universitätbonn

Memory Dumping for Fun and Profit

Reverse Engineering mit Hilfe von Memory Dumping unter Windows

Felix Bilstein

Bericht zur Projektgruppe

Prof. Dr. Peter Martini
Institut für Informatik IV
Arbeitsgruppe für Kommunikationssysteme
Rheinische-Friedrich-Wilhelms-Universität Bonn

Zusammenfassung

Schadsoftware ist eine immer wiederkehrende Bedrohung in informationstechnischen Systemen. Beide Seiten, sowohl die Forschung in der IT Sicherheit wie auch Angreifer von IT-Infrastruktur entwickeln fortwährend neue Techniken um Systeme zu kompromittieren bzw. zu verteidigen. Malware operiert versteckt und wird zunehmend in einem immer professionelleren Rahmen entwickelt.

Dieses Projekt verfolgt die Zielsetzung, ein weiteres Werkzeug in der Analyse aufzuzeigen, mit dem man den Prozess des Reverse Engineering vereinfachen kann, indem neben der Executable und bösartigen Dateien auch der virtuelle Speicher des jeweiligen Prozesses analysiert wird. Dadurch soll eine zuverlässigere Analyse möglich werden, da viele Tarntechniken das statische Analysieren der Executable erschweren oder gar unmöglich machen können.

Inhaltsverzeichnis

1 Einleitung

Dieses Projekt soll einen Einblick in die Analyse von Malware mittels Memory-Dumping geben. Während man eine Portable Executabe (PE)-Datei grundsätzlich statisch analysieren kann, ist Malware oft obfuskiert und gepackt. Dadurch kann es unmöglich werden, die Datei statisch zu analysieren. Da die statische Analyse in der Regel aber sehr viele Informationen in wesentlich kürzerer Zeit als die dynamische Analyse kenntlich macht, ist es oft hilfreich beim Reverse Engineering den Speicher von Programmen zu analysieren. Dadurch können viele Tarntechniken umgangen werden, da das Programm, um erfolgreich ausgeführt zu werden, sich immer zuerst selbst entpacken oder entschlüsseln muss.

In Kapitel 2 werden einige grundlegende Begriffe vorgestellt. So wird zum Beispiel der Begriff Schadsoftware allgemein definiert und es werden verschiedene Analysetechniken vorgestellt, die statische und die dynamische Malware-Analyse. Kapitel 3 umfasst eine kurze Einführung zum Thema Tarntechniken von Schadsoftware. Auf diesen wird in den folgenden Kapiteln aufgebaut und die vorgestellten Techniken werden detaillierter analysiert. Mit Kapitel 4 endet dann die Einführung in die Grundlagen zum Thema Malware-Dumping. Vorgestellt wird das PE-Format, in dem die meisten kompilierten, nativen Windowsanwendungen und damit auch Malware vorliegt.

Kapitel 5 behandelt das Memory-Dumping, insbesondere Fallbeispiele zum Thema Unpacking, Analyse von Speicher als zusätzliches Werkzeug zwischen der reinen statischen und dynamischen Analyse, sowie Wege um Speicheranomalien aufzuspüren, die üblicherweise im Zusammenhang mit Schadsoftware bekannt ist. Kapitel 5 endet dann mit einer praktischen Analyse von Schadsoftware durch Memory-Dumping. Das letzte Kapitel beinhaltet eine mögliche Weiterführung des Memory-Dumpings, das Thema: Import Table - Reparatur und Wiederherstellung.

Im Rahmen dieses Projektes wurde ein Memory-Dumper in Python implementiert, der insbesondere die Bibliotheken pymdmp[1], pefile[2] und psutil[3] verwendet. Die Bibliothek psutil wurde verwendet, um Informationen zu ausgewählten Prozessen abzufragen. Dadurch ist es unter anderem möglich herauszufinden, welche Prozess-ID zu welchem Prozessnamen gehört, oder mit welchen Benutzerrechten dieser ausgeführt wird. Pefile wurde verwendet, um PE-Dateien zu analysieren. Pefile erlaubt, nahezu jede Information, die gemäß der Spezifikation für PE-Dateien in den Dateien abgespeichert werden kann, auszulesen.

Um den Speicher von Prozessen zu dumpen gibt es viele verschiedene Möglichkeiten. Es existieren noch weitaus mehr Optionen als die nachfolgend aufgelisteten, es handelt sich hierbei lediglich um eine gekürzte Auswahl.

- Rekall-Framework
- pymdmp-Bibliothek

[1] pymdmp, eine Bibliothek um Speicher von Prozessen auszulesen - https://code.google.com/p/mdmp/wiki/pymdmp
[2] pefile, eine Bibliothek um Informationen aus PE-Dateien auszulesen - https://pypi.python.org/pypi/pefile
[3] psutil, eine Python-Bibliothek zum Auslesen von Prozessinformationen - https://pypi.python.org/pypi/psutil

- eine eigene Bibliothek implementieren

Für spezielle Anwendungsgebiete ist es manchmal erforderlich eine neue Bibliothek zu erstellen, in diesem Fall gibt es jedoch eine ausreichend große Auswahl von Bibliotheken. Eher denkbar wäre in diesem Zusammenhang die Verwendung des Frameworks „Rekall“. Das Framework ist sehr vielseitig und bietet neben einem Treiber, um den Speicher von Prozessen auszulesen, auch die Möglichkeit, diesen Speicher intensiv zu analysieren. Für ein größeres Projekt wäre Rekall die bessere Alternative gewesen.

Entschieden wurde sich für pymdmp, da es auf die Anforderungen des Projektes genau zugeschnitten ist. Die Bibliothek pymdmp bietet die Möglichkeit einen Memory Dump von einem einzelnen Prozess zu erstellen und operiert dabei auf Userland-Ebene. Die Installation eines Treibers ist damit nicht notwendig. Man kann auch eine Spanne von Speichersektionen festlegen, oder eine Liste von Prozessen dumpen. Es ist möglich den Heap und den Stack von Prozessen zu dumpen und dadurch an sensitive Daten zu gelangen, welche nur zur Laufzeit abrufbar sind.

Für die Verwendung in diesem Projekt wird bei jedem Dump eines Prozesses immer der gesamte verwendete Speicher gedumped (mit dump.py). Dabei schreibt das Programm jeweils die verwendeten Rechte des allozierten Speichers in den Dateinamen (Read, Write und/oder Execute) und untersucht die Datei des Dumps auf einen PE-Header. So können die DLLs des Prozesskontext, sowie die ursprüngliche .exe-Datei, die ausgeführt wurde, erkannt werden.

Sind die Dateien alle gedumped kann man mit Hilfe des FixPE.py Skripts die Adressen in der Section Table der PE-Datei mit den virtuellen Adressen überschreiben. Die Datei kann anschließend statisch analysiert werden, da das File Alignment dann wieder einer gültigen PE-Datei entspricht (zumindest weitestgehend bis zur Section Table).

Mit Hilfe des Skripts InfoPE.py kann man nun die PE-Datei vollständig auslesen und die Daten jeder Sektion übersichtlich in eine Logdatei schreiben. Dort befinden sich alle Informationen sortiert nach der Spezifikation des PE-Formats. InfoPE.py kann dabei sowohl .exe-Dateien auslesen, als auch DLLs oder andere Dateien, die dem PE-Format entsprechen. Die Funktionalität wurde erfolgreich an zwei Schadsoftware-Beispielen (citadel und urlzone) getestet, näheres dazu in Kapitel 5 (Malware Dumper).

