COMPACK – A Network Based RunPE for Software Piracy Prevention

M.Sc. Internship
Cybersecurity

Saptarshi Laha
Student ID: x18170081

School of Computing
National College of Ireland

Supervisor: Michael Pantridge

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Saptarshi Laha		
Student ID:	x18170081		
Programme:	M.Sc. Cybersecurity	**Year:**	2020
Module:	M.Sc. Internship		
Supervisor:	Michael Pantridge		
Submission Due Date:	17/08/2020		
Project Title:	COMPACK – A Network Based RunPE for Software Piracy Prevention		
Word Count:		**Page Count**	

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on NORMA the National College of Ireland's Institutional Repository for consultation.

Signature: Saptarshi Laha

Date: 17/08/2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

COMPACK – A Network Based RunPE for Software Piracy Prevention

Saptarshi Laha
X8170081

Abstract

Software piracy is a significant concern in today's world. Software developers and software development companies aspire to prevent piracy of their products by incorporating various software licensing mechanisms into it. Additionally, the software is delivered to the customers in a packed state to make reverse-engineering of the same exceptionally complicated and laborious, thereby preventing individuals from accessing the underlying software licensing mechanism and executable code. This paper proposes a mechanism of code delivery which aims to be a strong contender for both traditional and modern packing techniques at providing the desirable amount of security. The discussed method involves handcrafting a portable executable which enforces reliable protection due to its same section entropy, zero performance overhead, same file size, the self-modifying behaviour of the resulting executable file. These properties allow the constructed portable executable to be competent at subverting reverse-engineering tools targeted at unpacking packed executables, while also making manual inspection of the same an arduous toil. It further helps preserve the integrity and confidentiality of the software by retaining the logic of the software licensing mechanism and executable code present underneath, while keeping it away from the prying eyes of the analyst.

1 Introduction

Piracy of software has massively risen over the past decade. This rise has led most developers and development companies to use software licensing solutions and packing of software code, to prevent analysis and subversion of the same by an analyst. The incorporation of such software protection mechanisms in the software, however, has not dampened the spirit of the exceptionally inquisitive members of the technical community, including security researchers and black-hat hackers. They attempt to crack the newly implemented secure packing methods, taking it as a challenge to further their knowledge in understanding of the technology, or rendering the protection mechanism and licensing solution set in place useless, and distribute software for fame and profit, respectively. Thus, there arises a need for the development of new packing and code delivery algorithms from time to time that provides temporary security to the software products and their underlying software licensing mechanisms.

Before understanding the internals and working of the proposed code delivery mechanism, it is essential to have a brief idea regarding the general unpacking process of a packer to understand why it fails to provide sufficient security in today's evolving software piracy scene. This knowledge will enable an analyst to write an automatic unpacker or allow them to unpack the code concealed underneath for manual analysis. The most common mechanism of unpacking an executable in Windows starts with the saving of the register context at the entry point usually with a **PUSHA** instruction. Reserving of a portion of heap memory equivalent to the size of the unpacked **Portable Executable (PE)** sections or higher than it using one or a combination of functions such as **VirtualAlloc** or **VirtualAllocEx** follows if a previously reserved section does not exist in the section table of the PE file. The access protection flags of the reserved portion of the memory or reserved section that maps to the memory in case of a previously reserved section in the PE file, are then changed to Read, Write and Execute using one or a combination of functions such as **VirtualProtect** or **VirtualProtectEx**. The next step includes decryption and decompression of the code and data sections of the actual executable into the reserved memory, loading and linking of the libraries that the original executable imported, and restoration of the register context saved at the entry point using a **POPA** instruction. A jump instruction to the original entry point of the unpacked executable succeeds, executing the unpacked PE file as it would if it was executed without the unpacker performing the previous operations [1].

The information presented above helps an analyst devise their attack strategy against it. In this case, the attack strategy happens to be extremely simple irrespective of the complexity involved in packing the executable. The return value of the **VirtualAlloc** or **VirtualAllocEx** functions is the first address of the newly allocated memory. Additionally, the first parameter of the **VirtualProtect** and the second parameter of the **VirtualProtectEx** functions is the first address of the memory section whose access protection flags require alteration, which happens to be the same address as the one returned by **VirtualAlloc** or **VirtualAllocEx** functions if it exists and precedes this function. Analysing the disassembly generated by the packed executable and setting the appropriate breakpoints, an analyst can quickly unpack a packed executable for analysis if it uses the unpacking strategy mentioned above, and, can eventually also develop a custom toolkit to do so automatically, massively reducing the analysis time involved. The study of this scenario poses a challenging yet crucial question regarding the possibility of changing the source of code delivery so that the attacker cannot access the actual code underneath while the execution of the code occurs as and when required such that manual or automatic unpacking is not possible. The method of code delivery proposed by this paper attempts to answer this question in the Research Methodology section of the document.

