simple things like creating new files or compute complex mathematical formulas. To be able to access machine hardware like the hard drive or network gateways, the sample must employ the Windows native functions. A PE file can expand its own functionality with accompanying libraries that are also PE files, or be self-sufficient with all of its code in itself.

The underlying code in a PE file is known as machine code and cannot be understood by humans without the help of tools known as disassemblers. The machine code is a translation of a "human" format such as C or C# and is called compiled code. [14] When you try to revert back to human-readable code you disassemble it and receive Assembly code. Assembly code are short text word representation of the binary instructions and is not a copy of the written code that the author compiled.

The PE header is the first section of the file and contain a myriad of information about the file that the Windows operating systems use to be able to execute the file. It contains, but not limited to: Which Windows functions are used, what functions the PE file exports and other can use, what type of file it is and if it is 32-bit or 64-bit, the different sections of the file, where in the file to start executing code, and how to allocate space in memory and load the file.

Sections are where data is stored, be it code or variables. The sections can have permission that tell the operating system who has access to either write to it in memory or execute the data stored there as code. These permissions are strict and they are there so code cannot run outside of its own scope, and to limit malicious programs capabilities of hiding or destroying.

3 Method

In this chapter part we will explain how we are going to answer the defined questions of this master thesis. The project is split into different parts and phases, where there are different focus areas. Having a structured plan for execution of the project is important. Without a plan it is possible to miss crucial parts or glance over topics that should be given more time. It is also healthy to have time to think about and process findings over time so that you can find every nuance.

The in the first part we will contact and interview professionals that know malware analysis and reverse engineering and have experience with creating educational content or trainings. The interview objects will be from universities and companies, so that we get a broader specter of views of what is educational friendly malware. Next, we will dedicate time for theory research. Here we will go through resources that cover the topic of malware analysis and reverse engineering. These can be books, blogs or videos. We will also do our own reversing and use our accumulated experienced with reversing and education to find examples of theory in practice. We will also present what we believe to be indicators of educationally friendly malware.

When we have a good foundation of theory and interview insights, we will begin experimenting with different tools, and analyze usability within the product. With the experimentation we will begin writing code that implement the different tools that have shown promise and begin on the structure of the framework that is the end goal of the project. After we have a working framework that gives an output that we are satisfied with, we will begin the finalizing the report, and conclude our findings.

3.1 Interviews

To get a better understanding of what problems other educators within the field has, we conducted an interview with a select set of professionals and expert. In this section we will go through some of the answers to identify areas from improvement. The interviews topic was malware analysis education, and educational friendly malware. The interview objects will remain anonymous but will be denoted by the type of work they did: educator or company. Those with company affiliation had internal education and training responsibilities, while educators had more traditional education like lectures. Everyone that were interviewed had extensive knowledge about malware analysis and reverse engineering, as well as educational experience.

The interview was semi-structured where we have a set of base question we asked, but we encouraged tangents and gave ourselves room to ask follow-up questions where needed. We conducted the interview remotely over the Internet with audio and video, and we recorded the audio for later transcription. From the transcription we selected interesting and important answers and are using them as quotes later in this chapter.

When asked about if they used old and previously used sample sets for education, they all said that they most often kept the old. One company employee specified that they looked for new samples from time to time. We followed up with how the different interview objects found potential new samples for education. One company employee said, "Randomly searching through samples that I've already analyzed to find something appropriate." An educator said "Normally the first step of finding samples is through research. Next I look at how easily explainable they are". We then asked if they did this manually or if they had some automated process to help them find what they were looking for. "The only part automated is finding malware […] All other analysis is manual work" said one educator. "We mostly look at targeted attacks, so the analysis has to be manual to gather the right information." said a company employee.

An important part of the project is helping educators reduce time when creating malware analysis trainings. "It varies with complexity" said a company employee when ask about how much time they used to analyze and write reference notes for one sample. "If it is simple, it could be like an hour, and more complex samples might take a day". An educator said: "It might take a month or so to make the course content around it, making slides and writing notes. I am quite careful when choosing samples because it takes so much time." The interview objects all agreed that finding interesting samples and writing course content around it takes time.

We then ask about what made malware a no-go in an educational setting, if there were any signs or functionality that they stayed clear from. "Don't use many different ransomware samples because they often do the same thing", said an educator. A company employee noted; "Malware with network addresses that are live are no-go. Also, malware with destructive capabilities, such as deleting files or destroying the operating system." Another company employee said, "Malware protected by commercial packers and crypto would be too complex to analyze. Or implemented VM-protections or simulation."

3.2 Educational Friendly malware

In this chapter we will try to compile answers from different sources to try and determine; "What makes a malware sample educationally friendly?". We want to answer this question so that we can make a framework that can find and highlight the right indicators. Since we have specified that we want to work statically, we will not focus on finding that can only be obtained through dynamic analysis such as API logging. We will also utilize the authors knowledge within the field to pick out and specify indicators that we believe to be good indicators to find promising samples.

When wanting to find out makes malware educationally friendly, it makes sense to look at educational content. There are many sources for malware analysis education, but there is one book that is considered the best book for learning reverse engineering and malware analysis: Practical Malware Analysis (PMA). [1] The book covers topic such as basic static and dynamic analysis, understanding assembly and onto more advanced subjects such as anti-reversing and anti-analysis functionality. Chapter 7 "Analyzing Malicious Windows Programs", and later chapters in PMA, are of interest for us. It begins by highlighting file API calls as interesting calls to look for. "Find file"-related function can tell the analyst about some functionality of the malware sample, such as persistence or if it modifies existing files on the system. Next the book talks about registry related actions, which can be used and abused by the malware author for persistence, exploitation or circumventing anti-virus software. Examples of often used registry keys used by "simple" malware are the "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" or

"HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\RunOnce". Entries here are automatically executed at startup and is a common and easy to find persistence mechanism. The registry is used for many different things by the operation system and installed programs. There are therefore difficult to list every possible interesting registry key, and it is up to the education to find interesting keys and do research around them. It is however evident that highlighting registry actions should be a part of what makes a malware educationally friendly.

The book is quite thorough when going through network related actions malware can do. Network related activity is often crucial for malware to be able to achieve its overarching goal, be it remote access, receiving additional functionality or sending encryption keys. It would therefore be important to focus on network related calls and functionality when trying to find malware samples for education. All the network related API calls should be deemed interesting and investigated further. It is a common tactic to import low level network functions form the ws2_32.dll library and use the ordinal number instead of function name to obfuscate and hide functionality. These are just some of the interesting things one can look for when analyzing a malware sample. With network activity there is often strings associated with the different techniques used to connect to the Internet. There are different ways to connect, some require more verbose data than other. For instance, using ws2_32.dll creates sockets that does not need strings, but other high level API calls need user-agent strings, HTTP methods like "GET" and "POST" [15], which can be quite telling that the sample has internet activity.

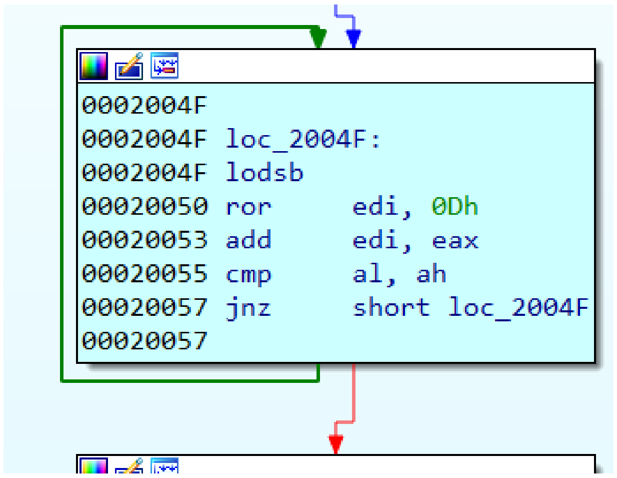
From the PMA-book, we have already found and highlighted multiple interesting stings and functions that we can use. These findings should alone be able to find some samples, but we will continue to find other useful indicators.

3.2.1 Import hiding

Then doing static analysis of a sample one of the first things an analyst might to is to look at the imported functions of the file. We do this so that we can get a "feel" for the malwares imported capabilities, because imported functions are crucial for any running program to be able to do actions on the system or use underlying hardware. Statically looking at the import table of a file has its problems; programmers might import whole libraries when they only use a few functions from it, or even none! The import table might therefore give a wrong image of what the sample does. When analyzing functionality related to imported functions, one should find and map out functions used in the code. This gives a much better view and understanding of the sample's functions and capabilities. There are some however some problems with this technique as well. To be able to analyze which imported functions are used, you first must decompile the binary code and do analysis on the rendered assembly code. This is much more resource heavy and require a complex tool, or more time of the analyst. The malware author can also implement techniques to hide which imported functions it uses, and it makes it much more difficult for the analyst. [16]

One method to dynamically link imports at runtime is to have list of strings, or more commonly a list of encrypted strings, that are then used with the API call "GetProcAddress". What this API function does is that based on the string given, and library handle, it returns an address that function. This address can then be saved to a variable and later be used as a normal call. If the strings are encrypted, it would be difficult to map out which functions the sample used without advanced static analysis. If the "GetProcAddress" is not hidden, then we can use it to locate the decoding and importing routine and use it as an example in a class. It could therefore be interesting to highlight samples with few imports that has the "GetProcAddress" and the "LoadLibraryA" imported. This would give good indications that the sample has dynamic importing functionality.

Another method that hides which function the sample imports by iterating over the export tables of loaded modules. The sample in this case has a multiple initiated modules or libraries, but does not look like the sample actively use them. The sample can have implemented the technique that finds the wanted function based upon a string-generated value.



```
0002004F
0002004F loc_2004F:
0002004F lodsb
00020050 ror     edi, 0Dh
00020053 add     edi, eax
00020055 cmp     al, ah
00020057 jnz     short loc_2004F
00020057
```

Figure 2: Simple example of generating hex value from as string.

The samples find the export table of a module and iterates over the function names as strings. The string can be modified many ways, but a criterion is that the resulting hex value must be the same each time. When it has created a value, also known as a hash value, it tests it against a hard coded value to see if it matches. If they are different, the subroutine starts again with the next string. When the generated hash and the coded hash match the sample remembers the offset and returns the address for the desired function. This address can then be saved to a variable and used as an ordinary API call. This method can be very tricky to detect and analyze. An analyst might have to debug the sample to be sure which functions are resolved. It should however be possible to recreate the hashing algorithm and create your own list and find matches manually without running.

These two techniques are often used by malware and can indicate a malicious program. They are however not too complex techniques, and show be highlighted and used for educational purposes.

3.2.2 Calling system functions

As noted earlier, malware might hide their import so that the analyst cannot easily gleam which system API calls the sample does. This is because it would reveal too much about the malware, what it tries to do and its capabilities. So, this would mean that malware samples with clearly visible API calls could be easier to analyze. It would not be enough to just list out the imported function list, because it can be filled with functions not used by the sample. You must find the calls used in the code to map out which functions are used, and you must disassemble the code to be able to do that. Once that is done, it should be easy to search through and list out actual called API functions and their locations.

All API calls that a suspected malware sample do should be of some interest of the analyst. There is such a large landscape of calls a program can do to the Windows API so only including some function would potentially make us miss interesting calls and techniques. We can however create a list of function we know malware uses for nefarious activities (Table 1). We especially want to investigate calls directly to the low-level library ntdll.dll from user written code. [17] [1] [18] [19]

Table 1: API often seen with malware. Removed extensions such as "A", "W", "Ex".

Function name	**Library**
CreateFile	Kernel32.dll
CreatePipe	Kernel32.dll
CreateNamedPipe	Kernel32.dll
OpenFile	Kernel32.dll
OpenFileMapping	Kernel32.dll
WriteFile	Kernel32.dll
WriteConsole	Kernel32.dll
FindFirstFile	Kernel32.dll
FindNextFile	Kernel32.dll
ReadFile	Kernel32.dll
ReadConsole	Kernel32.dll
SetFileAttributes	Kernel32.dll
SetConsoleMode	Kernel32.dll
LoadLibrary	Kernel32.dll
RegOpenKey	Kernel32.dll
RegQueryValue	Kernel32.dll
RegSetValue	Kernel32.dll
CreateProcess	Kernel32.dll
CreateProcessInternal	Kernel32.dll
ShellExecute	Shell32.dll
WinExec	Kernel32.dll
GetCurrentThreadId	Kernel32.dll
ReadProcessMemory	Kernel32.dll
WriteProcessMemory	Kernel32.dll
VirtualAlloc	Kernel32.dll
GetThreadContext	Kernel32.dll
SetThreadContext	Kernel32.dll
SetProcessAffinityMask	Kernel32.dll
TerminateProcess	Kernel32.dll
ExitProcess	Kernel32.dll
SetWindowsHook	User32.dll

CallNextHook	User32.dll
IsDebuggerPresent	Kernel32.dll
OutputDebugString	Kernel32.dll
CreateMutex	Kernel32.dll
CreateSemaphore	Kernel32.dll
DeviceIoControl	Kernel32.dll
GetDriveType	Kernel32.dll
Send	Ws2_32.dll
Recv	Ws2_32.dll
WSARecv	Ws2_32.dll
Connect	Ws2_32.dll
Gethostbyname	Ws2_32.dll
InternetGetConnectedState	Wininet.dll
InternetOpenUrl	Wininet.dll
InternetReadFile	Wininet.dll
InternetWriteFile	Wininet.dll
TransactNamedPipe	Kernel32.dll
CreateService	Advapi32.dll
StartService	Advapi32.dll
ChangeServiceConfig	Advapi32.dll
GetSystemDirectory	Kernel32.dll
GetSystemTime	Kernel32.dll
NtOpenProcess	Ntdll.dll
NtCreateSection	Ntdll.dll
NtMapViewOfSection	Ntdll.dll
NtOpenThread	Ntdll.dll
NtSuspendThread	Ntdll.dll
NtGetThreadContext	Ntdll.dll
NetSetThreadContext	Ntdll.dll
NtResumeThread	Ntdll.dll
NetReadVirtualMemory	Ntdll.dll
NtWriteVirtualMemory	Ntdll.dll
NtProtectVirtualMemory	Ntdll.dll
GetMessage	User32.dll
PeekMessage	User32.dll
SendMessage	User32.dll
GetClipboardData	User32.dll
GetAsyncKeyState	User32.dll
GetForeGroundWindow	User32.dll
GetKeyState	User32.dll
URLDownloadtoFile	Urlmon.dll
GetProcAddress	Kernel32.dll
GetModuleHandle	Kernel32.dll
CreateRemoteThread	Kernel32.dll

AdjustTokenPrivileges	Advapi32.dll
OpenProcessToken	Advapi32.dll
LookupPrivilegeValue	Advapi32.dll
IsDebuggerPresent	Kernel32.dll
CryptAcquireContext	Advapi32.dll
CryptReleaseContext	Advapi32.dll
CryptCreateHash	Advapi32.dll
CryptHashData	Advapi32.dll
CryptDeriveKey	Advapi32.dll
CryptDecrypt	Advapi32.dll
CryptEncrypt	Advapi32.dll
CryptDestroyKey	Advapi32.dll
CryptDestroyHash	Advapi32.dll

The function names in Table 1 can be used to search through the samples import table for matches, or in the extracted list of used function within the .text section of the file. The list of functions can all be used in some way by malicious program to achieve an objective. They themselves are not dangerous, but they can be used with malicious intent, or to detect analysis, prolog analysis and other ways to remain undetected for as long as possible.