2 Grundlegende Begriffe

2.1 Schadsoftware

Als Schadsoftware oder auch Malware bezeichnet man Computerprogramme, die „festgelegte, dem Benutzer jedoch verborgene Funktionen ausführen“ [Eck14]. In der IT-Sicherheit gibt es eine Vielzahl Kategorien von Schadsoftware, so zum Beispiel:

1. Computervirus
2. Computerwurm
3. Hintertür
4. ...

Malware wird in diesem Zusammenhang als Sammelbegriff verwendet, um jede Form von Software die dem Nutzer schaden kann zusammenzufassen. In diesem Report wird daher jede Subart von Malware behandelt, da Memory-Dumping prinzipiell auf jede Form von Schadsoftware anwendbar ist.[Eck14]

2.2 Analysetechniken

Um Malware zu Analysieren gibt es zunächst zwei verschiedene Optionen: Die statische und die dynamische Analyse. Unter einer statischen Malware-Analyse versteht man die Betrachtung der gegebenen Malware, ohne sie auszuführen. Durch das einfache Betrachten der Datei mit verschiedenen Hilfsmitteln wird die Umgebung, in der die statische Analyse durchgeführt wird, nicht gefährdet. Man kann eine statische Analyse auch von einem anderen Betriebssystem aus ausführen, damit man die Datei nicht versehentlich ausführt.

Dabei kann die Datei unter anderem auf Zeichenketten untersucht werden, man kann herausfinden ob die Datei bereits bekannt ist (z.B. durch Hochladen bei virustotal.com) oder die Datei disassemblieren und so den Kontrollfluss analysieren, noch bevor die Datei ausgeführt wird. Disassemblierung ist manchmal nur sehr beschränkt einsetzbar, zum Beispiel wenn die Datei gepackt wurde (dazu später mehr). [KM07]

Die dynamische Analyse ist das ergänzende Gegenstück zur statischen Analyse - die Datei (bzw. der Prozess der Datei) wird zur Laufzeit untersucht. Dabei kann man in einem Debugger zur Laufzeit die einzelnen Register der CPU betrachten, den Stack, den Heap, alle Veränderungen im Speicher des Prozesses können gelesen werden, sowie Netzwerkverkehr, Windows API Ausrufe, usw.

Währenddessen kann die Malware allerdings frei operieren und Schaden auf dem ausführenden Computer anrichten. Daher ist es notwendig für die Ausführung von Schadsoftware eine sichere Isolationsumgebung einzurichten, zum Beispiel durch Virtualisierungssoftware. [KM07]

Die Technik des Memory-Dumpings ist demzufolge eine Mischung aus beiden Arten. Die Schadsoftware muss ausgeführt werden, um den Speicher zu dumpen, allerdings kann der Dump anschließend statisch analysiert werden. Da es sich aber auch beim Dump nur um eine Momentaufnahme handelt, gibt er nicht so viele Informationen Preis wie eine Analyse durch Debugging. Oft ist eine komplette Debugging Analyse allerdings gar nicht notwendig um ausreichend Informationen über die Malware herauszufinden.

3 Tarntechniken

Malware wird immer professioneller entwickelt, damit werden auch die Tarntechniken der Malware immer umfangreicher. Während Schadsoftware in den 1980er und 1990er Jahren noch von Hobbyentwicklern entwickelt wurde, gibt es heute neben Entwicklerteams die von staatlichen Einrichtungen finanziert werden auch organisierte Kriminalität, die mit Schadsoftware jährlich Geld im mehrstelligen Millionenbereich umsetzt [GDA15], [fSidI14].

Programmierern stehen einige Funktionen zur Verfügung, um maliziöses Verhalten zu tarnen und zu verstecken. Bedingt durch die x86-Architektur, in welcher Daten und Code vermischt werden, ist es möglich den Kontrollfluss des Programms dynamisch zu verändern [Eck14]. Malwareautoren nutzen Techniken wie Process Hollowing, Memory Injections und andere Tricks um sich zu tarnen. Auf diese zwei Techniken wird später mit einem Beispiel eingegangen.

3.1 Packing

Eine inzwischen sehr weit verbreitete Technik ist das Packing. Unter Packing versteht man das Komprimieren oder Verschlüsseln einer ausführbaren Datei, deren Funktionalität dabei nicht verändert wird [Wik15]. Ursprünglich wird Packing benutzt, um PE-Dateien zu komprimieren. Man kann dadurch aber auch Software obfuskieren und so das Reverse Engineering von zum Beispiel kommerzieller Software erschweren.

Malware wird inzwischen sehr häufig gepackt verteilt [MCJ07]. Dadurch lässt sich die Datei nicht mehr statisch analysieren. Eine gepackte Datei enthält einen Header mit der Entschlüsselungs/Dekomprimierungs-Routine und einen verschlüsselten/komprimierten Body. Dieser Body wird nun in den Speicher geladen, dann entschlüsselt/dekomprimiert, und anschließend ausgeführt. Dabei gibt es jedoch einige Punkte zu beachten, auf die im Kapitel „Malware Dumping“ näher eingegangen wird.

Da die Datei im Speicher wieder entpackt werden muss um ausgeführt zu werden, kann man die entpackte Datei immer aus dem Arbeitsspeicher wiederherstellen. Dieses Vorgehen ist bei unbekannten Packern notwendig. Bei manchen Packern wie zum Beispiel UPX [1] kann die Datei auch statisch dekomprimiert werden, indem sie mit UPX selbst wieder entpackt wird, da UPX sowohl Packing- als auch Unpacking-Funktionalität bietet (ohne gepackte Software auszuführen und aus dem Speicher wiederherzustellen). Diesen Vorgang nennt man Malware Unpacking. Während Malware Unpacking grundsätzlich automatisierbar ist [MCJ07], lassen sich manche Tarntechniken nicht vollständig automatisiert im Speicher erkennen bzw. entfernen.

[1] UPX, ein Open Source Packer (Lizenzmodell: GPL), zu finden unter: http://upx.sourceforge.net/

3.2 Code Injection

Im Allgemeinen versteht man unter einer Code Injection die Ausführung von unerlaubtem Code in einem unerlaubten Prozesskontext. Durch die Ausführung von Code an ungewöhnlichen Stellen kann sich ein Prozess tarnen. Möglich ist diese Tarnung durch verschiedene Windows API-Funktionen, generelle Schwächen der x86-Architektur oder durch das Ausnutzen bestimmter Designentscheidungen, die in das Betriebssystem eingeflossen sind.

Generelle Konzepte unter Windows sind in diesem Zusammenhang zum Beispiel Process Injections, DLL Injections oder Process Hollowing. Viele dieser Attacken folgen unmittelbar aus der Mischung von Daten und Code in der Rechnerarchitektur, wodurch Prozesse Speicher allozieren, diesen mit ausführbarem Code füllen, diese Sektionen im Speicher als ausführbar markieren und schließlich diesen Code ausführen können. Diese Technik kommt sowohl beim Packing, als auch beim Process Hollowing zum Einsatz. Während beim Packing eine komplette Datei in den Speicher gemapped und ausgeführt wird, werden beim Process Hollowing Sektionen einer ausführbaren Datei in den Speicher geladen und ausgeführt. Eine nähere Betrachtung dieser Attacken folgt im Kapitel Dumping.

