Information Security

# Konzeption und Proof of Concept eines Honeypots für Industriesteueranlagen

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Andreas Plank

is161523

im Rahmen des

Studienganges Information Security an der Fachhochschule St. Pölten

Betreuung

Betreuer/in: FH-Prof. Dipl.-Ing. Dr. Paul Tavolato

Mitwirkung:

St. Pölten, 9. Oktober 2018 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

*

# Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/-Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

*Ort, Datum* *Unterschrift*

# Kurzfassung

Die vorliegende Masterarbeit befasst sich mit der Konzeption eines Honeypots für Industriesteueranlagen. Zuerst werden die wichtigsten Grundlagen und State-of-the-Art-Empfehlungen von Honeypots, wie Honeypot-Typen, Honeypot-Klassifikationen, Anforderungen an Honeynets und Netzwerkpositionierung von Honeypots erläutert.

Danach erfolgt eine Darstellung der wichtigsten Protokolle von Industriesteueranlagen, mit besonderem Schwerpunkt auf das weit verbreitete Modbus-Protokoll. Es wird ein Konzept ausgearbeitet, welches sich das Ziel setzt, reale Steuerungen inklusive deren Daten so realitätsnah wie möglich zu simulieren und dabei den erforderlichen Konfigurationsaufwand so gering wie möglich zu halten. Dafür werden die State-of-the-Art-Empfehlungen von Honeypots auf die Anforderungen von Industriesteueranlagen angewandt.

Das erstellte Konzept wird anhand einer Proof-of-Concept-Implementierung für das Modbus-Protokoll abgebildet. Zur Überprüfung der Implementation werden Honeypot und eine Test-Steuerung mit verbreiteten Angriffswerkzeugen attackiert und die Ergebnisse verglichen.

# Abstract

This master's thesis deals with the conception of a honeypot for industrial control systems. First, the key foundations and state-of-the-art recommendations of honeypots, such as honeypot types, honeypot classifications, honeynet requirements, and network positioning of honeypots, are discussed.

This is followed by a presentation of the most important protocols of industrial control systems, with particular emphasis on the widely used Modbus protocol. A concept is being developed, which aims to simulate real contol systems (including data) as realitically as possible while keeping the required configuration effort as low as possible. Honeypots' state-of-the-art recommendations are applied to the requirements of industrial control systems.

The concept is modeled using a proof-of-concept implementation for the Modbus protocol. To verify the implementation, the honeypot and a ICS are testet with common attack tools. Then the results are compared.