3.2.3 Process creation and threating

Many programs both malicious and benign use process creating and threating to multitask activities on the system. In malware it can be used to hide or obfuscate which functions are used, and even make debugging the sample quite tricky if different threads rely on each other. Process creation can not only start processes, but also start, for instance, cmd.exe with parameters so that commands are executed on the system. Finding samples that has process creation capabilities can be useful for the educator, because process creation leaves traces on the system that can be viewed by the student. It is often an early exercise to dynamically run and log the process creation three, and therefore finding samples that does this is helpful. Tracking process creation can also tell something about the purpose of the sample. For instance, if you see string reference to "vssadmin.exe Delete Shadows" and calls to the kernel32.CreateProcessA you can assume that this sample has destructive capabilities such as file encryption or destruction. [20] An educator can then filter out such destructive capabilities if they so choose.

Threading on the other hand is much sneakier than process creation. It leaves less traces of the execution and can be difficult to track in a debugger. A thread can be started in the parent process, but it is also possible to start a thread in another process. Both these methods are used to hide malicious code from an analyst or antivirus program. Finding and sorting samples by threading can be quite useful for an educator, because they can decide self if it is too advanced or withing scope of the class. Analyzing threaded malware is a skill that should be taught.

Services and scheduled tasks are also ways to start new processes, but the malware does it indirectly. It is also more of a persistence and hiding technique, but the result is the same; the malware can run any program or command on the system using services or scheduled tasks. These techniques also leave evidence of itself on the system and is therefore used early when doing dynamic analysis.

What both process creation and threating can do is being created in suspended mode. This allows the process to modify and edit the data inside of the process or thread before resuming it so it can execute its new functions. This is called process hollowing or process injection [21], and is often a common example on more advanced malware functionality to evade detection.

3.2.4 Interesting instructions

We have seen that we can find potentially interesting and educationally friendly malware by searching though the disassembled coded for API calls. We can also look for interesting instructions the same way. There is a large set of possible instructions that can be used, but there is a subset of uncommon instructions that malware might use to achieve an action. These instructions can be in benign software, but as we have stated earlier; We only want to find examples of usage so that student can study the use and understand how it is done. Some instruction might not be applicable to all operating system, but this of no concern for us when looking for them. Since instructions revolves around the use of many different instructions, we will further down use the "reg" keyword as an umbrella term for any registry space. We might use reg1 and reg2 in the same instruction to underline that they must be different registry.

Table 2: List of potential interesting assembly instructions.

Opcode	**Instructions**	**Description / Usage**
0xE8	call reg	Calls the address in a registry. Hide function called.
0xFF	call [reg+offset]	Call function based on offset and variables. Hide function call, indicate recreated IAT.
0x31	xor reg1, reg2	Xor two different values. Xor same value is not interesting.
0xF2 0xA6	repne cmpsb	Compare value in buffer, detects changes (breakpoints, patches).
0xF2 0xA4	repne movsb	Move value from ESI to EDI, detect code move, string move.
0xF2 0xAF	repne scasb	Scan the buffer for a value e.g., 0xCC (breakpoint).
0xEB 0xFF	jmp -1	Jump to itself, infinite loop, potential anti-analysis.
	push reg retn	Return pointer abuse, many ways of doing this.
0x68 0x33 0xCA	push 33h retf	Switches the CPU to interpret code as 64-bit code.

0x64 0xA1 0x30	mov reg, large fs:30h	Get access to the PEB, might be for anti-debug or manually getting IAT.
0x60 … 0x61	pushad … popad	Often used in decoding routines, needs instruction in between.
0x0F 0x31 … 0x0F 0x31	rdtsc … rdtsc	timing check, need instructions in between. Might give false positive.
0xB8 0x68 0x58 0x4D 0x56 0xB9 0x0A 0xBA 0x58 0x56 0xED	mov eax, 564D5868h mov ecx, 0Ah mov edx, 5648h in eax, dx	Using the in instruction to detect VMWare technology.
0x0F 0xA2	cupid	Get information about CPU, can detect VM.
0x0F 0x01	sidt	Old anti-vm technique; detection
0x0F 0x01	sgdt	Old anti-vm technique; detection
0x0F 0x00	str	Old anti-vm technique; detection
0x0F 0x00	sldt	Old anti-vm technique; detection
0x0F 0x01	smsw	Old anti-vm technique; detection
0x0F 0x05	syscall	Invoking kernel call
0x0F 0x3F 0x07 0x0B	vpcext 7, 0Bh	Instruction specifically for virtual PC, used for VM detection. [22]
0xCC	int 3	Interrupt, software breakpoint hardcoded.
0xCD	int [value]	Interrupt.

The list of instruction is not extensive, and there are most certainly other assembly instructions that can indicate malicious activities. The list is compiled so that we have a basis of what we are looking for, so that we can produce a system that can help us find these interesting instructions. Some of these instructions are for detection techniques that rely on older hardware, and how they worked with virtualization. They can still be interesting to detect so that we can have new examples for old techniques.

It would be difficult to create a system that can 100% detect malicious instruction use. One should expect to gather many false positives when using an instruction list to find interesting segments of code. It is therefore important to verify the findings so that you are certain that you have matched on code that you would want to use in educational setting.

3.2.5 Interesting strings

Strings are a natural place to start for any malware analyst. Strings can give a good baseline to start creating hypothesis about the sample, which you later can verify or throw away. The most interesting strings are the most difficult to list out; they are URLs and IP-addresses. URLs and IP-addresses are easy for humans to identify, but it is resource intensive to have an automatic system finding them for us. This is because URLs can vary greatly in length, usage of special characters and separator. IP-addresses however have many constraints what must be taken into consideration to be a valid address, and we are not interested in invalid "addresses". There are tools such as regular expression, which generates a matching machine against a special matching algorithm. [23] There are two problems with using regular expression to find URLs or IP-addresses; The matching rule is difficult to create and make accurate and since it is a complex rule it is going to be resource intensive. We can solve the first problem by using other's solutions to this problem. There are multiple tools or resources that have tried to create a good matching rule for URLs or IP-addresses, such as the toolbox "CyberChef" [24] or Didier Stevens' tool "re-search.py" [25]. Both tools can be used, although CyberChef is UI-based, but we can extract the regular expressions we need to use ourselves from the tools.

Network related strings as URLs and IP-addresses are very interesting, but we can list out other static strings that also can be of interest for an analyst or educator. They can be network related, host related or strings that indicate some sort of functionality. Strings can be found anywhere in the sample, but strings that are confirmed used in the code are most interesting. This is because samples can be compiled with libraries within themselves. This creates a surplus of benign strings that are not directly relevant for the malware's functionality. If we can detect strings created at runtime such as stack-strings that would be extra interesting. Table 3 is our compiled list of strings we deem interesting. Some of the strings in the list are not absolute and can be part of a longer string or the casing may all be upper or lower characters. The strings could also be split or obfuscated in other ways and more difficult to locate.

Table 3: List of interesting strings

String	Comment / Description
User-Agent	Network activity
Mozilla	Network activity, most common word in user-agent.
HTTP	Network activity
POST	Network activity, sending information.
GET	Network activity, receiving information.
cmd.exe	Shell, execution.
powershell.exe	Shell, execution.
wmic.exe	Shell, execution.
vssadmin	Deleting backup
shadow	Hits both vssadmin and wmic deleting
rundll32	Execution.
HKLM	Registry activity.
HKCU	Registry activity.

SOFTWARE	Likely registry activity.
CurrentVersion	Likely registry activity; persistence.
User Shell Folder	Likely registry activity; persistence.
Explorer\\Shell Folder	Likely registry activity; persistence.
RunServices	Likely registry activity; persistence.
Policies\\Explorer\\Run	Likely registry activity; persistence.
BootExecute	Likely registry activity; persistence.
Winlogon	Likely registry activity; persistence.
SELECT	Likely SQL query.
WH_KEYBOARD_LL	Keylogger constant.
WH_KEYBOARD	Keylogger constant.
shift	Keylogger logging.
ctrl	Keylogger logging.
esc	Keylogger logging.
backspace	Keylogger logging.
enter	Keylogger logging.
Zone.Identifier	Potential hiding downloaded file.
Your files	Likely cryptolocker.
encrypted	Likely cryptolocker.
payment	Likely cryptolocker.
.lock	Likely cryptolocker.
.enc	Likely cryptolocker.
.exe	Possible file extension WL/BL
.dll	Possible file extension WL/BL
.doc	Possible file extension WL/BL
.xls	Possible file extension WL/BL
.pdf	Possible file extension WL/BL
sample.exe	Possible detection of analysis.
malware.exe	Possible detection of analysis.
sandbox.exe	Possible detection of analysis.
myapp.exe	Possible detection of analysis.
vbox	Detection of virtualization.
vmware	Detection of virtualization.
vmx	Detection of virtualization.
ollydbg	Detection of analysis tool.
processhacker	Detection of analysis tool.
procmon	Detection of analysis tool.
procexp	Detection of analysis tool.
ida	Detection of analysis tool.
immunitydebugger	Detection of analysis tool.
wireshark	Detection of analysis tool.
hookexplorer	Detection of analysis tool.
importrec	Detection of analysis tool.
windbg	Detection of analysis tool.

joebox	Detection of analysis tool.
x32dbg	Detection of analysis tool.
x64dbg	Detection of analysis tool.
resourcehacker	Detection of analysis tool.
fakenet	Detection of analysis tool.

The list is in no way extensive and might not cover a fraction of what can be classified as interesting strings. It will be ongoing work to fill out and add new interesting strings so that we can achieve an as interesting sample selection as possible.

3.3 The framework

We have up until now focused on what other people believe to be educationally friendly malware, and we have highlighted some indicators that we believe can be used to find these. This is all well and good, but we will now try to create a framework that can take the findings we have brought and create usable output. In this chapter we will go through our thought process when picking tools, processes or techniques to create our framework.

To be able to create a framework that can take our input and create a usable output, we must define what we want to do with a sample. Our end goal is to take a randomly large set of samples and automatically analyze these. The analysis will then generate a report that an educator or malware analyst can read to find potential samples for further verification and ultimately be used to further knowledge and understanding. The analysis will be static so that we do not infect our environment with countless different malware samples. It would be possible to do this dynamic, but it would require more resources and more allotted time to each sample. There are two types of static analysis: Basic static analysis and advanced static analysis. [1]

3.3.1 Basic static analysis

Basic static analysis is collection of information from the sample without executing it. The information collection can often be categorized as "low hanging fruits", as in it is simple to gather them. There can be many different steps in a basic static analysis, and different analysts might have their own ideas of what is incorporated in the basic analysis. The book "Practical Malware Analysis" highlight a set of steps to do basic static analysis; Running antivirus, fingerprinting and hashing of the file to find previous analysis, extracting strings, detecting packed files and packers, analysis of imported or exported functions, and header and section analysis. [1]

Scanning the given sample with an antivirus engine can give valuable detection information and can often classify and give a family name to the file. This is useful to search for if you want a specific malware-family sample. There are however some problems with this method of gathering information. The first one is that you must be certain that the engine does not remove any files it deems malicious or dangerous. The next problem is that there is often a mismatch of family-name between the different antivirus companies. [26] This can create a problem where the educator or analyst can find the family they are after because the engine has classified it under another name that they do not know. The third possible problem is that if the engine is free or open source then the detection engine might have an older ruleset, and wrongly label samples or not at all!

Fingerprinting is creating some unique value based upon data from the sample. The most common one is to create MD5, SHA1 or SHA256 hashes from the whole file. These hashes can be used to find previously done analysis on the sample. It could be from large entities as VirusTotal.com, smaller blogs, free sandboxes or open malware databases. The value given from previously done analysis is more prevalent when doing manual verification and analysis and might be too difficult to integrate into an automatic framework. There are tools that create hashes based upon import tables that can be used to link different malware samples together, but this is out of this scope. In an automatic framework, hashing would mainly be to uniquely identify each sample so that we do not analyze duplicate files and have a searchable database to find earlier executed analysis.

String extraction is a common early step in malware analysis. In some cases, we can gleam the samples functionality and create hypothesis of what it is supposed to do. The most common way of extracting strings is to analyze the sample byte by byte, and check if they are withing ascii or Unicode range. [27] If there are enough "characters" after one another and above a given threshold then it is categorized as a string and reported. This is a simplistic way of extracting strings but can yield great results on unpacked malware. There are more sophisticated versions of strings that do more than simple scan, such as FLOSS. [28] FLOSS' different algorithms can filter out and recompile strings generated at runtime though "stack strings" [1] [29] and some decryption routines. [28] [30]

When we want to find interesting samples for education, we want to know it the sample is packed or has some sort of protection. This is because we often want sample that does not have this layer of complexity, unless it is an exercise goal to overcome it. There is a theory on generic detection of packers based upon Shannon entropy of the binary file. [31] In short, if a section of the file or the file itself has an entropy of 6 or above, then it is a high change it is packed. We can therefore use entropy tools to detect packed samples. We can also use tools that automatically detect packers such as PEiD [32] or the Yara [33] rule set that detect packers. [34] If we use both entropy and tools, we will have a higher change of classifying samples as packed correctly.

Any Windows executable need to import functions to be able to do actions on the system. These imported functions give us information about what the file can do "out of the box". Packed or obfuscated malware will often have few imports to hide its capabilities and resolve the functions at runtime. [35] This is something we cannot do anything about at this stage, but the absence of evidence is not evidence of absence, and we can use evidence of few imports as a categorizer. We can also categorize based upon which library is imported and functions imported. Imported library such as ws2_32.dll would indicate strongly that the sample has network capabilities, but imported Advapi.dll has many functionalities and does not indicate anything other than advanced functionality is imported. It would also be possible to categorize sample based upon exported functions. If a ".exe" file has multiple exported functions it is an abnormality and should be highlighted. Also, if a ".dll" file has few exported functions, then it might only contain malicious functionality and not work as a common library file.