3.2.1 Process Hollowing

Unter Process Hollowing versteht man das Erstellen eines Prozesses, der „ausgehöhlt“ wird und als legitim aussehender Prozess verwendet wird. Danach wird dieser Prozess mit Daten und Code einer anderen Executable „gefüllt“ und gestartet. Die Idee die sich hinter dieser Technik verbirgt ist eine Art Emulation des Windows Loaders, der die Ausführung einer .exe-Datei kontrolliert. Wie eine .exe-Datei ausgeführt wird, ist im Kapitel PE-Format näher beschrieben. Diese Form der Code Injection ist erkennbar, indem man zum Beispiel das Speicherabbild der ausführbaren Datei mit der Datei als die sich der Prozess tarnt auf dem Dateisystem vergleicht. Genauer betrachtet wird Process Hollowing in einem Fallbeispiel im Kapitel „Malware Dumper“. [LAHR10]

4 Das PE-Format

Ein zum Beispiel in C oder C++ programmiertes und anschließend kompiliertes Programm liegt unter Windows in der Regel im Portable Executable (PE)-Format vor. Bevor die Datei ausgeführt werden kann, muss sie jedoch in den Speicher geladen werden. Um eine aus dem Speicher extrahierte Datei wieder zusammenzusetzen ist es daher erforderlich, viele Details über dieses Format zu kennen. Als Grundlage wird in den folgenden Abschnitten (wenn nicht anders beschrieben) die simple.exe Datei aus dem Projekt „PE 101“ als Beispiel verwendet[Alb15].

Besonders hervorzuheben ist die Adressierung in PE-Files: Fast jede Adresse ist ein Offset zur Base Image Adresse, so muss zum Beispiel die Adresse des Entry Point im Speicher erst berechnet werden, indem die Adresse mit der Base Image Address summiert wird. Während das PE-Format von vielen Dateitypen unter Windows verwendet wird, beschränkt sich dieser Überblick auf .exe-Dateien. Generell funktioniert die Ausführung eines Programmes im PE-Format unter Windows nach folgendem Schema:

1. Der DOS-Header wird gelesen
2. Der PE-Header wird gelesen
3. Der Optional PE-Header wird gelesen
4. Die Section Table wird gelesen
5. Die PE-Datei wird in den Arbeitsspeicher extrahiert
6. Die Import Table wird gelesen
7. Es wird zum EntryPoint gesprungen und der Code wird ab dort ausgeführt

Zunächst wird ein DOS-Header gelesen, der jedoch nur implementiert ist um die Ausführung zu beenden falls die .exe-Datei unter DOS ausgeführt wird. Der PE-Header enthält ein Feld namens SizeOfOptionalHeader, dadurch kann intern erkannt werden, wie viele weitere Zeichen gelesen werden müssen und somit auch, ob dieser Header existiert (Executables benötigen den Header um ausgeführt werden zu können).

Anschließend kann mit dem Optional PE-Header fortgefahren werden, falls er existiert (bei einer .exe-Datei muss der Header verwendet werden, damit die Datei ausführbar ist). Darin befinden sich unter anderem Informationen zur Prozessorarchitektur, zur Base Image Address (der Adresse, an der die Datei in den Speicher gemapped wird), zum EntryPoint (die Adresse, an der später die Ausführung des Codes beginnt) sowie das Feld NumberOfSections (die Anzahl der Einträge in der Section Table).[Cor13]

4.1 File Mapping

Als nächster Schritt im Lade-Prozess wird die Section Table gelesen. Die Adresse der Section Table ergibt sich aus der Summe der Startadresse des Optional Header und der SizeOfOptional-Header zusammen. Zum Beispiel bei simple.exe[Alb15]:

- Die Größe OptionalHeaders beträgt 0xE0
- Die Startadresse des OptionalHeaders liegt bei 0x58
- Die Summe ergibt 0x138, dort beginnt die Section Table von simple.exe

Mit der Kenntnis der Position der Section Table und der Anzahl der Sections können nun die einzelnen Einträge analysiert werden. Nun ist es möglich die Datei in den Speicher zu mappen bzw. zu laden. Dafür muss nur die BaseImage Adresse, das SizeOfHeaders Feld und die Section Table bekannt sein.

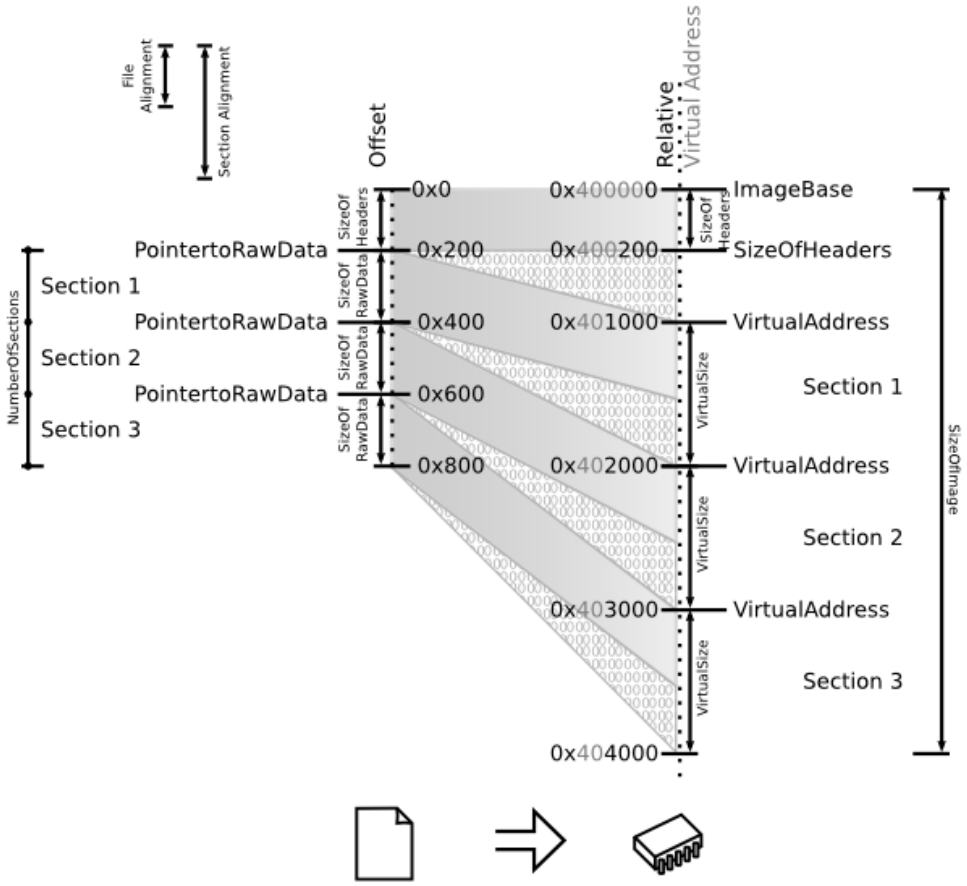


Abbildung 1: *PE-File wird in den Speicher gemapped[Alb15]*

Beim Ladevorgang wird nun die Datei an die ImageBase Adresse (in diesem Fall 0x00400000) gemapped, jede weitere Adresse wird relativ zur Image Base Adresse um ihren eigenen Betrag in den Speicher geladen. So zum Beispiel der Header: Der Header selbst ist 0x0200 Byte groß, daher wird er an die Stelle 0x00400000 geladen, ist aber an der Stelle 0x00400200 zu Ende. Bei Section Table funktioniert es analog. Die jeweilige Section wird an die Stelle der ImageBase Adresse + die VirtualOffset Adresse geschrieben, das heißt zum Beispiel bei der .text Section: 0x00400000 (BaseImage) addiert mit 0x1000 (VirtualOffset der .text Section) ergibt 0x00401000,

dort beginnt die Section .text, während sie bei 0x00402000 endet (VirtualSize beträgt 0x1000). Die .rdata Section beginnt jetzt bei 0x00402000, endet 0x1000 Byte später, da auch hier die VirtualSize 0x1000 Byte beträgt. Die nachfolgenden Schritte sind für das Projekt nur bedingt relevant, das Programm kann weder die Import Table rekonstruieren noch den Execution Point wiederherstellen. Eine mögliche Vorgehensweise wird im Kapitel „Weiterführendes“ vorgestellt. Die Informationen dienen dazu, dem Leser einen Überblick zu verschaffen.