# Inhaltsverzeichnis


**1. Einleitung** **1**

**2. Grundlagen** **3**

2.1. Honeypots . . . . . 3

2.1.1. Allgemein . . . . . 3

2.1.2. Honeynets und Honeytoken . . . . . 4

2.1.3. Klassifikation von Honeypots . . . . . 5

2.1.4. Positionierung . . . . . 6

2.1.5. Anforderungen an Honeypots . . . . . 7

2.2. Industriesteueranlagen . . . . . 7

2.2.1. Allgemein . . . . . 7

2.2.2. Protokolle . . . . . 8

**3. Konzept** **12**

3.1. Design der Architektur . . . . . 12

3.1.1. Topologie . . . . . 12

3.1.2. Angebotene Dienste . . . . . 13

3.1.3. Data Control . . . . . 14

3.1.4. Data Capture und Data Collection . . . . . 15

3.2. Anforderungen Hardware & Software . . . . . 15

3.2.1. Virtualisierung . . . . . 15

3.2.2. Betriebssystem . . . . . 16

3.2.3. Zusatzsoftware . . . . . 16

3.3. Spezielle Anforderungen für die Proof-of-Concept-Implementierung . . . . . 17

3.3.1. Information Gathering mittels nmap . . . . . 17

3.3.2. Modbus-Angriffe mittels metasploit . . . . . 18

**4. Implementierung des Proof of Concept** **21**

4.1. Hauptprogramm . . . . . 22

4.1.1. Konfig-Load . . . . . 22

4.1.2. Sanity-Check . . . . . 23

4.2. Honeypot - Echtbetriebsmodus . . . . . 25

4.2.1. Datenspeicher . . . . . 25

4.2.2. Verbindung Honeypot zu PLC . . . . . 26

4.2.3. Laufender Datenabgleich Honeypot und PLC . . . . . 27

4.2.4. Start der Server-Komponente . . . . . 29

4.2.5. ModpotServerFactory - Logging der Vorgänge . . . . . 30

4.3. Honeypot - Demomodus . . . . . 34

**5. Test des Honeypots** **35**

5.1. Vergleich mittels nmap . . . . . 35

5.2. Vergleich mittels metasploit . . . . . 36

**6. Fazit** **38**

6.1. Aufgetretene Schwierigkeiten und weiterführende Forschung . . . . . 39

**A. Sourcecode Modpot** **40**

A.1. main.py . . . . . 40

A.2. server.py . . . . . 45

A.3. protocols.py . . . . . 49

A.4. demoserver.py . . . . . 54

**Abbildungsverzeichnis** **56**

**Tabellenverzeichnis** **57**

**Listingverzeichnis** **57**

**Literatur** **61**

# 1. Einleitung

Die Europäische Union beschloss am 6. Juli 2016 die Richtlinie “über Maßnahmen zur Gewährleistung eines hohen gemeinsamen Sicherheitsniveaus von Netz- und Informationssystemen in der Union”. Besonders schützenswert sind dabei “Betreiber wesentlicher Dienste” wie zum Beispiel Energie, Verkehr oder Trinkwasserversorgung. Viele dieser “kritischen Infrastrukturen” sind zu einem hohen Maß abhängig von Industriesteueranlagen. [1]

In den letzten Jahren wurden ebendiese Industriesteueranlagen von verschiedenen speziell entwickelten Malware-Attacken bedroht. Die fünf bekanntesten Angriffe sind:

- Stuxnet
- Havex
- Industroyer
- TRISIS
- BlackEnergy

Die Folge von gezielten oder auch zufälligen Attacken sind meist Serviceausfälle, bis hin zu Zerstörung bzw. Sabotage von Anlagen. Industroyer zum Beispiel führte im Dezember 2016 zu einem großflächigen Blackout in Kiew, der weite Teile der Stadt für ca. eine Stunde von der Stromversorgung trennte. [2]

Jedoch auch bei Industriesteuerungen, die nicht in kritischen Bereichen, sondern in Produktionsunternehmen eingesetzt werden, ist es wichtig Serviceausfälle und damit Produktionsstillstände zu vermeiden. Die Herausforderung besteht oft darin, dass die Produktionsnetze nicht durch herkömmliche Methoden, wie zum Beispiel Antiviren-Software oder Firewalls, geschützt werden können, da es sich nicht um Standard Hard- und Software handelt.

Eine Möglichkeit diese Infrastrukturen vor Angriffen zu schützen und dabei mehr über die verwendeten Methoden zu lernen bilden Honeypots. Diese Arbeit beantwortet in diesem Themenbereich die folgenden zwei Forschungs-Fragen:

1. Wie sollte ein Honeypot für Industriesteuerungen konzipiert werden?
   Dabei werden Erkenntnisse und State-of-the-Art-Empfehlungen für herkömmliche Honeypots auf Industrie-Honeypots angewendet.

2. Wie ist ein solches Konzept technisch realisierbar?
   Dies wird anhand einer Proof-of-Concept-Implementierung (für das weit verbreitete Protokoll Modbus) und anschließenden Tests validiert.

Eine Implementierung und Behandlung sämtlicher Protokolle, die Industriesteueranlagen verschiedenster Hersteller verwenden, würde den Rahmen dieser Arbeit sprengen und wird daher nicht durchgeführt.

# 2. Grundlagen

## 2.1. Honeypots

### 2.1.1. Allgemein

Honeypots können im Security-Umfeld für verschiedenste Zwecke eingesetzt werden. Beispiele dafür sind Deception-Techniken, Intrusion Detection oder Sammlung von Threat Intelligence Daten. Dabei verfolgt der/die Betreiber/in eines Honeypots meist das Ziel verschiedene Informationen über Angriffe auf ein Unternehmensnetzwerk zu sammeln. Die grundlegenste Information ist dabei die Erkenntnis, dass Angriffe auf die Computersysteme stattfinden. Da es auch beim Thema Honeypot verschiedene Auffassungen darüber gibt, worum es sich nun im Detail handelt, wird in diesem Kapitel der Begriff definiert. Als übergreifende Festlegung des Begriffs Honeypot eignet sich folgendes Zitat:

> A honeypot is a security resource whose value lies in being probed, attacked, or compromised. [3, p. 40]

Da nicht näher festgelegt ist, was eine "security resouce" genau ist, muss ein Honeypot nicht zwangsweise ein physisches System (wie zum Beispiel ein Server) sein. Auch virtuelle Ressourcen, wie Services ohne produktiven Nutzen, können einen Honeypot darstellen. [3, pp. 40]
Zur Klarstellung der obigen Definition, eignet sich die folgende Konkretisierung:

> A honeypot is a resource which pretends to be a real target. A honeypot is expected to be attacked or compromised. The main goals are the distraction of an attacker and the gain of information about an attack and the attacker. [4, p. 3]

Ein Honeypot ist daher eine Security-Ressource, die sich wie ein echtes System verhält und dessen Zweck darin besteht Informationen über Cyberangriffe (und den/die eigentliche/n Angreifer/in) zu sammeln.