The easiest thing to analyze in the PE header would be section names and permission. The most common section names are: ".text", ".rdata", ".data", ".idata", ".edata", ".pdata", ".rsrc", and ".reloc". If a sample has a section name other than these, it should be highlighted. The permissions to write and execute a section are strict and are often restricted specific sections. [13]

Table 4: List of sections and permissions

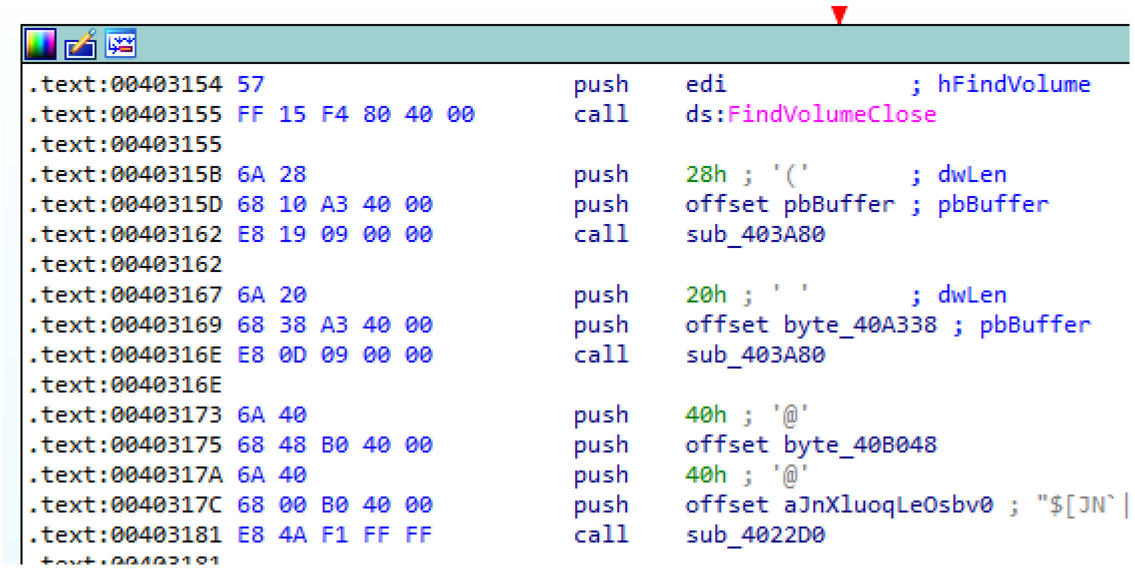
Section name	**Permissions it should not have**	**Permission it should have**
.text	Write	Execute, read
.rdata	Write, execute	Read
.data	Execute	Read, write
.edata	Write, execute	Read
.idata	Write, execute	Read
.pdata	Write, execute	Read
.bss	Write, execute	Read
.rsrc	Write, execute	Read
.reloc	Write, execute	Read

There is also another indicator that can be interesting for analysts and educators, and that is indicators of size different between section on disk and section in memory. The PE files sections has a "size of raw data"-value and a "virtual size"-value set in the header. If the "size of raw data" is much smaller than the "virtual size", then we have an indicator of packer or other moving and writing of data in memory at execution. For instance, if .text size of raw data is 0, and the virtual size is 700h then we can assume that executable code is written to the .text section at runtime. There can however be discrepancy between the raw size and virtual size of the ".data" section, as it is populated with variables when the program is executed. Finding these differences in section size can be interesting for an educator to highlight malware functionality that write malicious code at runtime to avoid static analysis.

There is a special section called ".tls" which is the "tread location storage"-section and also known as TLS Callbacks. This section is uncommon in most sample and is used to execute code before the entry-point of the PE-file. Malware often use this to hide its malicious code and activity so that it can be executed without being analyzed. [1] This is a capability that should be highlighted if found within a sample.

3.3.2 Advanced static analysis

Advanced static analysis can be what differentiate a security analyst from a malware reverser. It is when you disassemble the binary machine codes so that you have a human readable representation. The human readable format is Assembly if the compiled sample is C or C++. The analysis of the assembly code can be tedious and difficult and require deep knowledge of the language and the inner workings of the computer.



```
.text:00403154 57                push    edi             ; hFindVolume
.text:00403155 FF 15 F4 80 40 00 call    ds:FindVolumeClose
.text:00403155
.text:0040315B 6A 28             push    28h ; '('       ; dwLen
.text:0040315D 68 10 A3 40 00    push    offset pbBuffer ; pbBuffer
.text:00403162 E8 19 09 00 00    call    sub_403A80
.text:00403162
.text:00403167 6A 20             push    20h ; ' '       ; dwLen
.text:00403169 68 38 A3 40 00    push    offset byte_40A338 ; pbBuffer
.text:0040316E E8 0D 09 00 00    call    sub_403A80
.text:0040316E
.text:00403173 6A 40             push    40h ; '@'
.text:00403175 68 48 B0 40 00    push    offset byte_40B048
.text:0040317A 6A 40             push    40h ; '@'
.text:0040317C 68 00 B0 40 00    push    offset aJnXluoqLeOsbv0 ; "$[JN`|
.text:00403181 E8 4A F1 FF FF    call    sub_4022D0
```

Figure 3: IDAPro disassembled code

The analysis process has historically been a manual job, but in later years there has been an increase in tools helping advanced static analysis. There are more tools that rely on dynamic analysis and debugging to populate databases with gathered information, such as Talos' Dynamic Data Resolver [36], than tools relying wholly on static analysis. The FireEye company has released a tool that does advanced static analysis in an automated fashion. The tool, CAPA [9], analyze a sample for its capabilities and report back what it found and where in the code it was. The tool disassembles the sample into assembly code and generates code blocks and code flows as a normal disassembler. The disassembled "view" of the code is then passed to a smart pattern matching algorithm that can recognize and handle these blocks and flows and gives accurate and complex matching. The matching algorithm can filter out unnecessary code and based on the rule given follow the code flow. When it has found a match on a rule, it saves the location offset in the file for later logging. These offsets and detections can then later be verified by an analyst and determined accuracy and how interesting they are.

The CAPA tool is powerful and can help us tremendously when analyzing files for interesting indicators and how educationally friendly they are. The tools have a python module that can be integrated with the analysis framework, and its output can be parsed and used against our lists of interesting strings and function calls.

3.4 Building the framework

So now that we know understand what we can look for when doing basic and advanced static analysis, we will try to create the framework that can take an unknown set of malwares and filter out potentially educationally friendly samples. The new set of samples must have a form of report that can be used to verify the findings and can be search though to find samples that match a given criteria.

3.4.1 Malware detection

When analyzing malware that you are unfamiliar with you can get a lot of information “for free” by searching for the file’s hash on antivirus databases. There are organizations and private persons that upload new samples to different databases with accompanying report or analysis. These reports can include information that is interesting for an educator, such as how the sample is tagged. Automatically searching for a sample-match on online database could require registration or a payment to receive the desired information.

There is one large antivirus database that many antivirus companies are connected to: VirusTotal. [37] Here samples are manually or automatically uploaded for analysis by 65 (per writing date) different antivirus engines. If an antivirus recognizes the file as malicious, the report will display that antivirus company’s name for the malware, which you can use to search for already written reports. VirusTotal gives its users access to a web API where you can search and submit samples. The free version has 4 requests per minute to a total of 500 per day, and a paid version where cost determine the limits. [38]

Another database of malware samples that is searchable with an API is MalwareBazaar from abuse.ch. [39] MalwareBazaar is not a multi-antivirus scanning platform, but a community driven platform that upload confirmed malware samples and tag them accordingly. Once uploaded the samples are sent to different free or partnered platforms for analysis. The platform are sandboxes and unpackers and give a malicious score or malware family name for the sample. The service is free, to both download samples and query for information.

Yet another way to gather information about the file is to scan it with a set of virus-classification rules. These rules can be based upon an anti-virus engine ruleset and give inside into what type of malware family it might belong. We will not use a common anti-virus engine in our framework as we have not found any written in python that satisfy our criteria. We will however use YARA’s own malware-ruleset to try and identity unknown samples against a known sample family or functionality. By using YARA, we also mitigate the problem of an anti-virus engine uploading the sample to the cloud or deleting it from disk.

Through YARA we are matching the samples against the PEiD-ruleset. PEiD has a large set of rules that detect different aspect about the sample file, such as how it is compiled and if there are any indicators of know packers. Using PEiD we can filter out unwanted PE-files such as “Delphi” or Visual Basic or any other none-C to assembly compiled samples. These are unwanted because they contain a lot of uninteresting support code that fill up our analysis with hits on code that is not malicious. When we have filtered out samples, we can use the results from PEiD in the report so that educators can find samples that are packed or compiled in specific ways.

3.4.2 PE-header analysis

As we highlighter earlier in this paper, the PE-file contains all of the information the operating system needs to be able to execute the file properly. This header metadata can also be analyzed by us to extract important information. This information can then be used to create an insight into the file’s capabilities and usefulness in an educational setting. There are multiple indicators that we are looking for in the header. There are probably more indicators that we have not considered using, but those we have included are those we believe are of most use when finding educationally friendly malware.

The first information we need from the header is what kind of file do we have; is it an exe or a dll? Is it a 32-bit or 64-bit sample? These two points are not directly indicators of malware and malware functionality, but it is necessary for an educator to be able to find and sort out sample based on these criteria. Next are which imports are default resolved when running the file, and if and what exports there are. Malware often dynamically resolves import which do not show in the default import table, but there is data to gather none the less. If there are few imports containing runtime resolving of Windows API functions, e.g., LoadLibrary and GetProcAddress, then that indicates that this sample hides its function and import hiding is a topic in malware analysis. It is also interesting to note if the sample export functions for others to use. If it is an exe-file that export functions that should be flagged as it is unusual.

Not all imported functions are necessarily interesting for a malware analyst or educator. Many samples both good and bad can have more imported functions than it uses. This can be obfuscation or just bad coding and compiling practice. Through our own analysis and educational resources, we have compiled the list at "Table 1" with potentially interesting imports. We can use this list to filter out only these per file, so the educator has a clearer view if the sample contains interesting functions. The educator might have special interest in knowing if the imported function found are related to ransomware or network, for instance, and should be highlighted.

The other part of the PE header that we want to analyze are the sections of the PE file. The quickest and often used in show-casing malware subterfuge techniques is to check for the Thread Location Storage or ".tls" section. The ".tls" section house the code that is executed before the default entry point of the malware, and it is not common to see TLS sections in ordinary files. Next, we have defined at Table 4 a list of common section names with permissions they should have. It would be easy it iterate over the list of names and permissions of the sample and match them against the names an permissions we have defined. If there are any deviations, then we will flag them as interesting indicators.

There are two more interesting indicators we can look for in the sections; the Shannon Entropy and size difference is raw physical size and virtual size. The Shannon Entropy is a mathematical calculation of randomness and is useful to give an indicator as to if the section contains random bytes. If there is a higher concentration of random bytes than other sections, then we have an indicator that the sections might be packed or encrypted. The second indicator is the size difference between the real (raw) physical size on the disk against the virtual size it has in memory. If there is a size difference where the virtual size is 150% of the raw physical size, then we might have unpacking at runtime in that section. It is almost definitely true if the raw size is 0, and the virtual size is not. The usage of sections is common and is definitely something and educator would want to illustrate with a sample. [40]

3.4.3 Extracted strings

Extracting strings from a malware sample is often one of the first things learn to new students. It is the ultimate tool for getting "low hanging fruits" that can give good insight into the sample. As we have seen earlier, string tools are primitive in nature and captures many strings that are worthless, random characters that by chance were next to each other. As before, we have compiled a list of strings at Table 3, that educational resources and ourselves deem interesting and worthy to look after in a string output. If there are any matches between the extracted list and our compiled list, then that should be flagged and presented.

The tool that we use, FLOSS, has more functionality than just looking for visible characters, it also finds "stack strings" and emulates what it believes to be decode functions to decode strings statically. If there are any strings found this way, then they are interesting in and of itself. They should be flagged and presented, especially if they also match with our compiled list. We normally are not interested in matching found strings against interesting function names, but if they are decoded or form the stack then they should be flagged and highlighted. Those function matches would potentially reveal much about the sample's functionality and purpose.

3.4.4 Finding capabilities

Finding what the sample does without running it or doing manual code review is the most difficult task of the automatic process. A malware author can employ many techniques to obscure or hide what the sample tries to achieve on the system. Being able to accurately analyze the assembly code without running it can in many cases be near impossible. We have noted earlier that the FireEye tool Capa was able to do advanced static analysis of the assembly, and we are going to use this to find interesting sections within the sample.

As before, we have listed at Table 2 multiple of interesting assembly instructions that we believe is worth looking for when doing advanced static analysis. Since the capa tool is an advanced assembly search tool, we could create rules that match the instructions listed. We have however chosen to use the free ruleset that is accompanying the tool. The ruleset is extensive and includes many more rules than we want or need for this framework. We have also parsed the output for useful information such as what rule matched and exact location within the file. This is to simply the view for the educator they can more easily locate where they must go to verify the findings.

Capa also reports on calls to the Windows functions from the samples code. These are interesting as they can show directly which functions are used, where looking at the import address table can show us many functions not used. By looking at these used function calls an educator can get a good understanding of how the program works and what it can do. This gives a good basis to judge if the sample is educationally worthy.

3.4.5 Creating the reports

After the individual analysis is done and we have found all there is to find based upon or criteria, we must save the analysis to a report. There is a lot of information written in each report that comes from alle of the different tools use, and it can be difficult to find the information the educator wants to know. We have therefore created a tagging system that takes key-findings and create a tag that can be searched for more easily. We have previously in this paper hinted at these tags or flags, and we believe these are crucial for the framework to be useful to do simultaneously analysis of multiple files.

We have four sets of tags per analysis: Our own specified tags, matches on specific capa rules, tags on matches from the online database MalwareBazaar, and PEiDs matching results. Our tags behave in a true or false state, so that any tags set to false will not be shown in the report.

We have created our own tags based upon what we believe to be useful highlighting when getting an overview of each sample. These tags have we split into three groups: info, easy and intermediate. Which tag is in easy and which is in intermediate is decided by our own experience as reverse engineers and creators of educational content. Each time an indicator is found that activates a tag, we increment with one an internal counter for that sample. This gives us a count of hits that we later will use to create a "difficulty score".

The tags we have placed in the information category are *exe, dll, 32-bit, 64-bit, exports, few_imports.* These give important information about the file, but does not tell the educator how difficult the file may be. The next set of tags are placed in the "easy"-category and showcase indicator that are easy to find for students or easy to use in lectures. These tags are *interesting_static_strings, interesting_stack_strings, stack_function_names, stack_strings, anti_debugging, anti_analysis_tools, anti_vm, network, keylogger, and anti_av.* There are multiple tags involving "stack strings", but this is because we might be interesting in stack strings for different reasons. If there are interesting stack strings then we have found something that matches in our "interesting list", if we have found function names in the stack strings then we most likely has dynamic resolving of API functions. With anti_analysis_tools we mean that there are found references to tools commonly used with malware analysis and there is a high change that there are some detections of these tools to stop the analyst figuring out the real functionality of the malware.