4.2 Import Table

Bevor der Code der Datei nun ausgeführt werden kann müssen die Imports in den Prozesskontext geladen werden. Dafür wird nun die „DataDirectories“-Sektion im PE-File gelesen. Diese Sektion befindet sich in einer PE-Datei vor der Section Table und nach dem OptionalHeader. Wie viele Imports benötigt werden, kann man am Eintrag NumberOfRvaAndSizes im OptionalHeader erkennen. In der Beispieldatei sample.exe befindet sich diese Sektion an der Stelle 0xB8. Dort sind alle Einträge Null-Werte, außer dem Eintrag IMAGE_DIRECTORY_ENTRY_POINT, der an der Stelle 0x2000 liegt (durch das Mapping an der Adresse 0x00402000 im Speicher).[Alb15]

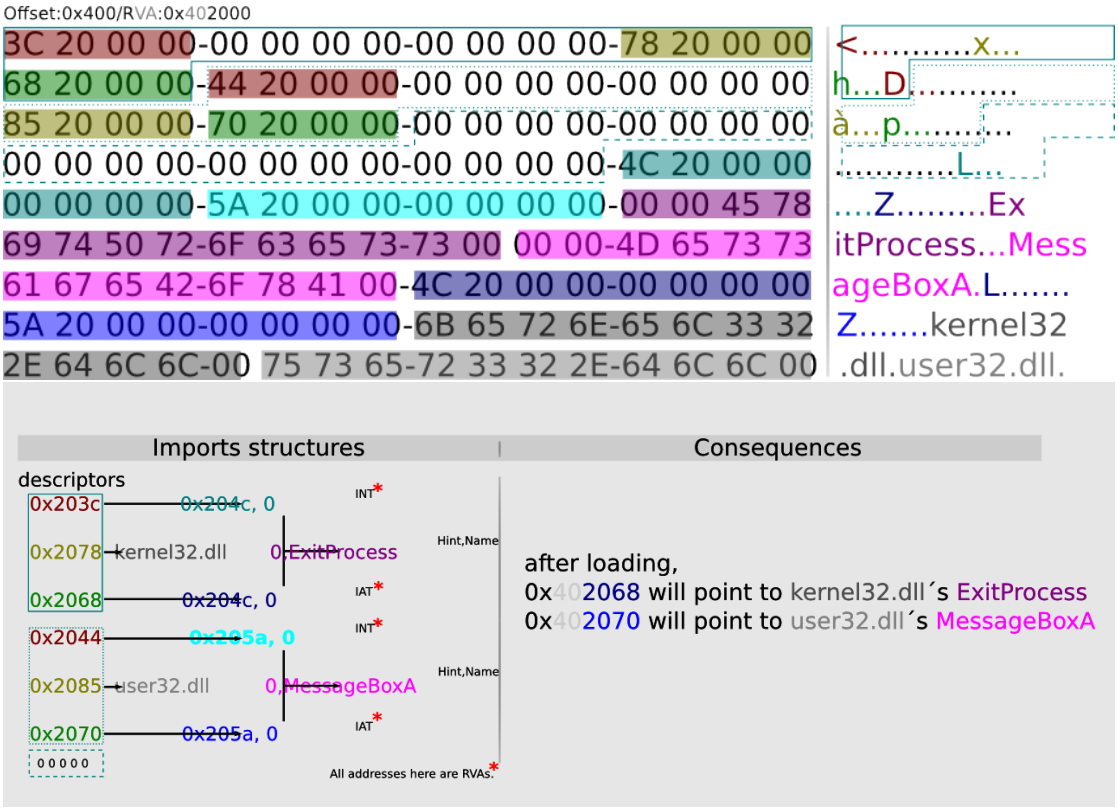


Abbildung 2: *Vollständige Import Table von simple.exe[Alb15]*

An der Adresse 0x00402000 liegen nun die Imports. Dort liegen jeweils die Adressen für den Eintrag in der Import Name Table (INT), eine Referenz auf den Namen der jeweiligen DLL und der Verweis auf den Eintrag der Import Address Table (IAT).

4.3 Execution

Um die Datei nun auszuführen, wird an die Adresse des Entry Point im Speicher gesprungen. Der Entry Point steht im IMAGE_OPTIONAL_HEADER, in unserem Beispiel 0x1000. Die Adresse im Speicher ist daher 0x00401000 (Image Base Address addiert). Dort befindet sich jetzt assemblierter Code. Diese Bytes können jetzt in mnemonische Assembler-Symbole disassembiliert werden um sie besser zu verstehen.[Alb15]

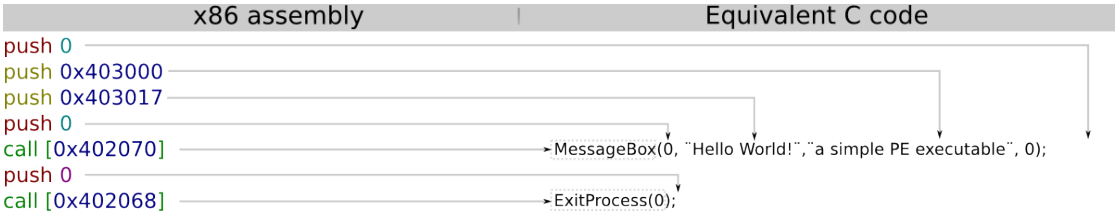


Abbildung 3: *Ausführbarer Code von simple.exe[Alb15]*

Jeder call-Aufruf erfolgt in Assembler über eine feste Adresse. Diese Adresse wird vom Linker beim Kompilieren der Datei (bzw. Beim linken) ausgerechnet. Diese Adresse wird in der Import Address Table (IAT) nachgeschlagen, sobald ein Aufruf durchgeführt wird. Der dynamische Linker von Windows löst die Abhängigkeiten zur Laufzeit auf. Die Adressen die in diesem Beispiel auf den Stack gelegt wurden (per push Operation) sind Referenzen auf die Strings in der .data Section.[TR10]

5 Malware Dumper

In diesem Kapitel wird der Malware Dumper aus dem Programmierprojekt vorgestellt. Um die folgenden Szenarien nachzustellen, ist es notwendig eine vergleichbare Laborumgebung wie diese zu erstellen:

- Eine Virtuelle Maschine mit Windows XP[1]
- Eine Python 2.7 Umgebung
- Eine Kopie des Malware Dumper inklusive aller Abhängigkeiten und Bibliotheken

Die genaue Benutzung der einzelnen Kommandos des Malware Dumpers lässt sich in der Readme-Datei nachlesen.

5.1 Dumping

Das eigentliche Konzept des Memory-Dumping mit dem Malware Dumper ist universell einsetzbar. Man kann Informationen über einen kompletten Prozess herausfinden. So wird ein Beispieldump von einer ausführbaren Datei Auskunft über jede geladene DLL im Prozesskontext geben, sowie über dynamisch allozierten Speicher. Der Malware Dumper ist damit in der Lage, im Rahmen einer Blackbox Analyse alle Informationen über einen bestimmten Prozess zu ermitteln. Die Benutzung des Malware Dumpers erfolgt dabei in der Regel nach folgendem Schema:

1. dump.py ausführen - Prozess(e) auswählen und dumpen
2. FixPE.py ausführen - die betreffende PE-Datei anpassen
3. InfoPE.py ausführen - möglichst viele Informationen aus der PE-Datei auslesen

Im erste Schritt verwendet das Skript dump.py die Bibliothek pymdmp, die bereits in der Einleitung vorgestellt wurde. Damit wird der gesamte Speicher eines Prozesses gespeichert. Dabei wird die Datei nach dem Namen des Prozesses, der ProzessID und der Startadresse im Speicher benannt. Die PE-Dateien, also in der Regel eine Executable und die zugehörigen DLLs, werden im Dateinamen zusätzlich mit „PE_“ benannt. Danach folgt im Dateinamen eine Auflistung der jeweiligen Speicherprivilegien. „RO“ steht dabei für „ReadOnly“, „RX“ für „Read, Execute“, „RWX“ für „Read, Write and Execute“ und so weiter.