### 2.1.2. Honeynets und Honeytoken

Die Begriffe Honeynet und Honeytoken sind spezielle Ergänzungen zu der allgemeinen Definition von Honeypots.

#### Honeynets

Honeynets bieten eine Honeypot-Lösung mit höchster Interaktionsmöglichkeit (High-Interaction-Honeypots). Es handelt sich dabei im Grund um ein Netz voller Produktiv-Systeme, deren Ziel einzig und allein daran liegt angegriffen zu werden. Da für diesen Zweck dedizierte Systeme verwendet werden, ist keine Einrichtung von `chroot` oder anderen "Jail"-Mechanismen verzichtet werden. [3]
Das folgende Zitat eignet sich als zusammenfassende Beschreibung von Honeynets:

> Honeynets represent the extreme of research honeypots. They are high interaction honeypots, which allow learning a great deal; however they also have the highest level of risk. Their primary value lies in research, gaining information on threats that exist in the Internet community today. A Honeynet is a network of production systems. Unlike many of the honeypots, nothing is emulated. Little or no modifications are made to the honeypots. This gives the attackers a full range of systems, applications, and functionality to attack. From this it can be learnt a great deal, not only their tools and tactics, but their methods of communication, group organization, and motives. [5, p. 9]

Welche Anforderungen Honeynets erfüllen müssen, um sicher betrieben werden zu können, wird in Kapitel 2.1.5 beschrieben.

#### Honeytoken

Bei einem Honeytoken sind vereinfacht Dinge oder Informationen, die dazu dienen auf unauthorisierte Weise verwendet zu werden. Das können zum Beispiel Dateien, Zahlungsinformationen oder Passwörter sein, die keinen produktiven Nutzen haben. Durch die Verwendung dieser Informationen kann darauf geschlossen werden, dass eine unauthorisierte Person Zugriff auf diese Daten hatte und ein Angriff auf das Netzwerk erfolgt ist. [5]

### 2.1.3. Klassifikation von Honeypots

Zur Klassifikation von Honeypots eignen sich zwei unterschiedliche Betrachtungsebenen:

1. Verwendungszweck
2. Level of Interaction

#### Verwendungszweck

Bei der Klassifizierung auf Basis des Verwendungszwecks, wird meist zwischen Produktionshoneypot und Forschungshoneypot unterschieden.

**Produktionshoneypots** sind primär dazu gedacht ein Computernetzwerk vor einem/einer Angreifer/in zu schützen. Dies wird dadurch erreicht, dass ein/e Angreifer/in Zeit und Ressourcen auf den Honeypot konzentriert und somit weniger Zeit und Ressourcen für sein/ihr eigentliches Ziel zur Verfügung hat. In Verbindung mit Monitoring und IDS-Funktionalitäten können Eindringlinge früher entdeckt und Angriffe rechtzeitig unterbunden werden.[5, pp. 13]

**Forschungshoneypots** dagegen werden betrieben um die Methoden von Cyberangriffen und Verhaltensmuster von den dahinterstehenden Personen zu erfassen und daraus zu lernen. [5, p. 14]

#### Level of Interaction

Bei der "Level of Interaction"-Klassifizierung werden Honeypots anhand ihrer Interaktionsmöglichkeit in drei verschiedene Level (High/Medium/Low) eingeteilt. [5, p. 14]

**Low-Interaction-Honeypots** stellen die simpelste Form eines Honeypots dar. Im Grunde handelt es sich um einen einfachen Listener, der auf einem bestimmten Port auf Netzwerkverkehr wartet. Der/Die Angreifer/in hat jedoch keine Möglichkeit mit dem System zu interagieren. [5, p. 15]