An insight into the Related Work section reveals that the methods currently used for packing do not hold good against the current advances in packer detection and automatic or manual unpacking of the same. Hence, there is a need to present a novel idea in PE file executable code delivery which incorporates techniques from various advances in general computing such as self-modification of code, code encryption, handcrafting of PE files to achieve the desired security. This security achieved is practical only if it does not come bundled with its fair share of performance overhead, and thus, ensuring the same with the utmost care is also an addition to the initial proposal by the paper. It is essential to understand the working of currently used packers and how the security enforced by them meets their demise at the hands of current date unpackers, discussed in the Related Work section of the document, before proceeding with the discussion entailing the proposition presented by this paper. This layout is the intended flow of the document to ensure that the reader is at a better position to grasp and conceptualise the techniques involved to gain an in-depth understanding of the subject matter. The understanding gained is, in turn, is crucial in suggesting a new executable code delivery scheme and its extensive analysis for potentially overlooked flaws outlined in the later sections of the document. In addition to the content mentioned above, a few more critical topics are also present in the Related Work section of the document which helps in the formulation of the proposed code delivery method, while also serving as an informative read.

2 Related Work

This section includes discussions of various types of packers and their unpacking methods followed by various automatic unpackers and their working. The first subsection dedicates itself to making the reader wary of the types of packers present and their unpacking methods. In contrast, the following subsections scrutinise the approach of detection of packing in a PE file, identifying the type of packing present, and the methodology involved in unpacking the same, done by different automatic unpackers. It further highlights the loopholes that each of these implementations has, which then gets leveraged to present the proposal for a new code delivery method. Practically, a security researcher would only be interested in sharing their findings in case the packing of malware utilises a packer. Hence, most of the automated unpacking tools mentioned here find its use in malware analysis; however, they can find their use for unpacking commercial software as well, packed using that specific packer or packing method.

Later subsections discuss critical information regarding miscellaneous topics which find their utilisation in the implementation of the proposed packing method. Generally, these topics do not directly lead to the crafting of a new code delivery method, but aid in the process of refining the same against current advances in unpacking and analysis methods by making them automatic unpacking and analysis resistant. It also serves as an excellent primer for learning questionable techniques that usually get exploited by black-hat hackers or security researchers in performing non-conventional tasks that are extremely difficult or impossible to perform using regularly used methods of development. Hence the reader must be wary of the techniques presented in these subsections to evaluate the advantages and limitations that each of these fancy methods brings along.

2.1 Types of Packers

This section discusses the different types of packers categorised based on their complexities, as researched by Ugarte-Pedrero et al. [2]. This information is crucial to understanding the discussion of the automatic unpacking methods that follow. The division of packers based on its complexity results in six different types as described below:

1) **Type 1 Packers** – These types of packers perform a single layer of unpacking before transferring the control flow to the unpacked code.
2) **Type 2 Packers** – These types of packers contain multiple unpacking layers. Each unpacking layer is responsible for unpacking the subsequent unpacking routine. On the entire reconstruction of the actual program, the control flow transfers to the unpacked code for execution.
3) **Type 3 Packers** – These types of packers are like the previous types of packers with the exception being that the unpacking does not follow linearly but gets organised in a more complex topology that includes loops. Due to this structure, the original packed code may not reside in the last unpacking layer. Instead, the last layer generally contains various anti-debugging mechanisms, integrity checks or part of the obfuscated code of the packer. A tail jump still exists to separate the packer code from the actual program code.
4) **Type 4 Packers** – These types of packers are single-layered or multi-layered packers that have a part of the packer code, which is not responsible for unpacking, interleaved with the execution of the original program. There, however, exists a precise moment in time when the entire actual program code gets unpacked in memory. The tail jump responsible for the transition from packer code to program code can be challenging to locate.
5) **Type 5 Packers** – These types of packers have the unpacking code jumbled with the original program code, such that the layer containing the original code has multiple frames, and the packer unpacks them one at a time. Though these packers have a tail jump for the transition, only one frame of code gets revealed, and thus, there is a requirement for a snapshot of the process memory after the execution of the entire program for analysis of the original code.
6) **Type 6 Packers** – These types of packers are the most complex packers as they only unpack a single fragment of the original program at any point of time.

The next few sections focus on different approaches to automatic unpacking and analysis of code, as discussed by different researchers. These approaches play a crucial role in making the reader understand the unpacking process carried out by different automatic unpackers to help them comprehend the motivation behind the confident design choices that exist in the code delivery method proposed by the paper, before introducing them to its nitty-gritty details.