[1] verwendet wurde VirtualBox, siehe https://www.virtualbox.org/

5.2 Reparatur von ausführbaren Dateien

Damit man die Datei besser analysieren kann, ist es hilfreich die Struktur der PE-Datei zu überarbeiten, um sie wieder in ein standardkonformes Format zu bringen. Hat man herausgefunden, welche Datei die ursprüngliche Executable war, kann man versuchen, die Datei mit dem FixPE.py Skript zu reparieren. Das Skript versucht, die Section Table der gespeichert PE-Datei wiederherzustellen, in dem es den PointerToRawData mit der VirtualAddress überschreibt, sowie die SizeOfRawData mit der VirtualSize überschreibt. Dadurch kann man die Datei analysieren wie eine gewöhnliche Datei, nur sieht die Datei jetzt auf dem Dateisystem identisch mit der extrahierten Datei im Speicher aus. Die Datei sieht nicht wie die ursprüngliche Datei aus, aber das ist in diesem Zusammenhang auch nicht notwendig.

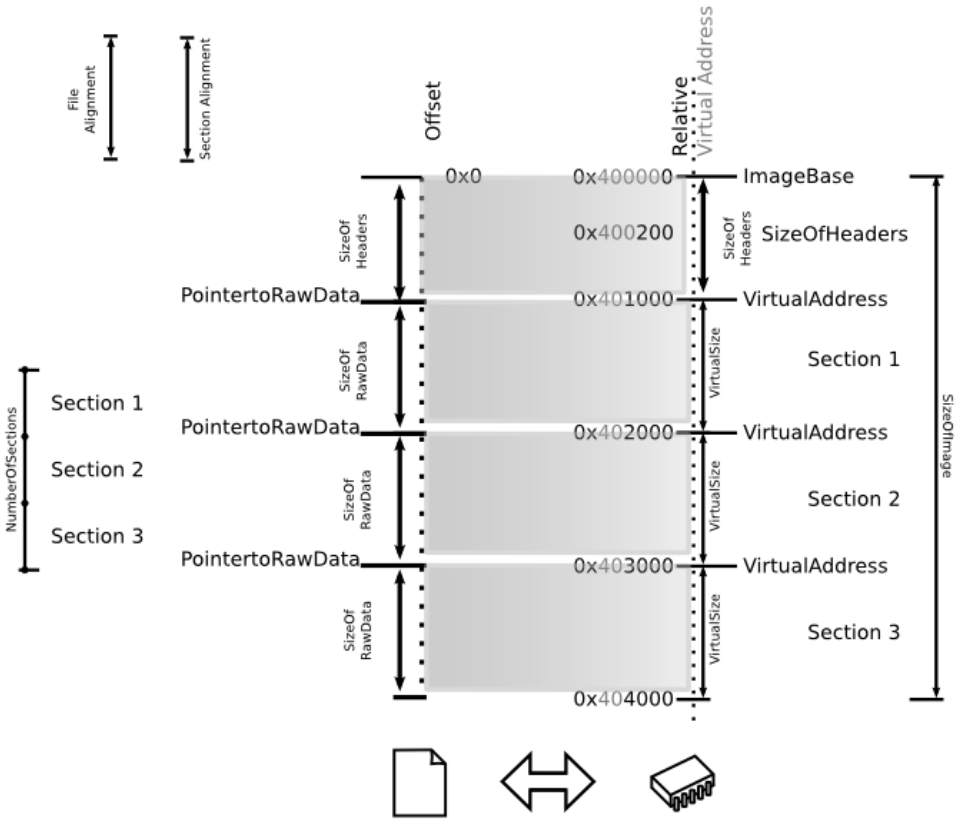


Abbildung 4: *Struktur der reparierten Datei, das Mapping ist manipuliert*

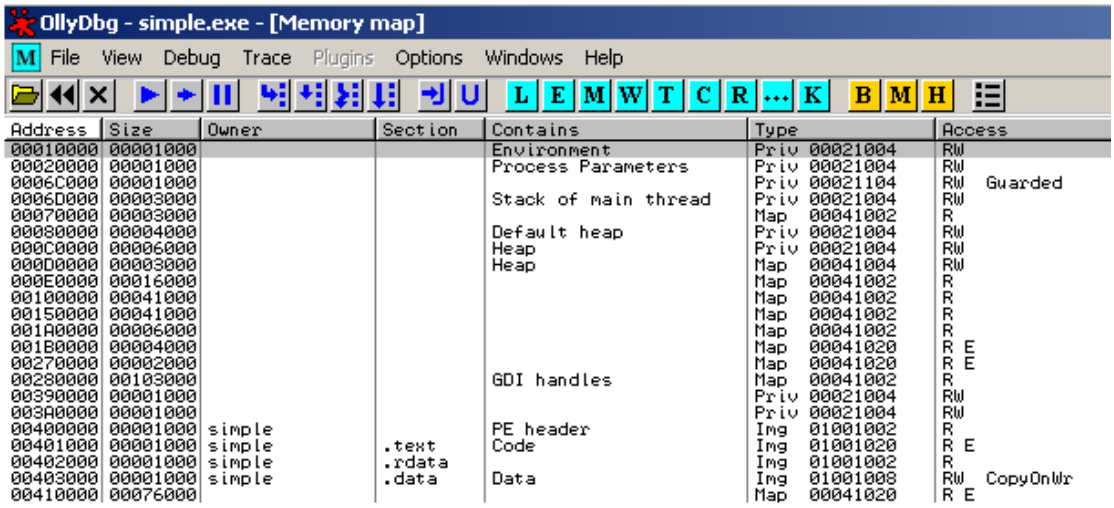
Nachdem die Section Table repariert wurde, kann man versuchen die Import Table zu fixen. Während die Import Table in der Regel (manche Schadsoftware importiert die benötigten Abhängigkeiten über Umwege) in der ursprünglichen PE-Datei zu finden ist, wird sie beim Ausführen der Datei nicht 1:1 in den Speicher geladen. Wie im Kapitel der PE-Section angedeutet, wird nur die jeweilige DLL in den Kontext des Prozesses geladen und die Funktionen werden über die Adressen in der Import Address Table aufgelöst. Um eine neue Import Table zu erstellen fehlt derzeit ein Skript im Malware Dumper. Näheres dazu und einige Konzepte die man betrachten kann um die Import Table wiederherzustellen findet man im Kapitel Weiterführendes.

Um dennoch etwas über die PE-Datei zu erfahren, kann man das InfoPE.py Skript verwenden. Bis zur Section Table sind die Einträge der Logdatei zuverlässig, alle nachfolgenden Einträge geben bei einer intakten PE-Datei Informationen über die verwendeten DLLs und den daraus ausgewählten Funktionen Auskunft. Das ist bei einer nicht-reparierten Datei aber nicht (zuverlässig) möglich. Dadurch kann das InfoPE.py Skript oft nur Hinweise auf die verwendeten DLLs geben, was aber bei einer Analyse von Malware immer noch hilfreich sein kann.

Darüber hinaus speichert der Dumper den gesamten Prozesskontext, daher auch die verwendeten DLL Dateien. Untersucht man diese DLLs näher oder vergleicht man sie mit anderen gedumpten DLL-Dateien, die man kennt, dann kann man dadurch ermitteln, welche DLLs der Prozess verwendet. Dann kann aus diesem Wissen eine neue Import Table erstellt werden, da die Import Name Table wiederhergestellt werden kann, neben der benötigten Import Address Table.