**Medium-Interaction-Honeypots** erweitern Low-Interaction-Honeypots um den Aspekt, dass sie ein Service vollständig abbilden. Der/Die Angreifer/in kann mit dem Service interagieren (z.B. Kommandos absetzen und eine Antwort vom Honeypot erhalten). [5, p. 16]

**High-Interaction-Honeypots** bieten den höchsten Level an Interaktionsmöglichkeiten. Bei diesen Honeypots kann der/die Angreifer/in mit dem Betriebssystem und darauf befindlichen Services interagieren. [5, p. 17] Die Tabelle 2.1 gibt einen Überblick über die Vor- und Nachteile der drei Arten in Bezug auf Aufwand, Nutzen und Risiko. Das Risiko bezieht sich dabei darauf, welche Möglichkeiten ein/e Angreifer/in im Fall einer Kompromittierung des Honeypots erlangt.

| Level of Interaction | Work to Configure Install and Configure | Work to Deploy and Mantain | Information Gathering | Level of Risk |
|---|---|---|---|---|
| Low | Easy | Easy | Limited | Low |
| Medium | Involved | Involved | Variable | Medium |
| High | Difficult | Difficult | Extensive | High |

Tabelle 2.1.: Vor- und Nachteile der Honeypot-Interaktionsklassen [3, p. 77]

### 2.1.4. Positionierung

Wie in der Abbildung 2.1 zu sehen ist, können Honeypots auf unterschiedliche Weise in ein Netzwerk eingebunden werden. Jede Position birgt seine eigenen Vor- und Nachteile (bzw. Risiken).

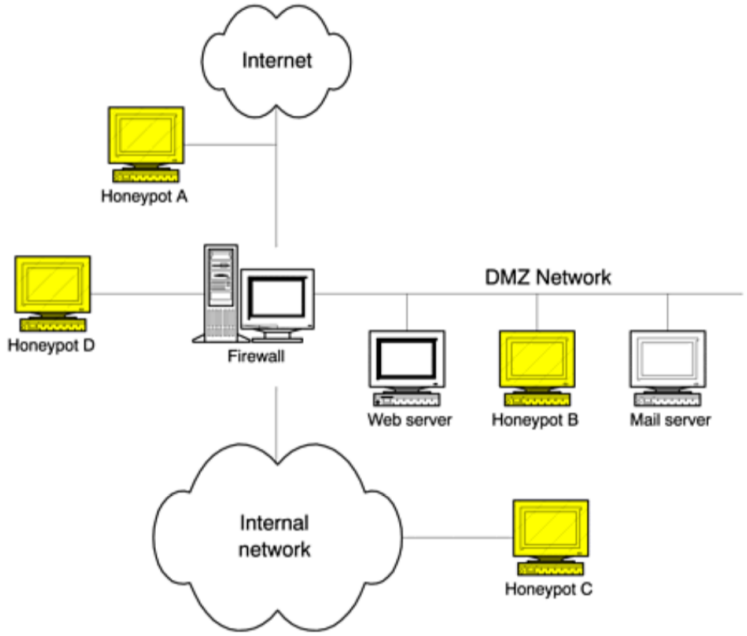



Abbildung 2.1.: Mögliche Positionierung von Honeypots [3, p. 287]

Durch die Positionierung vor einer Firewall wird sichergestellt, dass möglichst viele Personen das System bemerken und angreifen. Da durch diese Positionierung jedoch auch sämtliche Sicherheitsmechanismen eines Unternehmens ausgehebelt werden, ist diese Variante vor allem für reine Forschungshoneypots interessant. Ein Produktionshoneypot sollte eher in einem abgeschottetem Bereich wie einer DMZ oder einem eigenen Netzwerksegment betrieben werden. Diese Vorgehensweise reduziert Fehlalarme und unerwünschte Aufmerksamkeit. [3, pp. 286]

### 2.1.5. Anforderungen an Honeypots

Das Honeynet Project hat zur Entwicklung und den Betrieb von Honeypots (oder wie im Dokument beschrieben für Honeynets) Anforderungen definiert, die einen sicheren und kontrollierten Betrieb eines Honeypots gewährleisten soll. Die Anforderungen resultieren aus der Annahme, dass Honeypots kontinuierlich gescannt, angegriffen und kompromittiert werden. [6]

#### Data Control

Diese Anforderung beschreibt, dass der/die Betreiber/in des Honeypots in der Lage sein muss, Netzwerktraffic (zum oder vom Honeypot) im Bedarsfall kontrollieren zu können. Dabei geht es in erster Linie darum, im Fall einer Kompromittierung weiteren Schaden zu verhindern. Der/Die Angreifer/in soll die Kontroll-Maßnahmen jedoch nicht detektieren können. [6]