Lastly, we have the intermediate tag, which house indicator that we believe to be of a more complex character that might need some more experienced students or more analysis to become useful. These tags are *tls, exe_with_exports, packed, unusual_section_name, unusual_section_permission, raw_virtual_size_diff, interesting_decoded_strings, decoded_strings, decoded_function_names, ransomware, anti_disasm, indirect_call and registry.* The "decoded" tags work the same way as "stack strings", but are placed in intermediate because it might encoded in a difficult manner. Sections might not be difficult to work with but require some knowledge to know what to look for and what the different indicators mean. "Registry" can be anything related to the registry-hive, can be uninteresting or out of scope for some courses. The flags or tags in both easy and intermediate only indicate that there might be these functionalities, such as anti-debugging or network activity. It is up to the educator to verify the findings though information in the report or the sample itself.

Table 5: Lists of tags defined

Easy tags	**Intermediate tags**
interesting_static_strings	Tls
Interesting_stack_strings	Exe_with_exprots
Stack_function_names	Packed

Stack_strings	Unusual_section_name
Anti_debugging	Unusual_section_permission
Anti_analysis_tools	raw_virtual_size_diff
Anti_vm	Interesting_decoded_strings
Network	Decoded_strings
Keylogger	Decoded_function_names
Anti_av	Ransomware
	Anti_disasm
	Indirect_call
	Registry

We rely on the tool capa to “fill in the blanks” when it comes to finding sample functionality. We have reviewed the accompanying rule set that capa uses by default and created our own subset of both easy and intermediate capabilities. These subsets are picked out and categorized by us on the same premise as above, our own experience and how different topics are tackled in books and resources. The tags do not include the whole rule name, or what is matched so the educator must go to the individual report to review the findings. The easy tags are *anti_debugging, anti_av, anti_vm, anti_analysis_tools, stack_strings, keylogger, network, shell, execute, xor, bare64, embedded_pe, firewall, desktop_lock, change_wallpaper, cpu_info, mutex, registry, start_service, create_process, persistence.*

There is some overlap between our defined tags, and the tags created for the capa-analysis. This is because we want redundancy in the detection of indicator; if our analysis does not catch it, then the capa-analysis might catch it and vice versa. Each tag can be activated by multiple different rules, but each rule should have something to do with what the tag is named. It is up to the user to go and verify in the individual reports. It is worth noting that the tag “xor” does not mean that we have found one instance of the xor instruction, but that there are two different values being xor’ed, and is an indicator of easy encryption.

We also have our tags of intermediate difficulty capa-rule matches. These are *anti_disasm, packed, rc4, aes, des, rsa, ransomware, create_process_suspended, rwx_memory, create_thread_suspended, destructive, pusha_popa, peb, fs, dynamic_resolved_functions.*

Table 6: Capa tags defined

Easy capa tags	**Intermediate capa tags**
Anti_debugging	Anti_disasm
Anti_av	Packed
Anti_vm	Rc4
Anti_analysis_tools	Aes

Stack_strings	Des
Keylogger	Rsa
Network	Ransomware
Shell	Create_process_suspended
Execute	Destructive
Xor	rwx_memory
Base64	Create_thread_suspended
Embedded_pe	Pusha_popa
Firewall	Peb
Desktop_lock	Fs
Change_wallpaper	Dynamic_resolved_functions
Cpu_info	
Mutex	
Registry	
Start_service	
Create_process	
Persistence	

As you can see there are many more tags from the capa analysis than our own analysis. In the capa tags we have a more emphasis on capabilities such as process creation, encryption, and creating executable memory sections. We also flag if the sample accesses the PEB, which can be interesting for reasons such as manually checking for debugger or resolving imports. The capa also includes tags about encryption such as xor, base64, rc4, aes, des and rsa. These are interesting as many of these can be decoded manually and working with simple encryption is a common topic in malware analysis.

We utilize the open and free malware database MalwareBazaar [39] to look up and get "public" set of tags. These tags can revival more about the file than our static analysis, because MalwareBazaar gather information from dynamic analysis from sandboxes and manual analysis done by the publisher. We do not weight these tags, as the tags can be named things out of our control, and we do not know the accuracy of the tag. We only use them as a handy guidance.

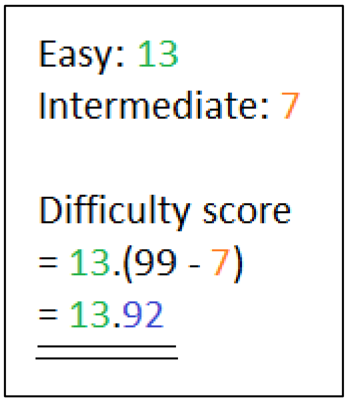
The last set of tags that we create for the sample are the tags given from the PEiD analysis. We do not modify these tags and include everything that is found. But we have created two set of lists of commonly used packers and categorized them into two difficulty levels intermediate and hard. This is the only place that the hard difficulty comes into play, but it is needed as some commercial packers are too difficult to even consider using for educational purposes. We also do not have the easy level, this is because even though it is packed with an easy technique, we have no notion of how difficult it is after we have unpacked it.

The tags for intermediate are *pecompact, themida, upx, aspack, nullsoft.* These packers have unpacking tools or tutorials that help unpacking and retrieving the actual sample. We also have these hard tags: *armadillo, exe_stealth, asprotect, obsidium, execryptor, vmprotect.* These packers are too complex and should not be used in educational purposes unless you want to highlight something with these packers.

There are still many of the PEiD detections that we do not handle, but we include everything in the report so that the user can search and find what they are looking for.

3.4.6 Creating the meta report

When we are done with our different individual reports, we want to create an overarching report that takes into account all of the other reports. We call this the meta report and is where an educator would start to look for potential interesting samples. The meta report does not include everything that the other reports do, but it takes the path of the sample and the four sets of tags that has been generated as well as how many times the different difficulties have been hit. The individual reports shown in the meta report are sorted first by how many easy hits it has then how many intermediate hits it has. These numbers are two independent numbers originally, but we merge them into a decimal number to be able to easily short them. We take the "easy number" and place in in the whole numbers, and the intermediate as the decimal numbers but "reversed". We want to sort the individual reports so that the sample with highest easy and lowest intermediate is at the top of the list. If two reports have the same easy score, then the one with the lowest intermediate score is above the other. To achieve this, we take the intermediate number and subtract it from 99. Example in Figure 4 depicts a sample that has an "easy score" of 13 and an "intermediate score" of 7. The resulting "difficulty score" is therefore 13.92.


Easy: 13

Intermediate: 7

Difficulty score

= 13.(99 - 7)

= 13.92

Figure 4: Creation of difficulty score

We also have the hard score to consider but since it is out of scope and highlight samples that are too complex to use in an educational setting, we have placed them at the bottom of the meta report. They might still be of interest for an educator, so we have therefore included them.

All this tagging, counting and sorting has been done to simplify the workflow of the educator, so they do not use time finding and open each individual report. The meta report with all of the analyzed files with tags gives the educator a good overview and the ability to choose promising samples and then review their more detailed reports. If the detailed report looks promising, they then must manually open the file in an analysis tool of their choice and verify the findings. At this stage, the educator should have a sample that has a high probability of containing the functionality desired.

4 Results

We will now present our findings. The tool is created to take an arbitrarily large set of unknown malware samples and do individual analysis on them. The framework filters out undesirable such as duplicates, wrong filetype or unwanted compilers. The result of the analysis is two-phased: First is an individual report per analyzed file with highlighted and all findings, second is a meta-report that list the different files analyzed with their descriptive tags sorted so the potential easiest are at the top.

We have decided to call the framework “Picky” [41] as in “a picky eater”, because an educator should be picky about which sample they are going to analyze. When testing the framework, we have downloaded a set of 40 000 different malware samples, Block.102, from the vx-underground site. [42] This is a too large of a set to test out our framework on, so we randomly selected subsets of 100, 200 and 400 samples though scripting. These unknown sets of malware samples are what we will base our finding on. When running our analysis framework, we used an AMD Ryzen CPU with 8 cores, 16 threads, that could clock 4.05 GHz. Our system also had 16 GB of RAM and were running Windows 10 version 2004. The framework is written in Python, and we used Python 3.8 when doing the automatic analysis. The tool is written so that it can be used in both Windows and Unix operating systems.

4.1 Execution

We chose the set sizes of 100, 200 and 400 to emphasize that the framework indeed can take an arbitrary large set of unknown malwares and gather consistent results. The framework defaults to using 4 processes at the same time when doing bulk analysis, but during this testing we used 8 simultaneously processes. The execution time for 100 samples took 7 minutes and 7 seconds, for 200 it took 15 minutes and 50 seconds and for 400 unknown malware samples it took 28 minutes and 27 seconds.

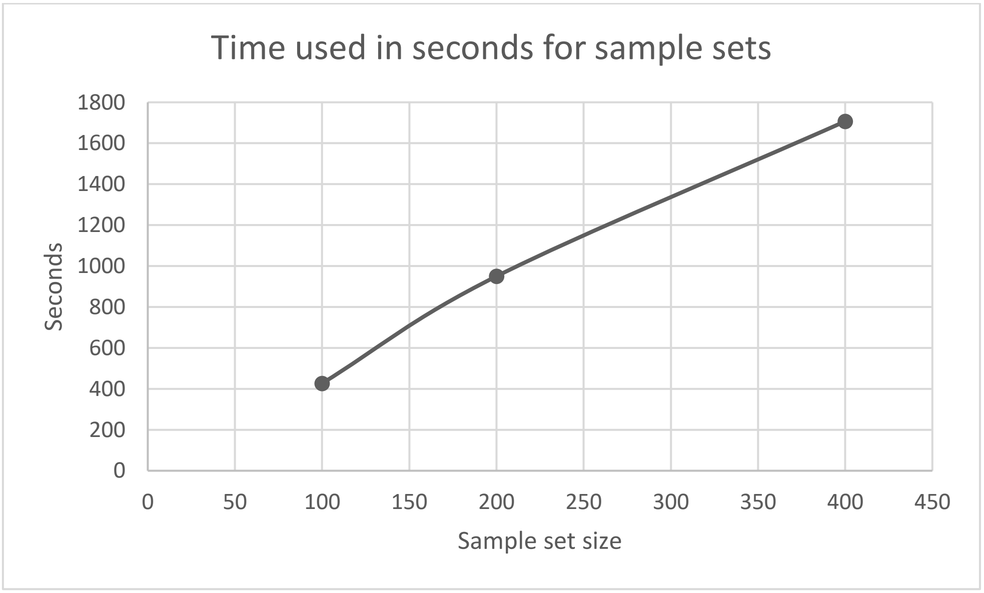


Figure 5: Graph of time used per sample set.

As we can see from Figure 5 the time used to analyze is an approximate linear function. This tells us that we in theory could analyze the whole 40 000 set of malwares given enough time.

In the analysis we filtered away samples that were not to our liking, such as non-PE files and PE files that were non-C compiled. In testing we found that some of the third-party tools would go into deadlock for unknown reasons. We added a timeout of 3 minutes per analysis, and we therefore have more begun analyzes than we have individual reports. Because of the filtering and force abortion of analysis, we had a lower amount of analyzed malwares than the original set size was. From a sample set of 100 we ended up with 65 analyzed, a reduction of 35%. The 200 set went down to 123 analyses, a reduction of 38,5% and 400 went down to 237, 40,8% reduction. From our 40 000-sample set we can assume we filter away and abandon analysis so that we lose on average 38,1% of the samples before analysis. This can vary to other sample sets, especially if the sample set is prefiltered with only PE files.

The average analysis time per full set was: 4,27 seconds for 100 samples, 4,74 seconds for 200 samples, and 4,27 seconds for 400. Adjusted for the number of analyzed without unwanted and aborted files is: 6,57 seconds for "100", 7,72 seconds for "200", and 7,20 seconds for "400". These are not the real time used per analysis since the analysis process is multi-processed with 8 simultaneous workers. We can 8 times each adjusted average to get an approximate estimate for how long one average analysis took.

Table 7: Average analysis time per sample in seconds

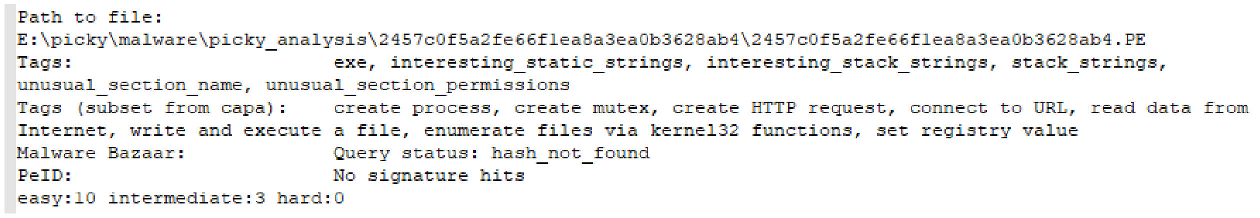
SIZE OF SET	AVERAGE (SECONDS)	AVERAGE ADJUSTED	REAL AVERAGE ESTIMATE
100	4,27	6,57	52,55
200	4,75	7,72	61,79
400	4,27	7,20	57,62

The real time average estimate includes files that were abandoned after 3 minutes of analysis, and therefore skewer slightly higher than what an actual flawless run would be. We could therefore reduce the time the program waits before aborting an analysis and not gain significant losses in unanalyzed files.

4.2 Testing use-cases

We will now present some fictional cases where an educator wants a specific sample to show some technique or example for the course content they are planning. We have created these use-cases from our own experiences when we have tried to find educational malware without this framework. We will use set meta report from the run of 400 samples, and we will start at the top and analyze downward until we find a sample and manually verify it with the individual report and manual reversing, our intended workflow.