2.2 PolyUnpack

This subsection discusses a common malware unpacking mechanism proposed by an unpacker named PolyUnpack [3]. The method of unpacking used by Royal et al. in their PolyUnpack project includes disassembling of the program to identify code and data in the first step of the process. The executable instructions gathered in this process helps form a set of original instructions of the program. Next, it executes an instruction in the program, saves the current value of the instruction pointer and performs an in-memory disassembly starting from the current value of the instruction pointer until it encounters a non-executing instruction. Finally, it performs a check to verify if the instructions gathered in the previous phase are a subsequence of the original instruction set. In case instructions gathered do not happen to be a subsequence of the original instruction set, then the unpacked code starting from the current instruction pointer is returned. Otherwise, this process continues until the last instruction of the executable is extracted.

The algorithm mentioned above is ingenious but fails to work in some instances including if the executable detects that it is being executed by PolyUnpack, as it can alter its flow of action thereby preventing automatic unpacking or leading to unpacking of unintended code. Hence, it is vital to keep this property in mind when suggesting the design for the code delivery mechanism. Apart from this, PolyUnpack also fails on step-by-step code execution as suggested by the future work of this paper as it extracts unpacked code starting from the first instruction change and exits. The algorithm can be slightly tweaked to run for the complete executable irrespective of an instruction change for better results. However, this will result in the extraction of N unpacked code, where N is the number of instruction changes that occur during the execution of the executable file. One further optimisation is possible, which is, during the extraction of the N unpacked code, the standard instruction subsequence undergoes omission and the changed instructions stitched together to form the actual unpacked code. This modification will eliminate the extraction of N unpacked code, and instead extract only once irrespective of the number of instruction changes occurring during program execution. This modification is helpful against packers which have multiple instruction changes during program execution. It is, however, essential to note that these modifications are not present in the original implementation or proposal of the same but are suggestions to cover up some of the flaws of the unpacker. Even in this case, if the instruction subsequence completely changes, then the unpacker will output garbage data that cannot undergo stitching to form a valid portable executable file.

The next subsection describes the working of another malware unpacker which serves as an improvement over PolyUnpack in detecting and extracting packed code and incorporates some of the functionalities suggested to make PolyUnpack better while taking an entirely different approach algorithmically.

2.3 Renovo

This subsection discusses another automatic unpacking algorithm proposed by Kang et al. and commonly known as Renovo [4]. Although the approach of Renovo is substantially different from PolyUnpack discussed in the previous subsection, they do have their fair share of similarities as well. Renovo works by generating a memory map whenever an executable gets loaded onto memory. This generated memory map is labelled the clean state. On encountering a memory write instruction by the executable, the corresponding memory region gets marked dirty. If an instruction pointer jump occurs to any of these newly generated regions, it gets marked as the original entry point for the unpacked code and extracts the code present in the memory region. Renovo, however, adds functionalities to unpack multi-layered packing by analysing the code and data sections in the newly unpacked code and recursively performing the previous operations until the final unpacking occurs.

This approach takes into consideration the possibility of multi-layered packing to exist, unlike PolyUnpack. However, Renovo, just like PolyUnpack also extracts N copies of data in case of N different jumps to memory regions marked dirty in memory. Just like in the previous case, this methodology of unpacking is rather inefficient against step-by-step code execution on code delivery and also is ineffective against other anti-reversing methods implemented such as code obfuscation and detection of Renovo unpacking the code by the executable and modifying its behaviour accordingly to thwart analysis. Thus, even though Renovo is a significant upgrade from PolyUnpack, it is still not the best or most efficient approach to unpacking hidden code.

The discussion of the next subsection entails the use of heuristics as well as a statistical model to aid the unpacking process of the packed executable file. This method is far more complicated than the currently reviewed methods to unpack packed executables while having a lower performance overhead and executing faster than the previously discussed methods under general circumstances. There, however, is another unpacking mechanism called OmniUnpack [5] that deserves mentioning before proceeding with this transition. The reason why this mechanism lacks a detailed explanation in the current text is that it is more focused on analysing malware rather than being generalised to any packed executable, unlike the previously mentioned methods, while following a similar methodology as Renovo in the background to help facilitate the analysis of packed code during its unpacking stage. It is also important to mention that OmniUnpack is more efficient than Renovo due to choices made by the authors that allow for lesser performance overhead.