#### Data Capture

Alle relevanten Informationen, die durch den Honeypot generiert werden, müssen erfasst werden. Dabei darf der/die Angreifer/in die Monitoring-Maßnahmen nicht bemerken. [6]

#### Data Collection

Sofern es sich um ein verteiltes System handelt, müssen die erfassten Daten auf einer zentralen Instanz gesammelt werden. Dadurch können Daten über mehrere Endpunkte korreliert und analysiert werden. Auch hier sollte darauf Acht gegeben werden, dass die Übermittlung der Daten durch den/die Angreifer/in nicht erfasst werden kann. [6]

## 2.2. Industriesteueranlagen

### 2.2.1. Allgemein

Wie in der Abbildung 2.2 zu sehen, bestehen Industriesteueranlagen in der Praxis aus einer Vielzahl an Komponenten. Das Herzstück bilden die Programmable Logic Controller (PLC). Die PLCs sind dafür verantwortlich Daten von Sensoren zu empfangen, zu verarbeiten und auf Basis einer Programm-Logik Aktoren zu steuern (z.B. mittels Motor ein Förderband zu bewegen). Da dafür Kommunikation zwischen vielen verschiedenen Geräten notwendig ist, unterstützen PLCs eine Vielzahl an Protokollen. Ein Überblick über die relevanten Protokolle wird im Kapitel 2.2.2 gegeben.

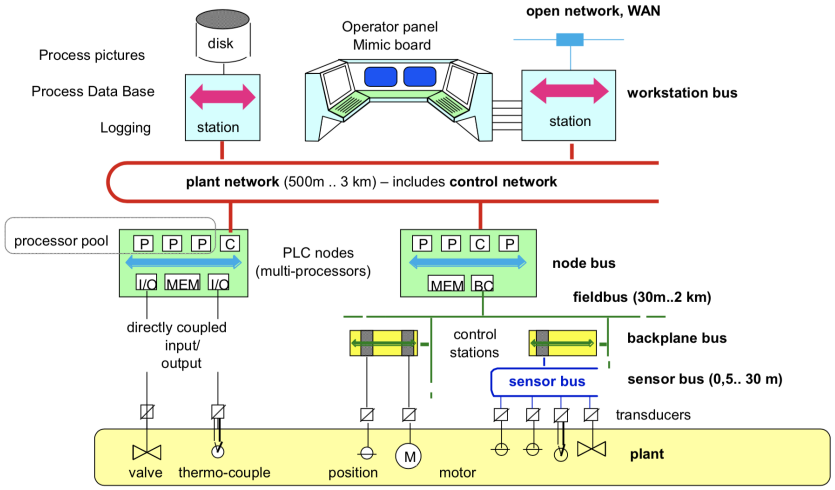



Abbildung 2.2.: Buses and processors in industrial plants[7]

Aufgrund der Tatsache, dass ein PLC nicht ohne zusätzliche Systeme (wie z.B. ein Human Machine Interface - HMI) kontrolliert bzw. ausgelesen werden kann und diese Systeme darüber hinaus auf die Authentizität der Rückmeldungen vertrauen müssen, ist ein Angriff auf die Protokolle der PLCs äußerst wirkungsvoll und kann schwer erkannt werden.

### 2.2.2. Protokolle

Je nach Einsatzgebiet werden auf PLCs unterschiedlichste Netzwerk-Protokolle eingesetzt. Auf der englischen Wikipedia sind unter der Rubrik “List of automation protocols” 87 Protokolle gelistet [8]. Wie im Kapitel 3.3 in der Tabelle 3.1 beschrieben, wurde als Referenzobjekt für diese Arbeit eine Siemens-PLC verwendet. Daher wird in den nachfolgenden Kapiteln speziell auf die Protokolle ISO-on-TCP (s7comm) und Modbus eingegangen. Diese beiden Protokolle werden standardmäßig von jeder Siemens-PLC untersützt. [9]

#### ISO-on-TCP (s7comm)

Das Protokoll ISO-on-TCP wurde 1987 als RFC 1006 spezifiziert. Es handelt sich dabei um eine Erweiterung des TCP Protokolls, um sogenannte ISO-Transport-Services (spezifiziert in ISO8072) über TCP abwickeln zu können. [10]