5.3 Stack und Heap

Der Memory Dumper speichert auch den Stack und den Heap (bzw. mehrere Heaps und Stacks, zum Beispiel durch Multi-Threading und per Design von Windows). Im Speicher des Prozesses befinden sich Stack und Heap bei einem Prozess vor der eigentlichen Executable.



Address	Size	Owner	Section	Contains	Type	Access
00010000	00001000			Environment	Priv 00021004	RW
00020000	00001000			Process Parameters	Priv 00021004	RW
0006C000	00001000				Priv 00021104	RW Guarded
0006D000	00003000			Stack of main thread	Priv 00021004	RW
00070000	00003000				Map 00041002	R
00080000	00004000			Default heap	Priv 00021004	RW
000C0000	00006000			Heap	Priv 00021004	RW
000D0000	00003000			Heap	Map 00041004	RW
000E0000	00016000				Map 00041002	R
00100000	00041000				Map 00041002	R
00150000	00041000				Map 00041002	R
001A0000	00006000				Map 00041002	R
001B0000	00004000				Map 00041020	R E
00270000	00002000				Map 00041020	R E
00280000	00103000			GDI handles	Map 00041002	R
00390000	00001000				Priv 00021004	RW
003A0000	00001000				Priv 00021004	RW
00400000	00001000	simple		PE header	Img 01001002	R
00401000	00001000	simple	.text	Code	Img 01001020	R E
00402000	00001000	simple	.rdata		Img 01001002	R
00403000	00001000	simple	.data	Data	Img 01001008	RW CopyOnWr
00410000	00076000				Map 00041020	R E

Abbildung 5: *Layout des virtuellen Speichers von simple.exe in OllyDBG*

Während die statische Analyse einer PE-Datei viele Informationen über Code geben kann, findet man in Heap und Stack dynamische Daten, die nur während der Laufzeit erreichbar sind. Daher kann eine Analyse dieser Dumps zusätzliche Informationen ergeben, so können beispielsweise verschlüsselte Strings einer Malware zur Laufzeit entschlüsselt auf einem Stackframe im Dump zu finden sein.

5.4 unbekannte Memory-Sektionen

Häufig erkennt man beim Dumping von Malware, dass der Prozess bestimmte Speichersektionen alloziert hat, die ein gewöhnlicher Prozess sonst nicht alloziert. Unüblich ist es beispielsweise, Speicher zu allozieren, ihn mit ausführbaren Instruktionen zu füllen und diesen anschließend

auszuführen. Anhand solcher, als „RWX“ gekennzeichneten Memory-Sektionen, erkennt man sehr häufig Tarntechniken. Sowohl Packer nutzen diese Art der Speicheranforderung, um eine komprimierte bzw. verschlüsselte PE-Datei in den Speicher zu mappen, als auch Prozesse die Process Hollowing zur Tarnung verwenden.

Daher ist es nicht nur wichtig den gesamten Speicher des jeweiligen Prozesses erfolgreich zu dumpen, sondern es ist auch notwendig die Memory-Sektionen zu klassifizieren. Dadurch kann man später erkennen, welche Sektionen Daten und welche Sektionen Code bei der Ausführung der Schadsoftware enthalten haben. War ein Speicherbereich beispielsweise als als „Read, Write, Executable“ markiert, dann kann man davon ausgehen dass die Schadsoftware die benannten Schritte durchgeführt hat.

5.5 Unpacking durch Dumping

Um eine gepackte Datei durch Unpacking wiederherzustellen, ist es zunächst notwendig einige generelle Informationen über die Funktionsweise eines Packers herauszufinden. Viele Packer funktionieren nach einem vergleichbaren Schema. Die zugrundeliegende Technik ist einfach: Die Datei, die gepackt wurde, ist eine gewöhnliche PE-Datei gewesen. Diese Datei hatte einen EntryPoint (im Folgenden als OriginalEntryPoint bezeichnet, da die gepackte PE-Datei natürlich auch einen EntryPoint besitzt), ab dem der Code der ursprünglichen Datei ausgeführt wird. Zusätzlich wird jede Section aus der Section Table komprimiert. Der Packer kopiert nun die gepackte Datei in den Speicher indem er jede Section in den Speicher entpackt und an den OriginalEntryPoint springt.

Von dort aus kann die Datei wie gewohnt ausgeführt werden. Damit die Ausführung gelingt, müssen allerdings die DLLs, die in der Import Table der ursprünglichen PE-Datei standen, auch in den Speicher geladen werden. Der Packer muss die Imports der PE-Datei nach dem Mapping wiederherstellen.

Zusammengefasst funktioniert die Ausführung einer gepackten PE-Datei nach diesem Ablauf:

1. Die Ausführung beginnt bei einer Dekomprimierungsroutine
2. Der Zustand aller Prozessorregister wird auf dem Stack gespeichert
3. Alle gepackten Sektionen werden entpackt in den Speicher gemapped
4. Die Import Table der ursprünglichen PE-Datei wird wiederhergestellt.
5. Die Prozessorregister werden in den gespeichert Zustand zurückgesetzt
6. Eine Sprunginstruktion an den OriginalEntryPoint erfolgt und die Datei wird ausgeführt

Dabei kann man leicht erkennen, dass die Datei vollständig ungepackt nach dem Sprung zum OriginalEntryPoint im Speicher liegt damit die ursprüngliche Ausführung funktionieren kann. Nun gilt es aus diesem Speicher die ursprüngliche Datei zu dumpen. Für das Reverse Engineering gibt es nun mehrere Vorgehensweisen. Man kann versuchen die PE-Datei aus dem Speicher wiederherzustellen, um sie ausführbar zu machen und so mögliches Debugging und die dynamische Analyse vereinfachen. Es ist aber auch möglich die Datei zu dumpen, zu reparieren und anschließend statisch zu Analysieren. Da im Rahmen dieser Arbeit der Fokus auf der statischen Analyse durch Memory-Dumping liegt, wird die Datei nur gedumpt, aber nicht ausführbar gemacht. Falls man eine Datei nach dem Dumping ausführen will, sollte man auf Tools wie ImpRec oder Scylla zurückgreifen.

Eine mögliche weitere Problematik, eine Datei mittels Dumping wiederherzustellen, besteht nun darin die einzelnen Sektionen aus dem Dump der Executable zu sortieren und zusammenzufügen. Manche Packer kopieren auch den ursprünglichen PE-Header an den dynamisch

allozierten Speicherbereich und bilden so die ursprüngliche Datei nahezu identisch bis zum Ende der Sektionen in den Speicher. Ist das jedoch nicht der Fall, muss zusätzlich ein kompletter PE-Header künstlich erzeugt und vorangestellt werden. Man kommt schnell zu dem Entschluss, dass Malware Unpacking grundsätzlich nach einem wiederkehrenden Schema bearbeitet werden kann, diese Vorgehensweise aber sehr begrenzt bleibt. Malware Autoren haben zu viele Möglichkeiten, Code zu verstecken und auszuführen, dass man mit einem Schema jedes gepackte Malware Sample entpacken kann. [Gef12]

Der Malware Dumper ist demzufolge bisher nur bedingt geeignet, um Unpacking zu betreiben. Für diesen Zweck gibt es allerdings bereits Tools wie ImpRec und Scylla. Der Malware Dumper liefert dafür andere Informationen, nämlich Dumps über den gesamten Speicher des Prozesses, wodurch sich mehr als nur die Executable analysieren lässt.