2.4 Eureka

This subsection dedicates to the analysis of another unpacking algorithm proposed by Sharif et al. known as Eureka [6]. Eureka combines heuristics-based and statistical-based unpacking to unpack hidden code from a packed executable during its runtime, along with child process monitoring. The heuristics-based unpacking used by Eureka waits for the termination of the running process by the interception of the **NtTerminateProcess** system call, for dumping a snapshot of the program's virtual memory address space. This approach assumes that since the program has unpacked and executed, the unpacked instructions are present in the memory. Besides, it also takes into consideration the creation of child processes by the executing process to aid in its unpacking or perform execution of a crucial subroutine and thus also waits for the interception of the **NtCreateProcess** system call to start monitoring the child process invoked and apply the same heuristics principles for its analysis. The statistical-based unpacking involves modelling the statistical properties of the unpacked code. This modelling bases itself on two assumptions which are – specific opcodes, usage of registers and instruction sequences are more prevalent than others in executables, and, the volume of code increases as the packed executable unpacks itself. When a particular section of memory defies the first assumption or behaves according to the second assumption, that memory section gets dumped.

Eureka is undoubtedly a much more complex unpacking algorithm compared to the previously discussed ones. It additionally has a negligible performance overhead compared to them as a result of the lack of continuous processing of the executing instructions of the program. Instead, it depends upon the invocation of one of the two pre-defined system calls or an anomaly in its statistical predictions, to extract the unpacked code from the memory region. Although Eureka is far more advanced in its unpacked code analysis patterns, it still fails to hold up against step-by-step execution of instructions or analysis of the execution environment by the packed executable to modify its behaviour, as mentioned by the author. A simple modification involving the interception of the **VirtualAlloc**, **VirtualProtect** and other calls with similar functionalities can, however, make this unpacker much more potent than it is. However, this would add to the performance overhead as these functions often invoked by non-packer based executables and could result in erroneous results and findings due to the same reason.

In the next subsection, the assessment of another automatic unpacking mechanism that utilises logging of every instruction executed and further parsing of the log to extract a partial unpacked file occurs. Thus, the next subsection marks the onset of a new generation of automatic unpacking algorithms, and hence these algorithms are more complicated compared to the previously discussed ones, apart from being modular in design. This modularity means that the algorithm depends on multiple tools at multiple points in the extraction process to assist with extracting the final unpacked code. However, before proceeding to the next subsection, there is another unpacking method that deserves special mention at this point, known as WaveAtlas [7]. This unpacker is however not discussed in much detail as this automatic unpacker works based on two hypothesis – the executed instructions starting from the initial memory image till the second to last memory image serve to protect the concealed program, and, the executed instructions in the last memory image contain the actual program. These assumptions do not hold in case of step-by-step execution of programs, and the paper also mentions the partial validation of the hypothesis, which leads to an uninteresting read. Hence, the creation of a separate subsection for the extensive analysis of the same did not seem worthwhile.

2.5 Mal-Xtract, Mal-XT and Mal-Flux

This subsection deals with the introduction of a rather complicated set of methods for hidden code unpacking. The first unpacking method discussed is proposed by Lim et al. and known as Mal-Xtract [8]. The methodology proposed utilises PANDA to execute the packed executable while recording the entire system emulation in a log file. The recordings in the log file get replayed to detect the memory addresses that got written to during the execution of the program, and, the written memory addresses and their adjacent memory addresses are marked as written. An empirical approach then gets used to determine the threshold value based on the number of instructions written for consideration of the memory sections for being dumped. Finally, Volatility is used on the physical memory dump to extract the sections that meet the threshold value to try and reconstruct the unpacked executable code. Further, the extracted section data gets parsed using IDA Pro, and their corresponding mnemonics gathered for comparison with the original executable file using a diff tool. The similarity percentage and the entropy then get used to determine the presence of packing in the executable.

Although this approach uses multiple toolkits and employs sophisticated methods for carrying out partial unpacking and validation of the presence of packing in an executable file, it still has its fair share of shortcomings. Before mentioning these shortcomings, it is crucial to highlight the improved methods that this automatic unpacker employs including logging of the entire emulated execution, which is very useful in analysing and extracting hidden data from packed executables. However, there is no check performed to verify if the extracted instructions at every stage get executed or not, leading to the generation of copious amounts of garbage data which thwarts analysis. Besides, the empirical approach used for the calculation of the minimum, and maximum threshold values also depend on the number of unpacked instructions at every given instance. It does not take much effort to dupe this mechanism into working incorrectly by merely extracting the same amount of code at any given instance of unpacking or code execution. Finally, the reconstructed executable cannot execute directly due to the Import Address Table (IAT) being damaged or missing, which further makes this automatic unpacker inept.

The following unpacking method of interest is an advancement from Mal-Xtract. Lim et al. propose it, and it is known Mal-XT [9]. Mal-XT replicates the previously discussed method but adds the functionality to track the execution of instructions. This addition further helps in separation of executable code from garbage code and data. In other words, the extraction procedure for hidden code only extracts the data from memory sections that undergoes execution. Although this additional enhancement to the initially proposed algorithm of Mal-Xtract helps in neglecting sections of code that do not get executed, it still does not take into account the presence of logically opposite pairs of code or ambiguous code being present in the memory section that executes. Besides, multiple executions of a program are required with different parameters to follow all the branches and paths to derive the complete executable, unlike the previously discussed method. Also, just like the previously discussed method, this method results in the reconstruction of an unpacked executable file that cannot execute directly due to the IAT being damaged or missing.