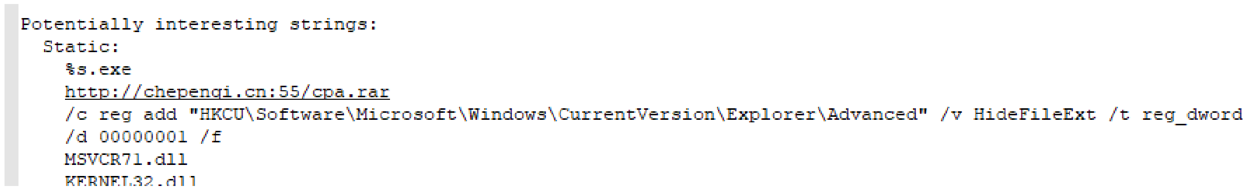
Our first use-case is: The educator wants a simple sample that show how it can look then a malware talks out to the internet. To answer this, we want a sample that is not packed, and has network related tags and possible unencoded URLs. We manually review the meta report and look for samples that do not include too many different elements. Within a minute or two, we find one that looks promising as they are sorted by difficulty score, see Figure 6.



```
Path to file:
E:\picky\malware\picky_analysis\2457c0f5a2fe66f1ea8a3ea0b3628ab4\2457c0f5a2fe66f1ea8a3ea0b3628ab4.PE
Tags:                          exe, interesting_static_strings, interesting_stack_strings, stack_strings,
unusual_section_name, unusual_section_permissions
Tags (subset from capa):       create process, create mutex, create HTTP request, connect to URL, read data from
Internet, write and execute a file, enumerate files via kernel32 functions, set registry value
Malware Bazaar:                Query status: hash_not_found
PeID:                          No signature hits
easy:10 intermediate:3 hard:0
```

Figure 6: Entry for first and simple use-case: unencrypted and network

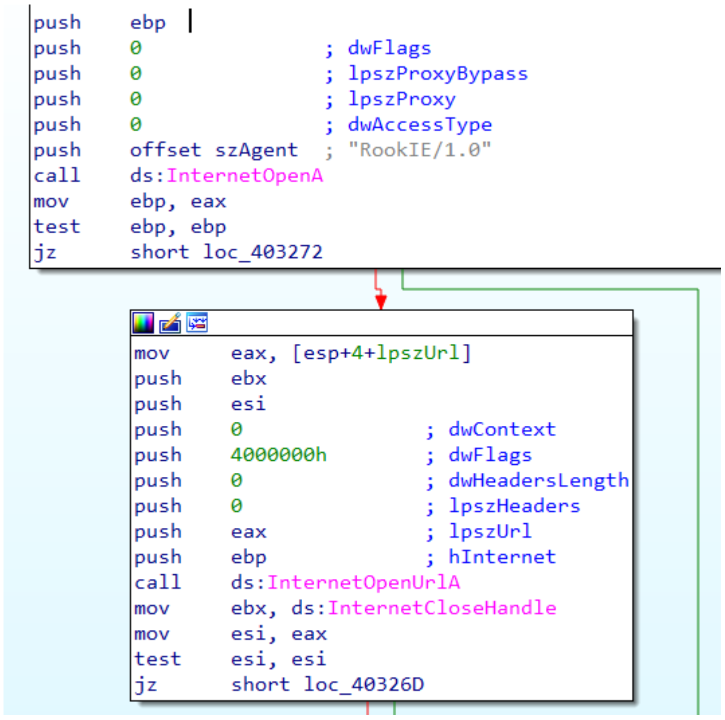
The sample entry tells us there are multiple indicators of network activity in this sample. It does not have a multitude of other functionality, making it potentially easy to navigate. It is not protected by a packer or protector, so there is a chance of unencrypted URLs. We open the individual report to verify our hypothesis. In the individual report we quickly see that the framework has wound a URL withing the code.



```
Potentially interesting strings:
  Static:
    %s.exe
    http://chepengi.cn:55/cpa.rar
    /c reg add "HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced" /v HideFileExt /t reg_dword
    /d 00000001 /f
    MSVCR71.dll
    KERNEL32.dll
```

Figure 7: Use-case one, URL in individual report

With capa finding network capabilities and the individual report found an URL, we take this file for manual verification to see if we can use this as an example in class. We open the file in IDAPro Disassembler [43] and look for the usage of that URL. We find that the URL is passed to a function, seen in Figure 8. Here we can see a user-agent that the framework did not identify. The code-flow and call graph are easy to follow, and this looks like a good example file to use for showcasing how one technique may look like.

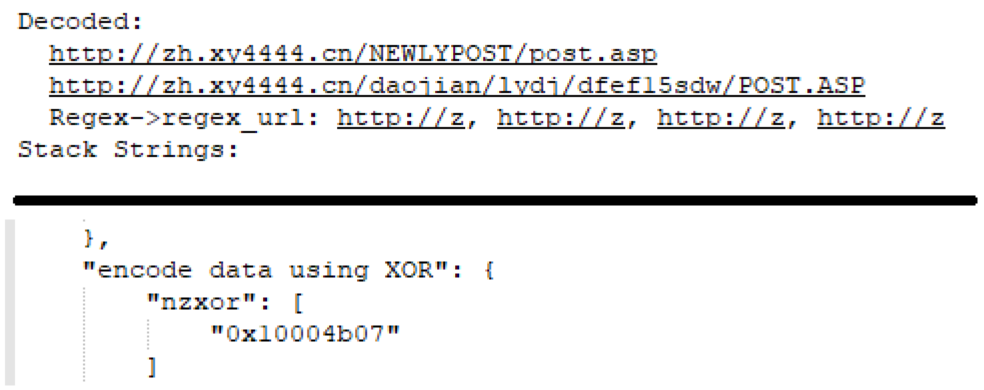


```
push    ebp
push    0               ; dwFlags
push    0               ; lpszProxyBypass
push    0               ; lpszProxy
push    0               ; dwAccessType
push    offset szAgent  ; "RookIE/1.0"
call    ds:InternetOpenA
mov     ebp, eax
test    ebp, ebp
jz      short loc_403272
```

```
mov     eax, [esp+4+lpszUrl]
push    ebx
push    esi
push    0               ; dwContext
push    4000000h        ; dwFlags
push    0               ; dwHeadersLength
push    0               ; lpszHeaders
push    eax             ; lpszUrl
push    ebp             ; hInternet
call    ds:InternetOpenUrlA
mov     ebx, ds:InternetCloseHandle
mov     esi, eax
test    esi, esi
jz      short loc_40326D
```

Figure 8: Use-case one, manual verification

The next case that we want to simulate is: The educator wants to find some encryption routine that the students' ether must decode manually or debug to get the decoded string. We open the meta report again search for "decoded_strings" to find potential interesting entries. There are multiple false positives where the decoded strings are garbage, but after a couple of minutes of searching and opening individual reports we find a promising sample that has URLs decoded. We find the location of where capa believes it has found a XOR-encoding routine. Given that we know XOR-encoding can be decoded with the same routine we go to that location in IDAPro disassembler to verify what it does.

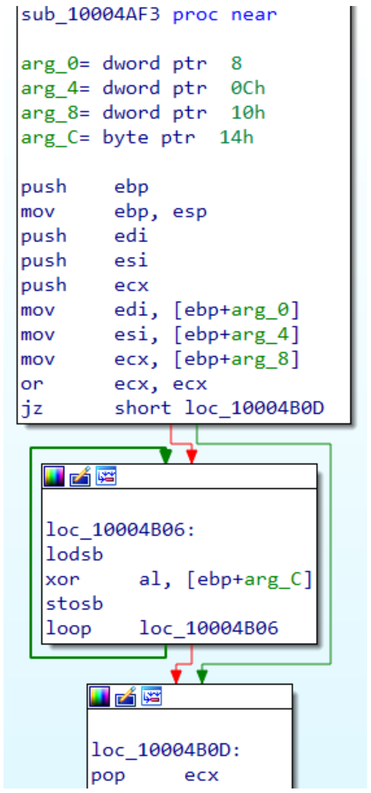


```
Decoded:
  http://zh.xy4444.cn/NEWLYPOST/post.asp
  http://zh.xy4444.cn/daojian/lydj/dfefl5sdw/POST.ASP
  Regex->regex_url: http://z, http://z, http://z, http://z
Stack Strings:
```

```
    },
    "encode data using XOR": {
        "nzxor": [
            "0x10004b07"
        ]
```

Figure 9: Use-case two, individual report with strings and found decoding routine.

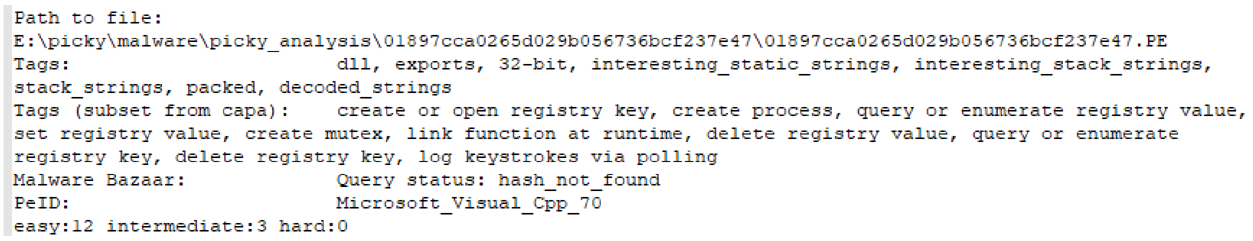
Inside IDAPro we find the function that capa has given to us. The function seems simple enough, and with a quick test we see that this is a simple XOR-function that takes encoded values and decrypts them, see Figure 10.



```
sub_10004AF3 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= byte ptr  14h

push    ebp
mov     ebp, esp
push    edi
push    esi
push    ecx
mov     edi, [ebp+arg_0]
mov     esi, [ebp+arg_4]
mov     ecx, [ebp+arg_8]
or      ecx, ecx
jz      short loc_10004B0D

loc_10004B06:
lodsb
xor     al, [ebp+arg_C]
stosb
loop    loc_10004B06

loc_10004B0D:
pop     ecx
```

Figure 10: Use-case two, found decryption routine.

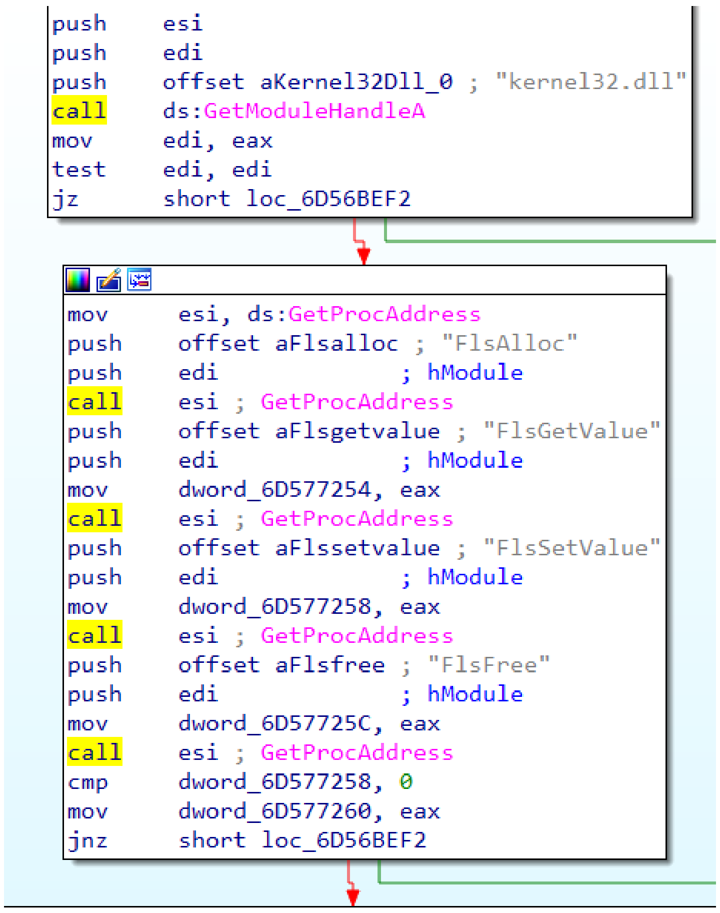
The third case is "The educator wants to showcase how a sample can hide its functions and how they are used, and only get the needed imports at runtime". To find suitable samples for this we will search for "link function at runtime" in the meta report. After a couple of uninteresting samples, we find an unpacked promising one. We choose this sample because it has few other capabilities that might heighten the complexity.



```
Path to file:
E:\picky\malware\picky_analysis\01897cca0265d029b056736bcf237e47\01897cca0265d029b056736bcf237e47.PE
Tags:                       dll, exports, 32-bit, interesting_static_strings, interesting_stack_strings,
stack_strings, packed, decoded_strings
Tags (subset from capa):    create or open registry key, create process, query or enumerate registry value,
set registry value, create mutex, link function at runtime, delete registry value, query or enumerate
registry key, delete registry key, log keystrokes via polling
Malware Bazaar:             Query status: hash_not_found
PeID:                       Microsoft_Visual_Cpp_70
easy:12 intermediate:3 hard:0
```

Figure 11: Use-case three, simple promising sample

There are not too many interesting things highlighted for this use-case. We again search "link function at runtime" to go down to where capa has found the locations. Inside of the overarching rule, there are display locations for GetProcAddress inside of the sample. We note down those and open the sample in the disassembler to verify that they can be used.



```
push    esi
push    edi
push    offset aKernel32Dll_0 ; "kernel32.dll"
call    ds:GetModuleHandleA
mov     edi, eax
test    edi, edi
jz      short loc_6D56BEF2

mov     esi, ds:GetProcAddress
push    offset aFlsalloc ; "FlsAlloc"
push    edi             ; hModule
call    esi ; GetProcAddress
push    offset aFlsgetvalue ; "FlsGetValue"
push    edi             ; hModule
mov     dword_6D577254, eax
call    esi ; GetProcAddress
push    offset aFlssetvalue ; "FlsSetValue"
push    edi             ; hModule
mov     dword_6D577258, eax
call    esi ; GetProcAddress
push    offset aFlsfree ; "FlsFree"
push    edi             ; hModule
mov     dword_6D57725C, eax
call    esi ; GetProcAddress
cmp     dword_6D577258, 0
mov     dword_6D577260, eax
jnz     short loc_6D56BEF2
```

Figure 12: Use-case three, common way to dynamically resolve API functions.

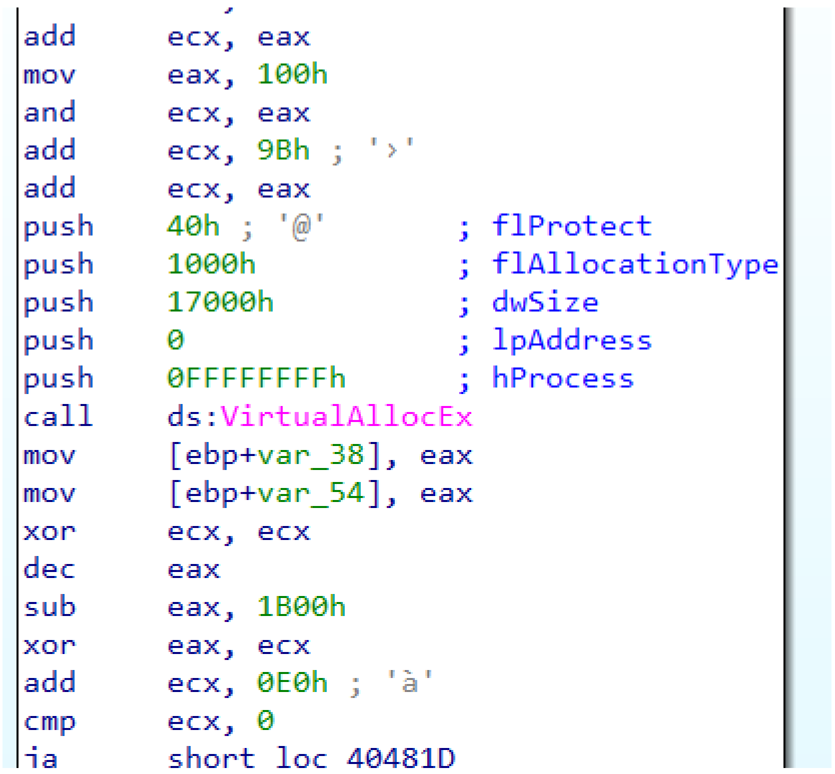
In Figure 12 we see one of the locations, and here we have a perfect example of dynamically resolving Windows API functions and storing them om variables for later use. This is a good example to use for educational purposes.