5.6 Analysebeispiel von citadel und urlzone

5.6.1 citadel

Als im Mai 2011 der Quellcode des bekannten Banking-Trojaners Zeus auftauchte, entstanden aufgrund der nun kurzen Entwicklungszeit viele Variationen des ursprünglichen Trojaners in kurzer Zeit. Im Januar 2012 wurden daraufhin erste Exemplare von citadel entdeckt. [Mil12]

Es erfolgt ein kurzer Ausschnitt aus einem möglichen Analyse-Szenario. Zunächst wird die Datei (sehr oberflächlich) statisch analysiert, das heißt es wird das InfoPE.py-Skript verwendet, um einen ersten Eindruck der Schadsoftware zu gewinnen. Da die Datei nicht gepackt ist, ist es möglich erste Informationen zu gewinnen. In der folgenden Tabelle ein Auszug einiger verwendeter Funktionen:

KERNEL32.dll	GetDateFormatA	KERNEL32.dll	HeapCreate
KERNEL32.dll	SetFileAttributesA	KERNEL32.dll	DeleteCriticalSection
KERNEL32.dll	GetEnvironmentVariableA	KERNEL32.dll	InitializeCriticalSection
KERNEL32.dll	GetLastError	KERNEL32.dll	IsValidCodePage
KERNEL32.dll	GetStartupInfoA	KERNEL32.dll	LocalFree
KERNEL32.dll	GetProcAddress	KERNEL32.dll	GetTickCount
KERNEL32.dll	HeapFree	KERNEL32.dll	LocalAlloc
KERNEL32.dll	FreeLibrary	KERNEL32.dll	LocalSize
KERNEL32.dll	LoadLibraryA	KERNEL32.dll	HeapDestroy
KERNEL32.dll	HeapAlloc	KERNEL32.dll	SetEndOfFile
KERNEL32.dll	GetThreadLocale		

Abbildung 6: *Alle Funktionen aus Kernel32.dll von citadel*

Es ist erkennbar, dass viele der aufgelisteten Funktionen direkt mit dem Betriebssystem kommunizieren, damit die eigentliche Schadsoftware unerkannt bleibt. So zum Beispiel HeapCreate und HeapDestroy, um einen neuen Heap zu erzeugen und unmittelbar zu löschen. Die komplette Funktionalität der Schadsoftware wird offenbar erst zur Laufzeit sichtbar, sobald durch „LoadLibraryA“ Bibliotheken nachgeladen werden.

Wird die Malware nun ausgeführt, ist schnell erkennbar, dass es keinen Prozess mit dem Namen der gestarteten .exe-Datei gibt. Verwendet man nun den Malware Dumper, um alle laufenden Prozesse zu dumpen, stellt man im erzeugten Ordner fest, dass es keinen Prozess gibt, der verdächtig aussieht. Wechselt man nun in den Ordner für den Explorer-Prozess, dann sieht man zwei Dateien, die die Zeichenkette „RWX“ enthalten, eine davon eine PE-Datei. Diese

beiden Dateien eignen sich besonders gut als Einstiegspunkt, da der Windows Explorer in der Regel keine Speicherbereiche mit diesen Zugriffsrechten alloziert. Es ist davon auszugehen, dass citadel durch eine Memory Injection in diesen Prozessbereich vom Windows Explorer schreibt.

Zunächst kann versucht werden, die PE-Datei zu reparieren, indem das Skript FixPE.py verwendet wird. Nachdem die Datei repariert wurde, kann InfoPe.py verwendet werden. Dadurch lässt sich erkennen, dass die Funktionalität der Schadsoftware nicht nur auf die in der ursprünglich ausgeführten PE-Datei beschränkt ist.

Es werden unter anderem DLLs wie ws2_32.dll, wininet.dll und netapi32.dll importiert. Diese Bibliotheken werden verwendet, um auf Netzwerkssockets zuzugreifen und dadurch über das Internet zu kommunizieren. Diese Datei kann nun mit einem Programm wie IDA Pro analysiert werden, um weitere Informationen über die Schadsoftware zu sammeln.

Übrig bleibt die andere Datei. Da bisher nur bekannt ist, dass es sich bei diesem Speichersegment nicht um eine PE-Datei handelt, kann der Malware Dumper nicht verwendet werden um Informationen zu finden. Es ist allerdings möglich, einen Überblick durch das Öffnen der Datei in einem Hexeditor zu gewinnen:

Abbildung 7: *citadel: explorer.exe-<Prozess-ID>-00EA0000-[RW]_[RWX]_[RW].dump, von 0x69D2 bis 0x75C6*

Man erkennt leicht, dass die Schadsoftware über HTTP-Funktionalität verfügt. Scheinbar reagiert die Schadsoftware auch auf Antiviren-Software, da Zeichenketten von bekannten Her-

stellern im Dump zu finden sind, so zum Beispiel „Kaspersky Security“, „avast! Antivirus“, „McAfee Security Center“ und „Symantec Security“. In einer vollständigen Analyse könnte man nun versuchen, den Code in einem Disassembler zu lesen und so weitere Details der Funktionsweise herauszufinden.

5.6.2 urlzone

Im September 2009 wurden erste Varianten von urlzone entdeckt. Der Banking-Trojaner urlzone versucht Bankdaten seiner Opfer durch eine „Man-In-The-Browser“-Attacke zu stehlen.[Tan12]

Wird nun urlzone ausgeführt und der Malware Dumper verwendet, entdeckt man wie bereits bei citadel im Prozessraum des Windows Explorer eine Memory Injection. Es sind drei Dateien zu finden, in denen sich nun Speicherbereiche mit den Privilegien „RWX“ befinden. Eine davon ist eine PE-Datei. Diese Datei lässt sich zwar Reparieren, sobald jedoch InfoPE.py verwendet wird, lassen sich in der Ausgabedatei keine Hinweise auf die Funktionsweise von urlzone finden.

Wird die PE-Datei aber mit einem Hexeditor geöffnet, sind unverschlüsselte Zeichenketten erkennbar, welche Hinweise auf die Funktionsweise von urlzone offenbaren:

Abbildung 8: *urlzone: explorer.exe-<Prozess-ID>-01410000-PE_[RWX].dump, links: von 0x00001DE2 bis 0x000023DC, rechts oben: von 0x000023DC bis 0x0000267F, rechts unten: von 0x0001C1D3 bis 0x0001C4FD*

Der linke Screenshot aus Abbildung 8 enthält zum Beispiel ähnliche Zeichenketten wie citadel, zum Beispiel „GET / HTTP/1.1“ oder „User-Agent: Mozilla/5.0“, die ebenfalls auf HTTP-Funktionalität der Schadsoftware hinweisen. Im Gegensatz zu citadel enthält urlzone aber zum Beispiel String wie „chrome.dll“, „opera.dll“, „iexplore.exe“, „firefox.exe“ und „opera.exe“. Diese Strings könnten ein Hinweis auf den „Man-In-The-Browser“-Angriff sein.

Der Screenshot rechts unten in Abbildung 8 weist auf möglicherweise komprimierten Code hin, den die Schadsoftware verstecken will. Der inflate Algorithmus ist ein Teil der zlib-Bibliothek[2]. Dadurch ist außerdem der Erstellungszeitpunkt erkennbar, offenbar wurde dieses Exemplar von urlzone 2010 kompiliert. Um die Schadsoftware nun detaillierter zu analysieren, könnte man die PE-Datei in einem Disassembler zu untersuchen.

[2] Eine Bibliothek um beliebige Daten zu Komprimieren, http://www.zlib.net/

6 Fazit und Ausblick

Durch Malware Dumping lassen sich viele Informationen über Schadsoftware gewinnen. Während die statische Analyse bei Malware wie zum Beispiel citadel nur sehr begrenzte Informationen über die Funktionsweise kenntlich macht, kann man in einem Speicherauszug viele Hinweise entdecken. So konnte zum Beispiel bei citadel und urlzone Netzwerkfunktionalität ermittelt werden, während in den ursprünglichen PE-Dateien nur Hinweise auf Memory Injections zu finden waren.