Next, the discussion of the unpacking method proposed by Lim et al. is known as Mal-Flux [10] is considered. This method, just like Mal-XT, is an advancement from Mal-Xtract. It also analyses the same memory addresses for being rewritten with code apart from monitoring new memory or section writes and rewrites. This property makes it much more advanced than the previously discussed automatic unpackers. This additional recorded information gets utilised in the reconstruction of the executable file at the end of the process, although just like the previously discussed unpacking methods, this method also results in the generation of an executable that cannot execute directly due to the IAT being damaged or missing. Besides, the most impactful disadvantage of Mal-Flux is the fact that it is hugely performance heavy, apart from not tracking the execution of instructions to separate actual code from garbage code and data which is crucial to the refining of the results.

It is of great interest to mention that all these automatic unpacking techniques mentioned above use additional methods for activities such as analysis of malware or segregation of the malware executables and the benign executables. However, as these methods do not aid the formulation of the code delivery mechanism proposed in this paper, they are omitted. Additionally, all the previous subsections were related to dynamic unpacking methods of packed executable files. The next subsection deals with the static unpacking of packed executable files which follows a different approach than the ones currently employed.

2.6 Static Unpacking

This subsection deals with the static unpacking of packed executables, as proposed by Coogan et al. [11]. Unlike the previously discussed methods, this method of hidden code unpacking does not depend on the execution of the packed executable but rather depends on disassembly, static code analysis, alias analysis, possible transition point detection, static unpacker extraction and static unpacker transformation to achieve the desired result. It starts by generating a static disassembly of the code, followed by identification of basic blocks to construct a control flow graph of the disassembled code. Next, it performs binary-level alias analysis to identify the memory addresses where indirect memory operations occur. The results generated in the previous step get used to determine potential transition points where the unpacked code gets executed by the transferring of the control flow. The operations performed following any individual path corresponding to any unique transition point is analysed to identify the memory locations that may be modified apart from performing backwards static slicing to identify the static unpacker. This static unpacker then gets analysed to perform various transformations such as detection and removal of code protection mechanisms on it. Finally, the statically analysed and modified code gets executed to result in the unpacking of the hidden code.

This method of unpacking attempts to render any defence mechanisms set in place in the unpacker code useless and, further, execute the unpacker to extract the unpacked code. This technique finds utilisation in single-level unpacking, but if the packing of code is present in several layers, then each layer would only unpack the subsequent layer of code. Additionally, if there is no code to analyse statically in the binary, then such methods fail to work practically due to the lack of analysis parameters on which the static unpacking depends. Due to these limitations, this type of automatic unpacker is not feasible for analysis of densely packed or radically instrumented binaries. This subsection marks the end of the discussions of various automatic unpacking techniques. In the following subsection, the discussion entails the importance of entropy for detection of packed executables. This information will help the reader understand the need behind keeping the entropy low, similar or the same in the resulting executables proposed by the paper for code delivery.

2.7 Entropy-based Packer Detection

This subsection deals with introducing the reader to the concept of entropy and further explains how it acts as a metric for packer detection. The entropy of a block of data describes the amount of information it contains, and its calculation is as follows:

$$H(x) = -\sum_{i=1}^{N} \begin{cases} p(i) = 0, & 0 \\ p(i) \neq 0, & p(i) log_2 p(i) \end{cases}$$

where **p(i)** is the probability of the **i**$^{\mathbf{th}}$ unit of information in event **x**'s sequence of **N** symbols. For packed program analysis, the unit of information is a byte value, **N** is **256**, and an event is a block of data from the packed program [12].

Compressed and encrypted code and data result in the calculation of high entropy values, whereas the general code and data result is much lower entropy. Hence identification of packed data is rather simple based on the calculation of its entropy value. Bat-Erdene et al. also propose a method to utilise entropy analysis for detection of multi-layer packing in executables [13]. This analysis calculates the entropy of all the unpacked sections recursively during to unpacking process to determine if any of the unpacked sections further contain packed code. If the entropy values for a section are above the threshold limit, the section is considered packed, and further unpacking of the section follows.

The information presented above is critical to the formulation of the method of code delivery proposed by this paper. However, a few more primers on essential topics are required to formulate the same altogether. The next subsection dedicates itself towards introducing the reader to the concept of self-modification of code and its linearization, which is one of the most critical concepts that need special mention due to the massive role it plays in the formulation of the proposed code delivery method by this paper.