The last case is a little more advanced: “The educator wants to show different techniques to hide running code. They want examples on both within itself and the use of another process”. We want to begin finding samples that showcase internal unpacking and code hiding. Therefore, we search for “allocate RWX memory” in the meta report for possible samples because we know that a common way is to allocate new memory area with executable permissions. After a couple if tries we find a sample with few other indicators that seems interesting.

```
Path to file:
E:\picky\malware\picky_analysis\1332966b92c3db095b36b7d9b1900ebb\1332966b92c3db095b36b7d9b1900ebb.PE
Tags:                         exe, interesting_static_strings
Tags (subset from capa):      allocate RWX memory, execute anti-VM instructions, link function at runtime
Malware Bazaar:               Query status: hash_not_found
PeID:                         No signature hits
easy:3 intermediate:2 hard:0
```

Figure 13: Use-case four: RWX memory

The individual report is short and does not tell much about the sample. It says that it is probably not packed but based upon the imports and few interesting indicators we believe this to be false. We go to the only entry in "allocate RWX memory" and open the sample in disassembler. The sample does not have many subroutines and has indication of anti-disassembly capabilities. We go to the reported location and find the VirtualAlloc call.

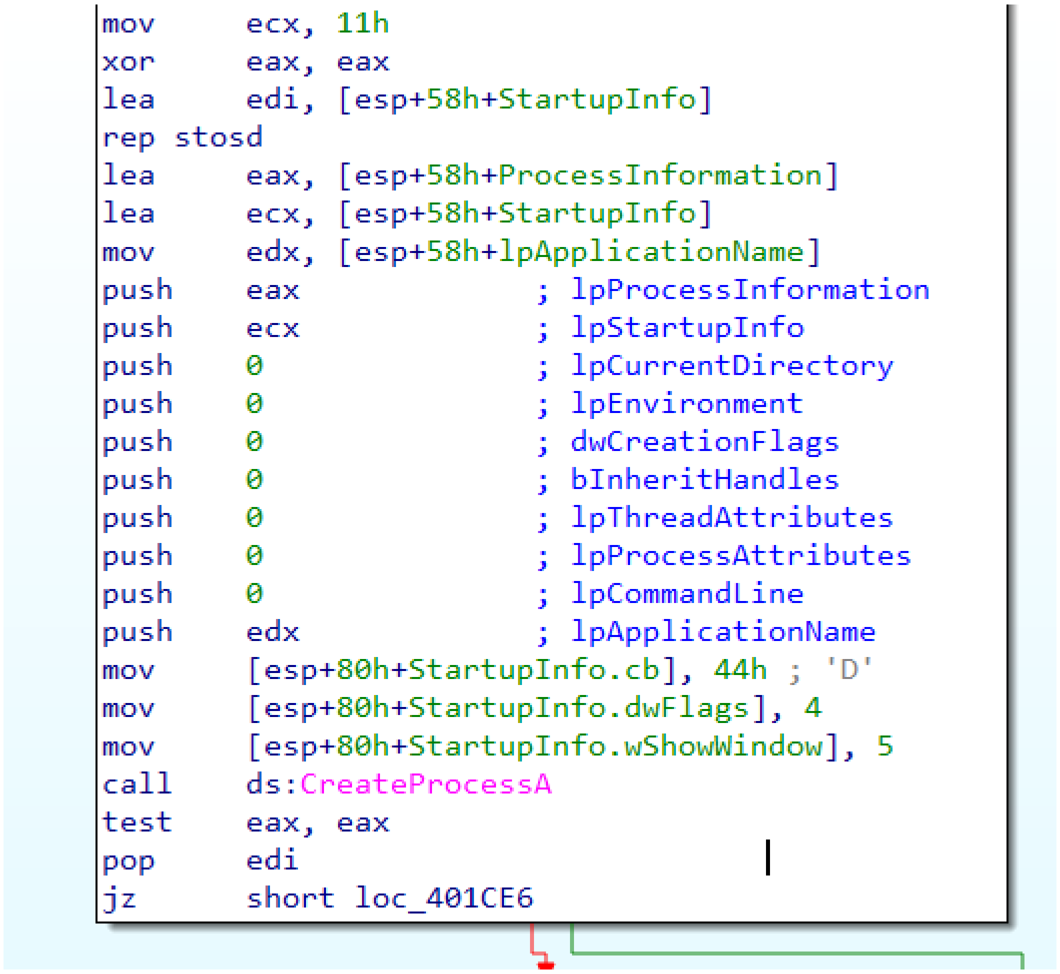


```
add     ecx, eax
mov     eax, 100h
and     ecx, eax
add     ecx, 9Bh ; '>'
add     ecx, eax
push    40h ; '@'           ; flProtect
push    1000h               ; flAllocationType
push    17000h              ; dwSize
push    0                   ; lpAddress
push    0FFFFFFFFh          ; hProcess
call    ds:VirtualAllocEx
mov     [ebp+var_38], eax
mov     [ebp+var_54], eax
xor     ecx, ecx
dec     eax
sub     eax, 1B00h
xor     eax, ecx
add     ecx, 0E0h ; 'à'
cmp     ecx, 0
ja      short loc_40481D
```

Figure 14: Use-case four, VirtualAlloc with RWX permissions.

We have clear example of a large allocation of executable memory. This is probably where all of the malicious code will be placed after decryption. The sample also has many other interesting attributes, but these are out of scope for this use-case.

To begin finding sample that can satisfy the other part of the use-case, we search for "create process suspended" in the meta report. We do the process as we have done multiple times before; find an entry that is promising, investigate the individual report, and verify the findings in the sample. In this case we found a sample that seems to do many other thigs as well as stating a process in suspended mode. The individual report only has one entry for process created suspended, so we open the sample in a disassembler.



```
mov     ecx, 11h
xor     eax, eax
lea     edi, [esp+58h+StartupInfo]
rep stosd
lea     eax, [esp+58h+ProcessInformation]
lea     ecx, [esp+58h+StartupInfo]
mov     edx, [esp+58h+lpApplicationName]
push    eax                 ; lpProcessInformation
push    ecx                 ; lpStartupInfo
push    0                   ; lpCurrentDirectory
push    0                   ; lpEnvironment
push    0                   ; dwCreationFlags
push    0                   ; bInheritHandles
push    0                   ; lpThreadAttributes
push    0                   ; lpProcessAttributes
push    0                   ; lpCommandLine
push    edx                 ; lpApplicationName
mov     [esp+80h+StartupInfo.cb], 44h ; 'D'
mov     [esp+80h+StartupInfo.dwFlags], 4
mov     [esp+80h+StartupInfo.wShowWindow], 5
call    ds:CreateProcessA
test    eax, eax
pop     edi
jz      short loc_401CE6
```

Figure 15: Use-case four, example process started as suspended.

Figure 15 show the highlighted location from the individual report. Immediately it seems that the capa-tool has created a false positive, since dwCreationFlags are set to 0, which is nothing. [44] At more thorough investigation we see that it tries to trick analysis programs by setting the dwCreationFlags inside of the structure on the stack. Here we see that it adds 4 to the dwFlags that corresponds to dwCreationFlags and means "CREATE_SUSPENDED". The command line for which process should be started suspended is not shown in this function but requires a little reversing to locate. This is a suitable example that require a little explanation, so the student understand the technique.

Each of these use-cases are relevant for what an educator would want to look for. In our testing and usage of the reports we never used more than 10 minutes from start until we had found a suitable sample. We had no prerequisite knowledge about these samples, and only used the reports and at last step verified through manual analysis.

4.3 Interview analysis

In this subchapter we will analyze the results of the built framework against the criteria given by the interviews. We will test and see if we have created a framework that can fulfill the expert's criteria when finding educationally friendly malware. We will be analyzing the answers presented in chapter 3.1 Interviews against the results found when running the framework.

From the interviews it came forth that many of the experts use samples that they already have analyzed because of other reasons than education. As one said, "*Randomly searching though samples that I've already analyzed to find something appropriate*.". This framework could help the process of searching though already analyzed samples to find appropriate matches. As the tool creates a meta report, it could help the educators remember how the different samples functions and if they are suitable for education. Reviewing the meta report should be easier to get an overview of the different samples than going into each analysis file the educator has, to look for clues and remembering how the sample function. The other process of finding samples where to "*Find samples though research. Next I look at how easily explainable they are.*" This process should the framework help with as well, by giving the educator a reasonable starting point with the meta report. The report may not be able to tell the educator how easily explainable the samples are but should give some hints as to which files to dedicate time to and find a suitable sample.

When it comes to time spent analyzing samples to find the best sample for educational purpose, the experts used different amount of time per sample. As one said: "*If it is simple, it could be like an hour, and more complex samples might take a day*". This is quite the stark opposite of what another said: "*It might take a month or so to make the course content around it…*". We realize that the educator would not use a month on doing analysis of the sample, but the example still stands. The educators can use multiple hours or days analyzing samples, especially when using brand new samples. This framework can cut that time significantly down by narrowing down the search to only samples that has a high probability of containing useful code and techniques.

A part of finding educationally friendly malware is to remove samples that are not educationally friendly. The educators agreed that malware with destructive capabilities and ransomware was undesirable. "*[I] don't use many different ransomware samples because they often do the same thing*", said one educator. "*[…] Malware with destructive capabilities, such as deleting files or destroying the operating system*", said another when ask about no-go malware. The framework tries to tag and categorize samples that has indications of being ransomware or has destructive capabilities, but we are uncertain how precise this is. The educators also agreed that samples with too complex encryption or that uses commercial, and advance packers should be avoided. We have implemented the PEiD ruleset for detection of these packers and techniques, so that the educators are aware which samples is packed by what packer. We also have created lists for what we believe to be packers of intermediate and hard reversing-challenge to make the sorted list better. We believe therefore that the framework fulfills many of the criteria set by the educators interviewed in this paper. We have seen through use-cases that we can identity and find samples that can be of interest in different cases, as well as filter samples away that contains unwanted packers or seems too complex.

5 Discussion

We will in this chapter present our discussion about this project, the framework and the presented results. The discussion chapter is important to highlight the nuances of the project, and to present both the positive the thesis achieved and how it could have been done better. We will try and highlight weaknesses in the paper and try to present alternatives as to how we could have done it. First, we will analyze and discuss how we have answered the research questions, and next we will discuss other flaws and shortcomings.

5.1 Reviewing research questions

In this subchapter we will discuss the research questions presented at the start of this thesis, and review that against our findings. We will test and see if we have answered the questions, and at what that answer is.

The first question we will be discussing is: "What constitute static malware analysis, and which tools are the "state-of-the-art" of statically analyzing malware samples?" The first part of this question we answered quite easily in chapter 2.1, as it is a question of definition. The second par was more difficult to answer as it is more of a question of preference. In chapter 3.3 we present and talk about different tools and their use-cases, and how we can use them in our framework. Some of these tools are old, such as PEiD and the library pefile [45], but we do not believe this to be a weakness. We believe that the tools used still are the best, or "state-of-the-art", when it comes to the work they do. We were also partial in the selection process as they are tools that we use in our professional work, and we had pre-thesis knowledge in their strengths and weaknesses. When it comes to the advanced static analysis tools capa and FLOSS, they are "state-of-the-art" as they are the only tools that tackle the advanced static analysis process. They also have shown that they can present incredible results only rivaled by dynamic analysis.

The second question are two-part question as well: "What indicates that a malware sample is educationally friendly or suitable?", and "How can we use tools to find and display these indicators or features?". The first question is what we in large try to answer in this master thesis. We have used interviews, educational resources and our own expert knowledge to answer this, and in chapter 3.2 we try to explain the different parts that we believe to be what makes a sample educationally friendly. We believe that through our methods of research and experience, has found good and general indicators on what makes C or C++ compiled malware educationally friendly. We have seen with our testing of the proof-of-concept framework that we can find suitable samples based upon our definition of what an educationally friendly malware could be. The second part of the question is answered in the chapter 3.4, where we go through how we use and build or framework. We went for a "run once" tool and review the text reports that show our findings for each analyzed sample as well as an overview meta report that makes it easier to find samples to start with. Proof-of-concept framework uses our indicators of educationally friendly malware, with tools and our own analysis to create extensive reports with the meta report sorted by difficulty score.

The third question we wanted to have answered were: "How much time do normally educators use to find and understand their malware samples to use in an educational setting?" We mainly answer this through our interviews, and our findings presented in chapter 3.1. The people interviewed might not be a representative group for all educators within malware analysis, and the timeframe for malware analysis given from them might therefore be larger or smaller than the global average. We however concur with the interview objects on the time estimate and believe it possible that it can take anywhere from an hour to days before you have a good sample set of educationally friendly malware.

The fourth question was: "Are we able with minimal manual verification able to list possible educationally friendly malwares from an arbitrarily large set of unknown malwares?" This is a simple yes or no question, but essential for the tool to be useful for other educators. We answer the question in the chapters 4.1 and 4.2 in "Results". In the execution of the program, we used sets of unknown malwares in different set sizes. The sets were quite large in our experiences but might be too small for other educators. We showed through the test execution that given time and processing resources, the framework can take any sample set size and produce results. In our case there were a high percentile of unanalyzed samples, but this is because of our defined delimitations and that the sample set were truly random with even uncompiled malware samples as well. We believe the percentile to be much lower in prefiltered malware sets. The creating of the easy, intermediate and hard categories, as well as the sorting by difficulty score was done to minimize the need for manual analysis to find suitable samples. Through our use-cases we walked through how an educator could use the reports and confirmed that we rarely opened a sample for manual analysis in vain. We see that the educator must be knowledgeable in their field of malware analysis to be able to find the potential samples in between all the other reported samples. It might also take time understanding our thought process and tags naming convention and what they represent.

The fifth question and possible the most important question regarding the usefulness of our findings and proof-of-concept is: "Can we through automated static analysis shorten the time an educator uses from having unknown samples until they have samples with the right techniques wanted to use as examples or exercises in an educational setting?" This question is tough to answer has we have not had quality assurance testing with people from outside of our project. We have created simulated settings through our use-cases that show how the tools can be used and how an educator might find the right sample. When we, as creators of this framework, tried to find suitable samples for the different use-cases we never used more than 10 minutes from having no sample until we had a sample matching the criteria. We believe that an uninvolved third party can use this framework and still achieve immense reduction in time spent analyzing malware samples. Even if they were to use 30 minutes to an hour to find a suitable sample, it is much faster than the expected time spent that is hours to days. We therefore strongly believe that we have created a tool that can help malware educators and analysts reduce time spent analyzing, and rather focus on creating the best possible educational content.

5.2 Weakness and improvements

Now we will present some weaknesses that we have found in our master thesis. We will discuss how we might have improved them, and maybe give examples of future work that the project could have.