Auf Siemens PLCs wird das Protokoll verwendet, um die Funktionen TSEND_C und TRCV_C in der Programmierung nutzen zu können. Diese beiden Funktionen bilden die in der ISO8072 spezifizierten Services TCON, TDISCON. TSEND und TRCV ab. TCON und TDISCON sind dabei notwendig um eine Verbindung auf- und nach der Datenübertragung wieder abzubauen. ISO-on-TCP unterstützt eine

Paketlänge von maximal 8192 Bytes. [11]
Da im Zuge der Diplomarbeit eine POC-Implementierung auf Basis des Protokolls Modbus durchgeführt wird, werden an dieser Stelle keine Details zum Ablauf von ISO-on-TCP-Kommunikationen dargestellt. Diese Details sind in der RFC1006, ISO8072 sowie dem Systemhandbuch der Siemens-Steuerung beschrieben (siehe [10], [11] und [12]).

### Modbus

Das Modbus-Protokoll wurde 1979 spezifiziert und bildet einen Defacto-Standard für die Vernetzung von Komponenten in einem Industriesteuerungs-Netzwerk. Die Kommunikation findet dabei auf Basis eines "Client zu Server"-Modells statt. Eine Industriesteuerung kann dabei die Rolle eines Modbus-Servers oder Modbus-Clients übernehmen. Es existieren verschiedene Implementierungen, beispielsweise für TCP/IP, UDP sowie serielle Verbindungen. [13]
Im Zuge der Diplomarbeit wurde ausschließlich TCP/IP over Ethernet betrachtet. Modbus wird im TCP/IP-Modus standardmäßig über den reservierten Port TCP 502 betrieben. Es handelt sich dabei um eine reine Request/Response Kommunikation.
Grundsätzlich besteht jedes Modbus-Paket aus einem Function-Code sowie einem Daten-Block.
Der **Function-Code** ist in Form eines Bytes codiert. Valide Werte sind dabei 1 - 255 (wobei 128 - 255 für Exception-Responses reserviert sind).
Der **Daten-Block** enthält alle Informationen, welche für die Durchführung der übermittelten Funktion notwendig sind. Der Daten-Block kann jedoch auch 0 Byte lang sein, wenn beispielsweise keine zusätzlichen Informationen notwendig sind.
Der Server antwortet auf einen Request auch immer mit der Kombination aus Function Code und Daten-Block. Im Daten-Block werden die für den Client interessanten Daten übermittelt. Der Function-Code wird folgendermaßen gesetzt:

- Im Falle eines erfolgreichen Response wird der angefragte Function-Code rückübermittelt
- Im Falle eines Fehlers setzt der Server den Function-Code auf:

$$\text{“}requestedFunctionCode\text{''} + 128$$

Beispiel: Der Client schickt dem Server einen Request mit Function-Code 0x01 "Read Coils".
Konnte der Server den Request erfolgreich verarbeiten, sendet er dem Client als Function Code ebenfalls 0x01 zurück. Trat im Zuge der Verarbeitung jedoch ein Fehler auf, so antwortet der Server mit Function-Code 0x81.

Die Tabelle 2.2 zeigt die wichtigsten Function-Codes. Diese Function-Codes werden dafür verwendet Daten vom Server zu lesen oder zu schreiben. Im Proof of Concept wurden diese Function-Codes implementiert. Die involvierten Datentypen werden in der Tabelle 2.4 beschrieben.

| Function-Name | Function-Code | Beschreibung |
|---|---|---|
| Read Coils | 01 | Lesen des Datentyps Coil |
| Read Discrete Inputs | 02 | Lesen des Datentyps Discrete Input |
| Read Holding Registers | 03 | Lesen des Datentyps Holding Register |
| Read Input Registers | 04 | Lesen des Datentyps Input Register |
| Write Single Coil | 05 | Schreiben eines Wertes des Datentyps Coil |
| Write Single Register | 06 | Schreiben eines Wertes des Datentyps Holding Register |
| Write Multiple Coils | 15 | Schreiben mehrere Werte des Datentyps Coil |
| Write Mutliple Registers | 16 | Schreiben mehrere Werte des Datentyps Holding Register |

Tabelle 2.2.: Überblick über die wichtigsten Function Codes

### Modbus TCP/IP-Protokoll

Wie bereits beschrieben, besteht ein Modbus-Datenpaket aus einem Function-Code und einem Datenblock. Der Aufbau des Pakets kann der Abbildung 2.3 entnommen werden. Bei einem Transport über TCP-IP, wird dem Modbus-Paket noch ein MBAP-Header (Modbus Application Protocol Header) hinzugefügt. [14]