2.8 Self-modifying Code

This subsection deals with the introduction of self-modification of code, while also explaining how the linearization of the same occurs using State Enhanced – Control Flow Graphs as proposed by Anckaert et al. [14]. Self-modification of code refers to the modification of any piece of the code within a running program, by the same running program for altering the operation it performs when it encounters that code. Data transfer operations in a writable and executable section in memory can facilitate self-modification of code to alter its behaviour. Malware writers generally exploit this method, but it can also lead to the generation of dynamic code in a specific section in memory and executing it to perform various fancy operations. In usual programs that do not exploit this behaviour, a control flow graph is constructible, which shows the various execution paths an executable can take during its execution. However, a general control flow graph is not constructible for a program utilising self-modification of code.

Anckaert et al. propose a method to generate a model of control flow graph that takes into consideration the different states the program is at and details the subsequent transition or transformation related to it. Unfortunately, this linearization using State Enhanced – Control Flow Graphs is only possible if the targets of indirect control transfer within a program are known. If they are not known, then the State Enhanced – Control Flow Graph model has to take into consideration every byte getting altered at any particular time within the program and the control flow getting passed to all of them which makes the problem infeasible to solve. Thus, in other words, if the addresses where the instructions are getting altered and executed are not constant, then it is challenging for this model to linearise the disassembly generated. In the case of the implementation presented by this paper, the VirtualAlloc or VirtualAllocEx type of functions find their utilisation, which in turn allocate a size of memory in the heap section starting from a random address, and thus the previous model cannot be used for analysis. Additionally, the control transfers are pseudorandom and can occur at any point leading to further complexities.

This subsection marks the end of the Related Work section. The techniques learnt in this section help in some or the other way to build a robust code delivery mechanism. A few additional topics were worthy of mention, but since they do not require extensive analysis for the formulation, they were ignored and are instead lightly touched upon in the subsections of the next section for the sake of brevity.

3 Research Methodology

As noted from the Related Work section, the packing methods present in today's world are largely inefficient with the exception of step-by-step unpacking packers. Hence there is a need to formulate a novel code delivery mechanism where the executable code which can be altered or patched does not reside on the computer of the analyst locally and is only present during its execution. One way to tackle this problem is to completely make the application a server sided application but this approach hinders with multiple applications where it is necessary for a particular piece of executable code to be present on the client's machine to perform certain tasks. Additionally, this approach makes organisations incur heavy costs for server operations due to the additional load in implementing such a mechanism. It is also noteworthy to mention that the valuable resources present on the client's computer would find zero use in such an approach and thus it can be considered a waste of computational resources.

Hence there is a valid reason to propose an idea of secure code delivery to the client's computer so that the computational resources on the client's system are utilised to generate the results while keeping the executable code away from the prying eyes of the analysts to prevent piracy and other sorts of malicious operations or inspection. A way in which this can be achieved is by sending an executable program to the client with its **.text** section completely stripped so that there is no way to analyse the code or data manipulations that occur during execution locally using a disassembler or a debugger, thus preventing software piracy from the ground-level of the design of code delivery. The **.text** section can then be provided by the server as and when required to the client which will be patched into memory using memory manipulation operations by a loader executable.

The proposed and implemented method uses a loader to utilise the **VirtualProtectEx** function to perform in-memory section access protection flag modification to utilise th**e WriteProcessMemory** operation to write the executable code upon execution. This code is received from the server as and when required by the client. Additional benefits to this approach lie in the ambiguous functionality of the code received. This allows for a programmer to craft an executable with a wide array of functionalities and present only a following portion of the code to client depending on the functionality requested by them. Further details regarding the approach is discussed in the Implementation section of the document.

The evaluation criteria for such an approach plays a crucial role in determining the feasibility of the approach. In this case, the criteria for evaluation considers the size of the resulting stripped PE file, the size of the code emitted by the server to the client, the section entropy of the resulting stripped PE file and the execution time of the PE resulting stripped PE file upon receiving the executable code from the server. All these criteria are discussed in detail in the Evaluation section of the document.

4 Design Specification

The proposal mentioned here has been implemented only on the Windows operating system in both x86 and x64 environments and work for executable files with five sections that are considered but can be extended to more sections by simply modifying the code. Additionally, it is also important to mention that the section name checks are not performed and hence the .**text** section is assumed to be the first section of the PE file followed by the **.rdata** section, **.data** section, **.rsrc** section and the **.reloc** section, as is the case with MSVC compiled binaries. However, this code too can be changed to look for section names before performing any operation specific to a section.

Another consideration that is taken into account in this case is that, the building of the executable is only possible through the MSVC or Visual Studio compiler. It should ideally be able to compile with the MinGW compiler that is present for the Windows operating system but the same is not tested as a part of the project. Linux environments and other OS are not supported due to the difference in native functions that get utilised in this project. The functions utilised in this project are Windows specific and execute on Windows XP+ operating system, meaning it executes on any operating system that is Windows and has a version higher than that of Windows XP.