The first weakness with this master thesis we want to highlight is the interviews that we conducted. The interviews were a little shorter than anticipated. The semi-structured interview guide bore the mark of being constructed by an inexperienced researched that did not know exactly what they wanted answered, and how technically focused it should be. We did try to run quality assurance tests on it by consulting an experienced researching professor, and we did not receive any damning remarks. The question asked in the interview did not give us a large, differentiated material baseline as we believed it would give us. The answers we received where quite uniform and did not present any new insight into the process of creating an educational course. The amount of people that we were able to interview was lower than expected, with only three willing participants. They were however experts in their fields and had great experience in the educational sphere as professor or as teaching leads in an organization. We used our supervisor's influence and contacts to gather as many interview objects as possible, and it is his help that we had three interview objects. The interviewees were from around the globe, giving us potential insight into professional differences if there were any. We should have programmed into the projects plan a review of the product by the interview objects as to get feedback on the proof-of-concept framework. The feedback could have given us clear answers on if the framework could help them solve the problem of finding educationally friendly malware.

The next problem with this master thesis is the fact that there is little new data gathered in the project. We realize that the project manly focuses on finding existing resources and gathering information around the subject of educating people in analyzing malware. Much of the more official resources like books are becoming outdated with no real alternative. We have therefore relied extra on other sources, and we tried to compensate with using our own professional knowledge. Especially in choosing which indicator belonged in what category of difficulty we had to rely on our own expertise as we did not find that anyone else had done something similar. What we have tried to achieve in this paper is to compile extensive lists based upon our research that can help educators find educationally friendly malware form those indicators listed. We believe that we have gathered and presented information in such a way that it could be easily used by others in their work with finding simple and educational malware. If anything, we have created a good basis for tagging and categorization that could be useful for machine learning. As our framework has shown, it is capable of tagging unknown malware quite well with tags that can be used to describe its functionality.

We have not listed or presented many "state-of-the-art" projects or software in this thesis. This is because this topic of defining educationally friendly malware does not seem to have been researched any notable extent in any found projects. We think this is because those who present malware educational content has an instinctive understanding of what educational friendly malware is. The most accurate "state-of-the-art" withing educationally friendly malware would be the book "Practical Malware Analysis" from the year 2012. We hope at this master thesis can help start a continuous work with finding and presenting educationally friendly malware content.

When programming the framework, we encounter multiple problems and had to go for solutions that were not ideal or considered best coding practice. We early identified that the programming language Python were the best suit for the project, as we were most proficient with it and relevant tools were written in it. Python is primarily a scripting language for smaller tasks and require specific knowledge around optimization to be used in an efficient way with larger tasks. Under the performance testing we saw that the time used per analysis averaged around six to eight seconds. That is quite low given the amount of analysis done per sample, but it is not considered that there were eight simultaneous processes running. When approximated with for what one analysis actually used, we see that the average analysis time per sample were around one minute. This a little too much time used per automated analysis, and there should be work done to lower the time used. If a user were to analyze thousands of files, the time used now would be too high to expect the user to wait for the analysis completion. The biggest time-consuming tasks were the capa- and FLOSS-analysis, which is natural since they do advanced static analysis. We could reduce the time used by implementing FLOSS as a python library as we did with capa, but we did not have time or resources to inline it the same way. We therefor had to rely on the precompiled exe-version and start it as a new process. Future work would definitely be implementing the floss-python scripts.

Another cause of the analysis time being so high is that some samples for unknown reasons made the tools run into a dead-locked and become unable to continue. We created a fail-safe that aborted an analysis if it took more than three minutes, but this seems to have pulled the average upwards. Since we now know that the average time used per file is around one minute, we could lower the fail-safe timer to potential lower the average, at the cost of increasing the unanalyzed rate. We could also write more of our own tools and not rely on other prebuilt tools to do the work. Writing our own tool is a little against the spirit of the open-source and sharing community that is around cyber security. Since we are not educated programmers, there is a high chance that our programming skills has had a negative impact on the time used. In the framework we have tried to take measures to lower the time, such as loading rules and other shared data at start of the program and not each time a sample is analyzed.

The program we have created analyze and write a report on every file that fits the criteria of being C or C++ compiled. This filles the analysis directory and the meta report with samples that are not necessarily educationally friendly. This is by choice, as we saw that many samples could have educational properties even if they were packed by advanced packers. We have therefore decided to include all files in the meta report, so that the educator has a broader sample set to work with. We have sorted the samples by easiest first, so it would be easier for the educator to find possible samples, but it still requires manual work to find the right sample. When the educator has learned how to read and understand the meta report, they should quite easily be able to overlook samples that are uninteresting and locate promising samples and verify those instead.

6 Conclusion

We will in this chapter go through what we have been doing in this master thesis, what we wanted to find out and how we went about doing it. We will present our core findings and conclude if we believe this project to be a success.

In this master thesis we wanted to find out "*Can we with the help of automated analysis shorten the time used to find malware samples suitable for educational purposes by categorizing and capability tagging malware samples from an arbitrarily large set of unknown malwares?*". We wanted to create a proof-of-concept that could take an arbitrary large set of unknown malwares and do analysis on all the files and create a report that said something about the files capabilities to be an educationally friendly malware. To be able to create a framework to find educationally friendly malware, we first had to define what educationally friendly meant.

To understand what was implied when talking about educationally friendly malware, we conducted semi-structured interviews with professionals and experts within the field of malware analysis and educating others in the field. After we had the interviews, we research different resources on the subjects of educating in malware analysis. Through our research, and professional experience, we compiled lists with different indicators that we believed to directly connected to malware often used in educational settings. These compiled lists took information from many different sources and are the basis for how we constructed our proof-of-concept framework.

When we had a basis of what educationally friendly malware was, we began creating a framework that could pick out these indicators from a set of malware files. We called the framework "Picky" [41], as in "being a picky eater" because there is no reason in not choosing the best sample when creating an educational program. We tested the framework against up to 400 different and unique malware samples to show that it was able to take large sets of samples. The results were two reports: one individual report for each sample containing verbose information, and one meta data report containing crucial tags for each analyzed sample. The meta data report was created so that the educator would have one place to begin is hunting, and to quickly get an overview of what the different samples were capable of.

We tested the reports against different lifelike use-cases to verify that our findings did in fact find educationally friendly malware. We saw through our use-case analysis that one could not blindly trust a that a sample was educationally friendly, as it required some knowledge and analysis of the individual reports to choose which samples you wanted to verify further. We did however show that the samples picked by the "educator" did solve their use-cases and that the reports were accurate in their tagging and location within the file. We also saw that the framework can, albeit given time and resources, analyze however many samples at the educator whish.

Lastly, we discussed the weaknesses and problems that there are with this master thesis. The thesis is a little light on presenting new data and information and focuses a little too much on collecting existing data and using that. The project also could have been better planned around the use of interview and scheduled for follow-up interviews where the experts are presented with the proof-of-concept and prompted for feedback.

We believe that we have successfully been able to create a framework that can reduce time spent analyzing and help educators and malware analyst finding samples that fit their use-cases. The framework is not meant to do malware analysis, but to tag and present findings in such a way that an educator can more accurately find suitable samples. This does however not exclude that the analysis done on the samples could also be used for real life malware analysis.

Future work with this project would first to make it more stable and precise. Then we could introduce machine learning to create a smaller, but more fitting set of found malware that are more likely to be containing the desired functionality that an educator would want. This could reduce the time searching for the perfect sample even greater than already achieved. We could also introduce analysis of other files PE files from compiled C or C++. PE files created from C# would be the obvious next step, and later non-compiled files such as scripts.

This concludes our master thesis, where we created an automatic framework that could take an arbitrarily set of malwares and create a subset of potential educationally friendly malwares. We hope that this framework will help educate the next generation of malware analysts and reverser so that we have a better chance of defeating the malwares of tomorrow.

7 References

[1] M. Sikorski and A. Honig, Practical Malware Analysis, San Francisco, CA, USA: No Stach Press, 2012.

[2] G. Dalakov, "The First Computer Virus of Bob Thomas (Complete History)," [Online]. Available: https://history-computer.com/the-first-computer-virus-of-bob-thomas-complete-history/. [Accessed 13 05 2021].

[3] R. A. Grimes, "9 types of malware ad how to recognize them," CSO , 17 11 2020. [Online]. Available: https://www.csoonline.com/article/2615925/security-your-quick-guide-to-malware-types.html. [Accessed 13 05 2021].

[4] NTNU, "IMT4116 - Reverse Engineering and Malware Analysis," [Online]. Available: https://www.ntnu.no/studier/emner/IMT4116#tab=omEmnet. [Accessed 19 05 2021].

[5] B. Herzorg, "Malware Against the C Monoculture," Checkpoint, 20 05 2019. [Online]. Available: https://research.checkpoint.com/2019/malware-against-the-c-monoculture/. [Accessed 27 05 2021].

[6] D. Bisson, "Threat Actors Use Delphi Packer to Shield Binaries From Malware Classification," 09 10 2018. [Online]. Available: https://securityintelligence.com/news/threat-actors-use-delphi-packer-to-shield-binaries-from-malware-classification/. [Accessed 27 03 2021].

[7] F. O. Catak, A. F. Yazi, O. Elezaj and J. Ahmed, "Deep learning based Sequential model for malware analysis using Windows exe API Calls," PeerJ Computer Science, 2020.

[8] S. Gadhiya and K. Bhavsar, "Techniques for Malwre Analysis".

[9] W. Ballenthin and M. Raabe, "capa: Automatically Identify Malware Capabilities," 16 07 2020. [Online]. Available: https://www.fireeye.com/blog/threat-research/2020/07/capa-automatically-identify-malware-capabilities.html. [Accessed 21 03 2021].

[10] J. Arends, "Malware Analysis Tools and techniques," University of Applied Science Bonn-Rhein-Sieg, Sankt Augustin, 2018.

[11] R. Rivest, "RFC 1321: The MD5 Message-Digest Algorithm," MIT Laboratory for Computer Science, 1992.

[12] D. Eastlake and T. Hansen, "RFC 6234: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDP)," Internet Engineering Task Force, 2011.

[13] Microsoft, "PE Format," Microsoft, 31 03 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. [Accessed 14 05 2021].

[14] A. V. Aho, R. Sethi and J. D. Ullman, Compilers, Principles. Techniques, and Tools, Addison-Wesley Publishing, 1986.

[15] Microsoft, "HttpSendRequestA function (wininet.h)," 12 05 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/wininet/nf-wininet-httpsendrequesta. [Accessed 29 05 2021].

[16] R. A. Suryadi, "Malware, hiding api call from static analyses 1," 14 06 2020. [Online]. Available: https://rioasmara.com/2020/06/14/malware-hiding-api-call-from-static-analyses-1/. [Accessed 29 05 2921].

[17] S. Gupta, H. Sharma and S. Kaur, "Malware Characterization Using Windows API Call Sequences," River Publishers, 2018.

[18] R. Jullian, "Formbook In-depth malware analysis," 2018.

[19] agentzex, "The-Nice-Ransomware," 2019. [Online]. Available: https://github.com/agentzex/The-Nice-Ransomware. [Accessed 01 05 2021].

[20] T. Lambert and B. Donohue, "It’s all fun and games until ransomware deletes the shadow copies," Red Canary, 21 08 2019. [Online]. Available: https://redcanary.com/blog/its-all-fun-and-games-until-ransomware-deletes-the-shadow-copies/. [Accessed 29 05 2021].

[21] A. Hosseini, "Ten process injection techniques: A technical survey of common and trending process injection techniques," Elastic, 18 06 2017. [Online]. Available: https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process. [Accessed 14 05 2021].

[22] Unknown, "Virtual Machine Detection Techniques," 17 07 2017. [Online]. Available: https://shasaurabh.blogspot.com/2017/07/virtual-machine-detection-techniques.html. [Accessed 02 05 2021].

[23] The Open Group , "Regular Expressions," IEEE, 2018. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html. [Accessed 02 05 2021].

[24] GCHQ, "CyberChef," 2021. [Online]. Available: https://github.com/gchq/CyberChef. [Accessed 02 05 2021].

[25] D. Stevens, "re-search.py," 21 02 2021. [Online]. Available: https://github.com/DidierStevens/DidierStevensSuite/blob/master/re-search.py. [Accessed 02 05 2021].

[26] K. Hahn, "Malware Naming Hell Part 1: Taming the mess of AV detection names," GDataSoftware, 08 12 2019. [Online]. Available: https://www.gdatasoftware.com/blog/2019/08/35146-taming-the-mess-of-av-detection-names. [Accessed 30 05 2021].

[27] M. Russinovich, "Strings v2.53," Microsoft, 17 09 2020. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/strings. [Accessed 15 03 2021].

[28] FireEye, "FLOSS," 05 02 2021. [Online]. Available: https://github.com/fireeye/flare-floss. [Accessed 15 03 2021].

[29] C. Young, "Ghidra 101: Decoding Stack String," Tripwire, 26 01 2021. [Online]. Available: https://www.tripwire.com/state-of-security/security-data-protection/ghidra-101-decoding-stack-strings/. [Accessed 03 05 2021].

[30] M. Raabe and W. Ballenthin, "Automatically Extracting Obfuscated Strings from Malware using the FireEye Labs Obfuscated String Solver (FLOSS)," FireEye, 23 06 2016. [Online]. Available: https://www.fireeye.com/blog/threat-research/2016/06/automatically-extracting-obfuscated-strings.html. [Accessed 03 05 2021].

[31] E. O. Osaghae, "Packed MalwareDetection using Entropy RelatedAnalysis: A Survey," IOSR Journal of Engineering, Lokoja, 2015.

[32] Aldeid, "PEiD," 11 04 2020. [Online]. Available: https://www.aldeid.com/wiki/PEiD. [Accessed 15 03 2021].

[33] VirusTotal, "YARA," [Online]. Available: https://support.virustotal.com/hc/en-us/articles/115002178945-YARA. [Accessed 15 03 2021].

[34] VirusTotal, "Yara-Rules," [Online]. Available: https://github.com/Yara-Rules/rules. [Accessed 15 03 2021].

[35] T. Roccia, "Malware Packers Use Tricks to Avoid Analysis, Detection," McAfee, 16 05 2017. [Online]. Available: https://www.gdatasoftware.com/blog/2019/08/35146-taming-the-mess-of-av-detection-names. [Accessed 30 05 2021].

[36] H. Unterbrink, "Dynamic Data Resolver(DD) - IDA Plugin 1.0 Beta," Talos Intelligence, 28 05 2020. [Online]. Available: https://blog.talosintelligence.com/2020/05/dynamic-data-resolver-1-0.html. [Accessed 05 05 2021].

[37] VirusTotal, "VirusTotal," [Online]. Available: https://www.virustotal.com/gui/. [Accessed 15 03 2021].

[38] VirusTotal, "VirusTOtal API version 3 Overview," [Online]. Available: https://developers.virustotal.com/v3.0/reference#overview. [Accessed 15 03 2021].