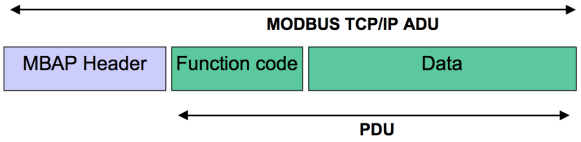



Abbildung 2.3.: Modbus request/response over TCP/IP

Der MBAP-Header hat eine Größe von 7 Byte und enthält die Informationen, die für die Modbus-Transaktionen relevant sind. Siehe dazu Tabelle 2.3. [14]

| Feld | Beschreibung |
|---|---|
| Transaction Identifier | ID für eine Request/Response-Transaktion. D.h. alle Pakete einer Transaktion haben die gleiche ID |
| Protocol Identifier | Wird für "intra-system multiplexing" verwendet. Ist bei MODBUS über TCP-IP immer 0 |
| Length | Anzahl der noch folgenden Bytes |
| Unit Identifier | ID des Slaves (nur relevant bei Bussystemen) |

Tabelle 2.3.: Darstellung der MBAP-Header-Felder

### Modbus Datamodel

Modbus bietet für die Abbildung der Daten vier verschiedene Datentypen. Diese unterscheiden sich grundsätzlich durch zwei Merkmalen:

1. Zugriffstyp: read-only oder read-write
2. Objekttyp: Single Bit oder 16-Bit Wort

| Bezeichnung | Objekttyp | Zugriffstyp |
|---|---|---|
| Coils | Single Bit | Read-Write |
| Discrete Input | Single Bit | Read-Only |
| Input Register | 16-Bit Wort | Read-Only |
| Holding Register | 16-Bit Wort | Read-Write |

Tabelle 2.4.: Darstellung der Modbus-Datentypen

Die Tabelle 2.4 zeigt die daraus resultierenden vier Datentypen. Man kann erkennen, dass jeder Datentyp über eigene Function Codes verfügt. Daher ist bereits durch den Aufruf eines Function Codes der Rückschluss auf den angefragten Datentypen möglich. Im Request wird innerhalb des Daten-Blocks vermerkt, welche Adresse(n) ausgelesen oder manipuliert werden sollen. Das Modbus-Protokoll sieht einen Adressraum von 0 - 65535 vor.

# 3. Konzept

## 3.1. Design der Architektur

### 3.1.1. Topologie

Ziel des Industrie-Honeypot ist es, eine effiziente und realistische Spiegelung einer Industrieanlagen (konkret der PLC - Programmable Logic Controller) durchzuführen und einem/einer Angreifer/in die Identifikation des produktiven PLCs zu erschweren. Die Abbildung 3.1 zeigt die einzelnen Schritte der Spiegelung inkl. der Interaktion mit dem/der Angreifer/in.

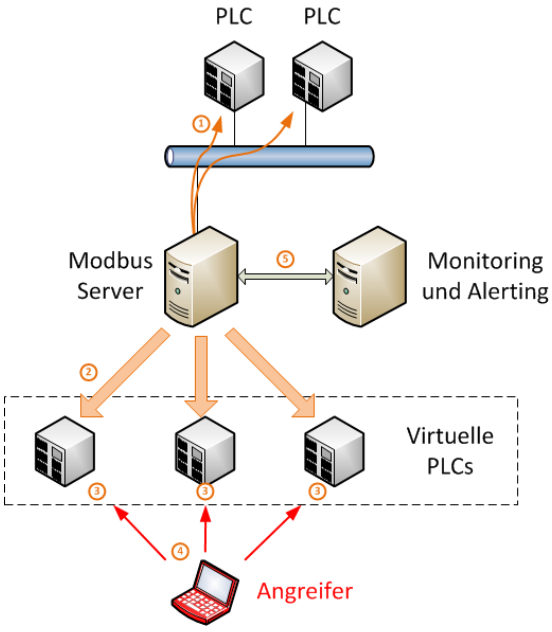



Abbildung 3.1.: Topologie des Honeypots

Der Honeypot stellt die Verbindung mit den echten Systemen, sowie den Datenabzug folgendermaßen her:

1. Verbindung mit der physischen PLC über ein IP-Netz (Modbus Protokoll). Die verfügbaren Ressourcen (Adressen) werden automatisch identifiziert. Die Modbus-Daten werden im Anschluss laufend aktualisiert.