Der Malware Dumper weist in einigen Teilen seines Anwendungsgebietes Schwächen auf. Der Dumper ist nicht in der Lage, eine gedumpte Datei wieder ausführbar zu machen. Für das erfolgreiche Reverse Engineering von Executables ist es aber in manchen Situationen hilfreich die Datei erneut ausführen zu können, und zwar entschlüsselt. Das kann eine dynamische Analyse unter Umständen sehr viel einfacher machen. Damit das machbar ist, fehlt dem Malware Dumper ein Skript welches eine neue PE-Datei auf Basis des Dumps erzeugen kann.

Manche Packer verwerfen den ursprünglichen Datei Header, da er für die korrekte Ausführung des Programms nicht vollständig benötigt wird. Damit man die Datei aber wieder ausführen kann, muss die PE-Datei unter anderem einen intakten Header haben. Denkbar wären verschiedene Ansätze, für die meisten gedumpten Programme wäre eine existierende PE-Datei denkbar, von der der Header kopiert und upgedatet wird.

Dann müsste der Dump in die ursprünglichen Sektionen zerlegt werden, gegebenenfalls müssten neue Sektionen erstellt oder Sektionen zusammengelegt werden. Diese müssen nacheinander in eine neue Section Table eingebunden werden. Dann fehlt der PE-Datei noch eine gültige Import Table. Die nächsten Schritte in der Implementierung eines verbesserten Malware Dumpers bestehen aus einem Skript, welches die Import Table neu bilden kann.

6.1 Wiederherstellung der Import Table

Die Wiederherstellung einer Import Table ist vergleichsweise komplex. Der schwierige Teil besteht darin, aus den ermittelten Daten eine Struktur zu erstellen, die der Windows Loader versteht, damit dieser alle DLLs in den Kontext des Prozesses laden kann. Die Import Table funktioniert nach diesem Schema:

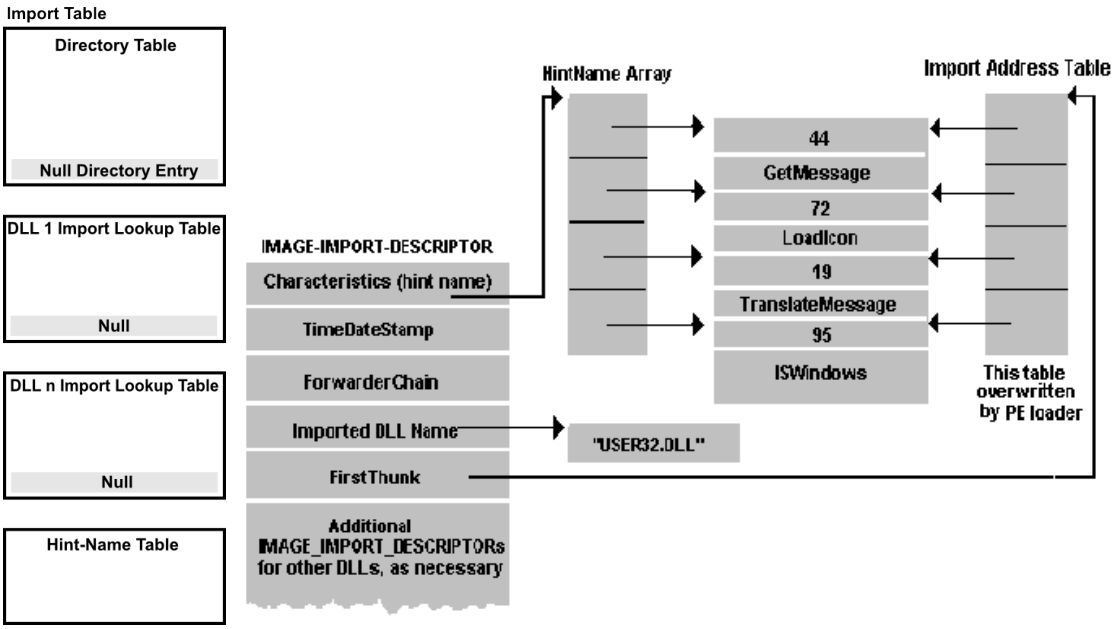


Abbildung 9: *Links: Import Table[Cor13][S. 86], Rechts: Struktur einer importierten DLL[Pie94] (jeweils ein IMAGE_IMPORT_DESCRIPTOR)*

Diese Import Table kann man wiederherstellen, indem man die Directory Table standardkonform erstellt und dort auf die Import Table verweist. Anschließend muss für jede geladene DLL eine Liste verwendeter Funktionen erstellt werden, die jeweils den Namen der verwendeten Funktionen enthalten. Diese Informationen müssen anschließend in die PE-Datei geschrieben werden.

Zu beachten ist auch, dass sich in einer PE-Datei nur NULL Pointer anstelle von Funktionszeigern in der Import Table befinden. Die Adressen werden beim Start der Datei durch den Windows Loader dynamisch ermittelt, da die genaue Adresse variieren kann. Deshalb ist es nicht möglich, die Adressen einfach in die Import Table zu schreiben. Es ist notwendig, während der Ausführung des Programms, welches gedumped werden soll, den Namen der DLL und der zugehörigen, aufgerufenen Funktion zu ermitteln. Dieses Vorgehen ist zur Laufzeit notwendig, da sich zum Beispiel durch ASLR (Address Space Layout Randomization) bereits beim nächsten Start des Programms die Adressen der importierten Funktionen ändern können.

Konkret bedeutet dass, ein Debugger muss an den laufenden Prozess angehangen werden, dann müssen alle Funktionszeiger in der Code Section gesucht werden, diese müssen in den Namen der DLL und der zugehörigen Funktion aufgelöst werden und dann kann man eine Import Table erstellen. Wenn diese Import Table dann die Spezifikation einhält, kann die Executable die Funktionen aus verschiedenen DLLs verwenden und ist ausführbar.

7 Literaturverzeichnis

[Alb15] Ange Albertini. Pe - the portable executable format on windows, 2015. [Online; Stand 28. Oktober 2015].

[Cor13] Microsoft Corporation. Microsoft portable executable and common object file format specification, 2013. [Online; Stand 26. Oktober 2015].

[Eck14] Claudia Eckert. *IT-Sicherheit: Konzepte, Verfahren, Protokolle*. Studium. de Gruyter Oldenbourg, München, 9. aufl. edition, 2014.

[fSidI14] Bundesamt für Sicherheit in der Informationstechnik. Die lage der it-sicherheit in deutschland 2014. *Bericht zur Lage der IT-Sicherheit in Deutschland*, 2014.

[GDA15] GDATA. Malware - die frühen jahre, 2015. [Online; Stand 26. Oktober 2015].

[Gef12] Jason Geffner. Unpacking dynamically allocated code, 2012.

[KM07] Kris Kendall and Chad McMillan. Practical malware analysis, 2007.

[LAHR10] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley Publishing, 2010.

[MCJ07] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441, Dec 2007.

[Mil12] Jason Milletary. Citadel trojan malware analysis, 2012.

[Pie94] Matt Pietrek. Peering inside the pe, 1994. [Online; Stand 26. Oktober 2015].

[Tan12] Neo Tan. Urlzone reloaded: new evolution, 2012.

[TR10] Reji Thomas and Bhasker Reddy. Dynamic linking in linux and windows, part one, 2010. [Online; Stand 9. November 2015].

[Wik15] Wikipedia. Executable compression — wikipedia, the free encyclopedia, 2015. [Online; Stand 28. Oktober 2015].