The design of the code delivery mechanism also takes into account the presence of a server and a client; however, the same device can act as the server and the client for the testing purposes using the loopback address. Finally, the last consideration in the design lies in the zeroing out of the .**reloc** section of the executable file and the zeroing out of the **number of relocations** and **pointer to relocations** field in the section header of all sections, leading to the executable loading at the same address on every execution. This approach needs future work as suggested in the Future Work section of the document to enable the executable to load in any address and maintain the security in terms on ASLR (Address Space Layout Randomisation).

5 Implementation

The implementation of this project consists of two major parts – one being the 2FileCreatorServer.exe and the other being RunPE Client.exe. The first executable is written in C using Windows specific headers and functions and is responsible for carrying out three operations – stripping the deliverable executable off the **.reloc** and the **.text** section, dropping two files – the RunPE Client.exe and the resulting stripped executable file, and hosting the **.text** section of the stripped file on a particular port for the client to connect to, so that it can send the data over for execution by the client.

The second part of the project deals with the RunPE Client.exe which is also written in C using Windows specific headers and functions and is essential as it performs multiple major operations. It is responsible for launching the stripped executable in suspended state, changing the access protection flags corresponding to the **.text** section of the stripped binary to read, write and execute, writing the received executable code from the server to the .**text** section of the file in memory, changing the access protection flags to their previous state and resume the process so it can undergo normal execution. The output generated using these two files result in a normally working executable file as it would in case if the code was present locally than being sent over by the server.

The RunPE Client.exe achieves all of these operations by first calling the **CreateProcess** function to launch the stripped binary in the suspeneded state, then changing the access protection flags of the **.text** section of the executable file in memory to read, write and execute, writing the received code to the process memory at the location of the **.text** section, changing the access protection flags of the **.text** section in memory to its previous state and then using an undocumented API call to **NtResumeProcess** to resume execution of the process.

Both the 2FileCreatorServer.exe and RunPE Client.exe utilise the **WS2_32.lib** to create raw sockets for establishing a TCP connection. The TCP connection is essential out here as TCP connections guarantee the delivery of packets unlike an UDP connection and this property is abused to guarantee the delivery of code to the requesting client. Another consideration that needs mention here is the data limit of TCP is 65535 bytes which is more than enough to contain the entirety of the **.text** section even for very large executable files.

6 Evaluation

The analysis of the experiments performed are mentioned in this section in detail. There are four experiments that get performed and all four result in interesting results.

6.1 File Size, File Section Entropy, Execution Time

After extensive experimentation it is found out that the size on disk remains similar for both the non-stripped and the stripped binary, which is a plus point to the project as even though there is no size reduction, there is no executable code present to be analysed by automatic unpackers, code analysers, debuggers or disassemblers.

It is also found that the file section entropy for the .text section and the .reloc section is zero compared to their higher old value leading to marking the file as not packed or encrypted under any circumstance irrespective of the delivery of executable code that comes in from the server.

The execution time after extensive testing remains similar as the same instructions run even after the delivery of the code on the actual hardware/CPU of the client if the execution time of the loader is ignored and the delivery time of the code from the server is ignored. Both of these values are minimalistic and hence can be ignored apart from the fact that the code is compiled natively which makes the process faster than an interpreted language or VM based language.

6.2 Size of Code Emitted by the Server

The size of the code in this case is the size of the **.text** section of the original PE file. The TCP data field supports up to 65535 bytes of data which results to 0xFFFF in hexadecimal meaning that the **.text** section can be up to 0xFFFF in size which is very unusual. Additionally, since its barely 64KB of data, the transmission of the **.text** section can occur very fast with today's evolving internet speeds.

6.3 Entering of the Wrong Encryption Key

When hosting the **.text** section of the PE file, the server encrypts it with an encryption key. This encryption key needs to be provided by the client to decrypt the binary data. On entering the wrong decryption key, the wrong instructions get written to memory. This leads to invalid execution of instructions which do not result in the actual expected output of the binary, thereby adding another layer of protection in the code delivery mechanism.

6.4 Polymorphic Code

Another important usage of this kind of code delivery mechanism lies in the fact that the same code can be used to provide multiple functionalities depending on the requirements of the client. This is because a modified version of the code can be provided to the client on one request which may differ from another request, all while maintaining the same logic and code underneath. Additionally, since the original .text section of the file that is with the client is of the same size, there is no patches or updates that need to be made on the client side in order to execute it giving more room to the programmers to incorporate multi-dimensional functionality to their projects.