[39] abuse.ch, "MalwareBazaar API," [Online]. Available: https://bazaar.abuse.ch/api/. [Accessed 15 03 2021].

[40] Kindredsec, "The Basics of Packed Malware: Manually Unpacking UPX Executables," 07 01 2020. [Online]. Available: https://kindredsec.wordpress.com/2020/01/07/the-basics-of-packed-malware-manually-unpacking-upx-executables/. [Accessed 30 05 2021].

[41] A. Bjørkhaug, "picky," 05 2021. [Online]. Available: https://github.com/xor-al-ex/picky. [Accessed 15 05 2021].

[42] vx-underground, "vx-underground samples," [Online]. Available: https://vx-underground.org/samples.html. [Accessed 15 05 2021].

[43] Hex-Rays, "Hex-Rays," 2020. [Online]. Available: https://www.hex-rays.com/products/ida/. [Accessed 12 03 2021].

[44] Microsoft, "Process Creation Flags," 27 07 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/procthread/process-creation-flags. [Accessed 30 05 2021].

[45] erocarrera, "Github pefile," 18 04 2019. [Online]. Available: https://github.com/erocarrera/pefile. [Accessed 18 03 2021].

Attachment 1: Interview guide

To whom it may concern,

My name is Aleksander Bjørkhaug. I am a student at the Norwegian Institute of Technology and is currently doing my master thesis. My supervisor is associate professor Geir Olav Dyrkolbotn. My thesis aim to automate the process of finding malware samples that are educational friendly, for training both entry level and intermediate level malware reversers.

Even though there is an abundance of malware samples available, it is difficult and time consuming to find samples suitable for a pedagogic approach in an educational setting. This is especially evident for reversers that just received their basic training and want to move on to intermediate level.

In my project the goal is to fill this gap and provide educators and students with suitable samples for training their malware reversing skills. The thesis will focus on building a repository for windows binaries, both 32-bit and 64-bit.

The first step in this project is to interview people engage in education and training of malware reversers, both in an academic and business perspective. It is relevant to talk to people training others but also people training themselves. The purpose of the interview is to get a better understanding of how other organizations find potential interesting malware samples, and what the different educators consider an interesting or useful sample. All question will be in the context of creating educational courses or training.

The interview will be semi-structured based upon the questions in this interview guide, with the option of follow-up questions. It is estimated that the interview lasts 30 minutes. We would like to record the audio of the interview to make analysis of the results easier. All recordings will be deleted when my master thesis is delivered and approved.

In order to meet the privacy rules and regulations, we have prepared a consent form. See attached document. You can, of course, at any time revoke your consent and all your data will be deleted.

Best regards,
Aleksander Bjørkhaug
Student

Norwegian University of Science and Technology

Q1: How often do you work with something related to malware reversing?

Q2: When choosing a malware sample for example use, do you most often find new samples or use previously used samples?

- How do you feel about that? If you often use an “old” set, would you want to update?

Q3: What is an educational friendly malware to you?

Q4: When looking for malware for educational training, are there any telltale signs you are looking for?

- Do these differ from entry level education to intermediate?

Q5: Are there anything that would make a malware no-go for use?

Q6: How do you work when you are looking for malware for training? Do you have a specific workflow or mindset?

Q7: Approximately how much time do you use to evaluate and take reference notes for a potential sample?

Q8: Are any of your steps automated? What are your thoughts on usefulness on automated tools? Would you rather analyze yourself?

Q9: Which tools do you use when analyzing PE files?

Are there different tools you use when finding educational malware vs. normally?

Attachment 2: Transcribed interviews

Interview 1:

Interview 1: Company employee 1

How often do you work with something related to malware reversing?

I work with it every day. As it is my role in the company. I do reversing analysis, read reports, create tools or write documentation for an analysis. I'm deep inside the assembly code, and analyzing cryptology functions and reimplementing them, to a tactical level where I analyze the actors and their threats.

When preparing for internal education, do you most often use older, already used or analyzed samples, or do you prefer to use new samples unseen for you before?

I don't do formal education, but I do internal learning for what we do want with a malware analysis. We start with something that I've already analyzed, from a earlier incident. The trainee then does his analysis and I know what he is supposed to find. After that we continue on to new samples that comes through incident response. We analyze the sample together, where I do the analysis and the trainee does the documentation. After that we switch, where I look on as the trainee works and comment.

What do you mean is educational friendly malware?

A sample that mostly does one thing, like exfiltration through either HTTP or DNS, so that the analyzed sample isn't too complex. It's very specific in it's purpose so that the trainee isn't side tracked while working. A sample that exploit one specific technique. The next step would be combinations of multiple techniques, so the trainee has to understand multiple aspects at the same time to understand the malware. Shouldn't be any difference between 32 or 64 bit malware.

When analyzing malware, are there any indicators or telltale signs that tell you this is a educational friendly malware?

Important to find different implementations of techniques. Find and categorize from simple to more complex sample for later use.

Are there any indicators in malware that makes it no-go for educational training?

Malware that does destructive actions on the system, such as deleting files or encrypting files. It can still be used, but the training must be facilitated for it so the analysis actually can “go wrong”. For a introduction course, you wouldn’t give the trainees malware that could destroy their environment.

Malware protected by commercial packers and cryptors would also be to complex to analyze. Or a implementation of VM protections / simulation.

Do you have any different mindset or workflow when you know you are trying to find malware for educational purposes?

For internal education I use earlier analyzed samples, because I have a good overview of what the different samples did, and how I can use them. And I can use previous analysis documentation if for reference. Or I can reimplement techniques in my own written “malware” samples.

Its important that when you create a educational course, that there are no surprises with the samples. You as the educator should know every aspect of the malware.

How much time do you use on one malware sample when analyzing and writing documentation/reference?

Or company has a policy that everything we do must be documented, or it hasn’t been done. We have a structured documentation framework that we work with. There are a lot of different aspect of the analysis that has to be documented, like reimplementation, which takes a long time.

An analysis can take everything from a couple of hours to multiple days , depending on complexity and situation around the analysis.

Are any of your steps automated, or is it mostly manual?

Sample gathering is automated. We mostly look at targeted attacks, so the analysis has to be manual to gather the right information. Some steps are automated like gathering of metadata.

What tools do you use?

There are no difference in tool between education and work. Our most prominent tool is Hex-Rays IDA Pro. Other tools are python, hex or pe editors, and we write our on tools and scripts for processing samples.

Interview 2:

Interview 2: Educator

How often do you work with something related to malware reversing?

Every two weeks at least. Work with education and research.

How are you choosing your samples, do you often use old samples or do you find new samples?

It depends. Starting classes usually use older samples as they are less sophisticated. Further out in the class or more advanced classes get more and more sophisticated malware. Often hunt new samples.

What is educational friendly malware for you?

You cant call one sample educational friendly, but a set of different samples showing different techniques that are used. Gradually becoming more and more advanced. First maybe just C2, then some packing or obfuscation, then inside a another media like Word Document macros.

You're interested in C2 and packing, are there any other tale tail sign that you are looking for?

Depends on the course. Interested in macro and PDF malware. I call it multi media malware. Also like to have ransomware in my courses. IOT malware as well.

Do the sample has to have easy string and low obfuscation, or do you want more difficult samples?

I depends on the level of class. Often use sophisticated malware with many different techniques.

Are the any thing that you think is nogo for a sample?

Trying to avoid DOS malware. They don't offer anything. Don't use many different ransomware because they often does the same.

How do you work when you are looking for educational friendly malware? Any different mindset or workflow? Or do you find interesting samples through your research?

Normally the first chain of finding samples are through research. Next I look at how easily explainable they are. If the malware has no line or reason they are uninteresting. My be used in part. Interested in finding malware that are specialized in doing a special kind of damage. Easier to build a scenario around it for the course.

How much time do you use to write documentation and take reference notes?

Not spending much time vetting the malware. The sample should just take my interest right away. It my take a month or so to make the course content around it, making slides and writing notes. Quite careful in choosing samples because it takes so much time.

Are any of your steps automated?

Only part automated is finding malware, searching though databases. Other than that all other analysis is manual work.

What tools do you use?

Mostly use GHIDRA for reversing. Rest are “normal” reverse engineering or malware execution tools.

Interview 3:

Interview 3: company

How often do you work with something related to malware reversing?

All the time. Primary purpose of my job. Education and preparation is about half of my work hours.

In education, do you use old known/analyzed malware or do you like to use new samples?

Mostly use old stuff that other have prepared, that I've improved somewhat. Although I look for new stuff from time to time.

Would you like to find new samples, or is it alright to use the old set?

Mostly good, but with improvement of the class we look for new samples. We use both inhouse made samples, and wild samples that have been modified.

What is educational friendly malware to you?

Real malware with C2 that is safe (not up or controlled by us). And any other destructive elements are removed or changed. Sanitized real malware.

What are some tell tale signs?

Complexity. Something with a appropriate level of complexity for the class. Look for something that illustrate what I'm teaching.

What kind of samples would give a intermediate reverser?

Obfuscation. More branching code. Maybe fully feature backdoor. Specific functionality that isn't obvious at first glace.

What makes a malware nogo?

Network address that is live and malicious. Any kernel level functionality. Any destructive capabilities, removing files or destroying OS.

Any different mindset or workflow when finding educational malware?

Randomly searching through samples that I've already analyzed to find something appropriate. Can I analyze it myself without any problem. No set workflow.

How much time to use to evaluate and take reference notes?

Vary with complexity. Use samples that have been analyzed before. If its simple it could be like an hour, and more complex maybe a day.

Are some of your steps automated already?

No, not really. All manual analysis.

Is it because you rather would do manual analysis?

Don't know any tools that would to the work, and haven't had the need to do the bolk analysis. Familiar with different samples so know where to find what I need.

What would be useful for you in automated tools?

Detect a packer. Remove heavily obfuscated samples. Detect malware capabilities, filter on those. Downloading files for instance. Filter out size, wouldn't want anything large or with statically linked library. No odd programing languages; no python or go.

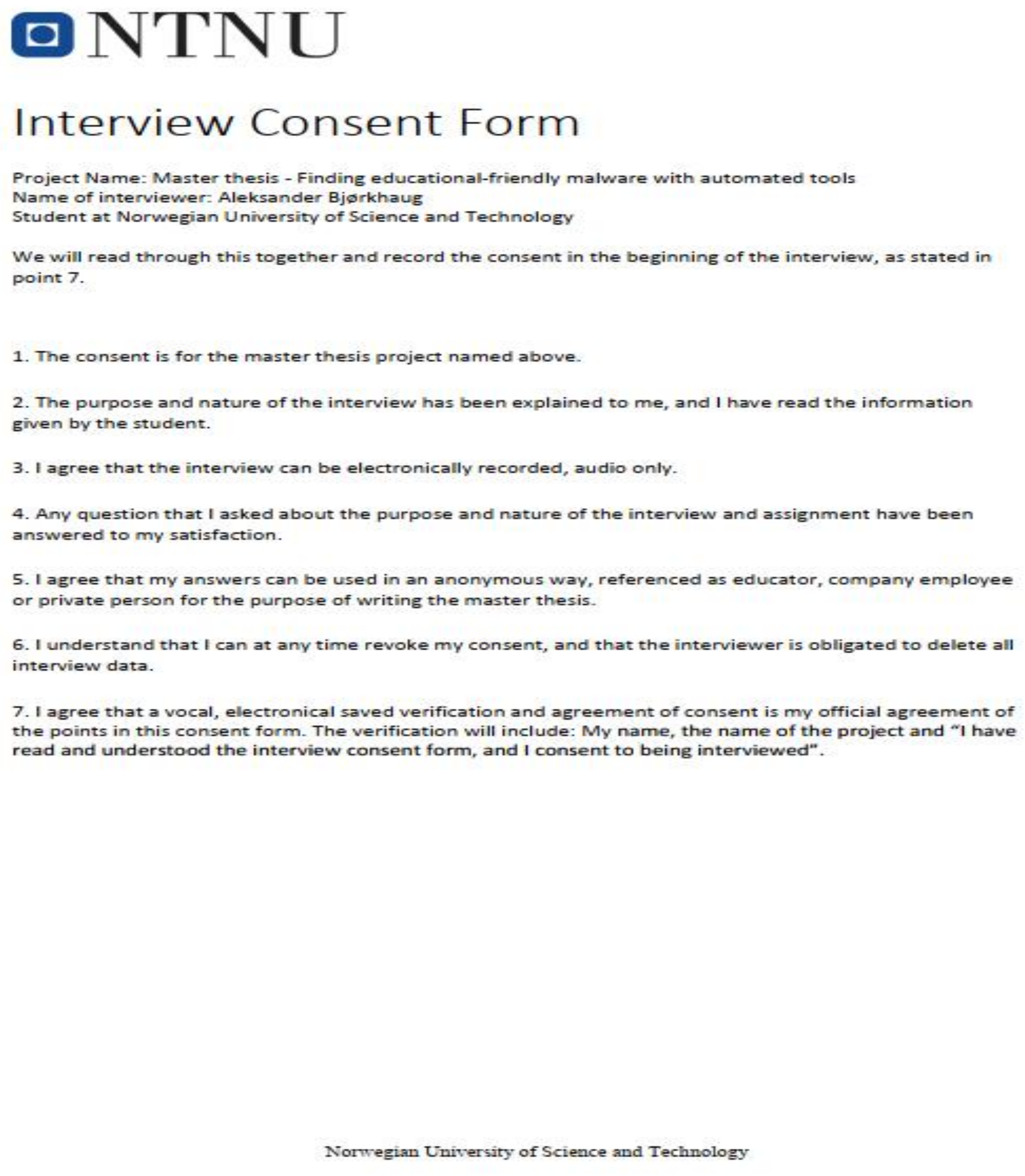
Which tools do you use?

Basic tool, CFF explorer, proc mon, proc exp, IDA Pro, strings, Fakenet, floss, capa, hexeditors.

Would like command line or UI tools?

I like UI.

Attachment 3: Interview Consent form



NTNU

Interview Consent Form

Project Name: Master thesis - Finding educational-friendly malware with automated tools
Name of interviewer: Aleksander Bjørkhaug
Student at Norwegian University of Science and Technology

We will read through this together and record the consent in the beginning of the interview, as stated in point 7.

1. The consent is for the master thesis project named above.

2. The purpose and nature of the interview has been explained to me, and I have read the information given by the student.

3. I agree that the interview can be electronically recorded, audio only.

4. Any question that I asked about the purpose and nature of the interview and assignment have been answered to my satisfaction.

5. I agree that my answers can be used in an anonymous way, referenced as educator, company employee or private person for the purpose of writing the master thesis.

6. I understand that I can at any time revoke my consent, and that the interviewer is obligated to delete all interview data.

7. I agree that a vocal, electronical saved verification and agreement of consent is my official agreement of the points in this consent form. The verification will include: My name, the name of the project and "I have read and understood the interview consent form, and I consent to being interviewed".

Norwegian University of Science and Technology