6.5 Discussion

Having mentioned the outcomes of all the above experiments, it is essential to point the downsides to such a method. The biggest downside is the fact that even though a static disassembly is not possible, a memory snapshot can be taken, and this can be analysed or recorded to promote software piracy thereby completely rendering the code delivery mechanism in place useless. The other downside that lies in the project is that, if the executable cannot load at the address 0x401000, it will quit as the .reloc section along with the relocation values for each section is completely nulled out. Both downsides make the project seem extremely inefficient but one of them can be tackled easily. However, the solution to both of them are discussed in the next section.

7 Conclusion and Future Work

The research done here provides an invaluable way to deliver executable code to the client with its own share of advantages and disadvantages. However, in general the advantages outweigh the disadvantages making it an efficient solution. The methods applied in the implementation already tackle most of the problems that are present in today's world in regard to software piracy. However, some of the downsides mentioned in the Discussion section of the document might make the entire project seem pointless. This is where additional work is required. With the plethora of memory based RunPEs that are available, it is not difficult to replicate a code to perform the relocations as necessary for the PE file and hence this is considered an easy task.

The actual future work lies in step-by step execution of code so that a memory snapshot cannot be taken for the process under consideration. The implementation of this has already been done by me up to 99% and is still pending due to a misalignment in the ESI registers in one of the operations towards the end and adding the same to this adds a totally new perspective towards secure code delivery and execution and thereby prevents piracy close to cent percent.

References

[1] W. Yan, Z. Zhang, and N. Ansari, “Revealing Packed Malware,” IEEE Secur. Privacy Mag., vol. 6, no. 5, pp. 65–69, Sep. 2008, doi: 10.1109/MSP.2008.126.

[2] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, “SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers,” in 2015 IEEE Symposium on Security and Privacy, San Jose, CA, May 2015, pp. 659–673, doi: 10.1109/SP.2015.46.

[3] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, “PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware,” in 2006 22nd Annual Computer Security Applications Conference (ACSAC’06), Miami Beach, FL, Dec. 2006, pp. 289–300, doi: 10.1109/ACSAC.2006.38.

[4] M. G. Kang, P. Poosankam, and H. Yin, “Renovo: a hidden code extractor for packed executables,” in Proceedings of the 2007 ACM workshop on Recurring malcode - WORM ’07, Alexandria, Virginia, USA, 2007, p. 46, doi: 10.1145/1314389.1314399.

[5] L. Martignoni, M. Christodorescu, and S. Jha, “OmniUnpack: Fast, Generic, and Safe Unpacking of Malware,” p. 10.

[6] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, “Eureka: A Framework for Enabling Static Malware Analysis,” in Computer Security - ESORICS 2008, vol. 5283, S. Jajodia and J. Lopez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 481–500.

[7] J. Calvet, F. L. Lévesque, J. M. Fernandez, E. Traourouder, F. Menet, and J.-Y. Marion, “WAVEATLAS: SURFING THROUGH THE LANDSCAPE OF CURRENT MALWARE PACKERS,” p. 7.

[8] C. Lim, Y. Syailendra Kotualubun, Suryadi, and K. Ramli, “Mal-Xtract: Hidden Code Extraction using Memory Analysis,” J. Phys.: Conf. Ser., vol. 801, p. 012058, Jan. 2017, doi: 10.1088/1742-6596/801/1/012058.

[9] C. Lim, Suryadi, K. Ramli, and Suhandi, “Mal-XT: Higher accuracy hidden-code extraction of packed binary executable,” IOP Conf. Ser.: Mater. Sci. Eng., vol. 453, p. 012001, Nov. 2018, doi: 10.1088/1757-899X/453/1/012001.

[10] C. Lim, Suryadi, K. Ramli, and Y. S. Kotualubun, “Mal-Flux: Rendering hidden code of packed binary executable,” Digital Investigation, vol. 28, pp. 83–95, Mar. 2019, doi: 10.1016/j.diin.2019.01.004.

[11] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, “Automatic Static Unpacking of Malware Binaries,” in 2009 16th Working Conference on Reverse Engineering, Lille, France, 2009, pp. 167–176, doi: 10.1109/WCRE.2009.24.

[12] S. Cesare and Y. Xiang, “Classification of Malware Using Structured Control Flow,” Parallel and Distributed Computing, vol. 107, p. 10, 2010.

[13] M. Bat-Erdene, T. Kim, H. Park, and H. Lee, “Packer Detection for Multi-Layer Executables Using Entropy Analysis,” Entropy, vol. 19, no. 3, p. 125, Mar. 2017, doi: 10.3390/e19030125.

[14] B. Anckaert, M. Madou, and K. De Bosschere, “A Model for Self-Modifying Code,” in Information Hiding, vol. 4437, J. L. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 232–248.