2. Die Daten des PLCs werden regelmäßig erneuert und in einem virtuellen PLC zur Verfügung gestellt.

3. Die virtuellen PLCs verhalten sich wie echte PLCs und beantworten Anfragen mit authentischen Daten.

4. Der/Die Angreifer/in kann mit den Fallen (virtuellen PLCs) interagieren und sie in weiterer Folge attackieren.

5. Das Verhalten des/der Angreifers/Angreiferin kann über den Monitoring-Server analysiert werden. Auf Basis der generierten Logs kann auch eine Alarmierung implementiert werden.

### 3.1.2. Angebotene Dienste

Um Angreifer/innen zur Interaktion mit dem Honeypot zu verleiten, muss dieser relevante und authentische Daten anbieten. Das Ziel des Industrie-Honeypots ist die möglichst realistische Darstellung einer Industriesteueranlage im Produktiv-Einsatz. Aufgrund dieses Ziels müssen auch die bereitgestellten Daten realitätsnah sein. Die interagierenden Personen sollen im besten Fall nicht zwischen einer echten Anlage und dem Honeypot unterscheiden können. Daher sieht das Konzept vor, dass der Honeypot die Daten der Produktiv-Anlage in regelmäßigen Abständen (im Sekundenbereich) einliest und einem/einer potentiellen Angreifer/in präsentiert (siehe Kapitel 3.1.1).

Für einen Echtbetrieb sollten alle möglichen Daten (also durch alle bereitgestellten Protokolle, wie z.B. Modbus, ISO-on-TCP, etc.) abgebildet werden. Im Proof of Concept beschränkt sich diese Arbeit auf das Modbus-Protokoll. Es wird daher ein Modbus-Listener als Dienst zur Verfügung gestellt. Der Modbus-Dienst wird auf dem Modbus-Standardport (tcp 502) betrieben. Der Fokus wird in der Implementierung auf das Modbus-Protokoll gelegt, da es sich dabei um ein Protokoll mit hoher Verbreitung und offener Spezifikation handelt. [13]

Unter der Voraussetzung, dass ein ständiger Abgleich der Daten zwischen Honeypot und Industriesteuerung erfolgt, ist der Honeypot unmittelbar abhängig von der Verfügbarkeit der Industriesteuerung. Denkbar wäre allerdings auch, dass der Honeypot über einen definierten Zeitraum die Zustände der Industriesteuerung aufzeichnet und diese dann in einer Schleife wiedergibt. In diesem Modus wäre keine Abhängigkeit mehr gegeben. Der Nachteil ist jedoch, dass der Honeypot in regelmäßigen Abständen das Verhalten der Industriesteuerung aufzeichnen muss, um authentisch zu bleiben. Dieser Abgleich führt zu einem hohen Betriebsaufwand des Honeypots.

Für dieses Konzept wurde der Ansatz einer ständigen Verbindung gewählt. Mithilfe dieser Verbindung ist eine Spiegelung der Echtdaten möglich (ohne manuelle Eingriffe). Der Honeypot kann somit im Dauerbetrieb bleiben, ohne dass ein manueller Datenabgleich oder sonstige Wartungstätigkeiten notwendig sind.

Die derzeit aktuellste und aktiv in Entwicklung befindliche Honeypot-Implementierung namens Conpot ist Teil des Honeynet Projects. [15] Conpot bietet die Möglichkeit, einen Industriesteuerungs-Honeypot mit den Protokollen modbus, s7comm, http und snmp zu betreiben. Das Verhalten von Conpot (also welche Daten einem Angreifer übermittelt werden) muss durch ein XML-File konfiguriert werden. Die Konfigurationsdatei wird beim Start von Conpot eingelesen. Eine Änderung des XML-Files während Conpot läuft, ist nicht vorgesehen. [16]

Da das geplante Konzept eine Spiegelung einer echten Industriesteuerung vorsieht, ist dieser Ansatz nicht optimal. Um die notwendige Spiegelung abzubilden, wären weitreichende Anpassungen des Conpot-Sourcecodes notwendig. Aus diesem Grund wurde für den Proof of Concept der Weg einer Eigenentwicklung gewählt. Sofern das Konzept zu einem produktiven Honeypot weiterentwickelt werden sollte, muss man diesen Ansatz überdenken. Es ist davon auszugehen, dass der Conpot-Code weit ausgereifter ist als die PoC-Eigenentwicklung.

### 3.1.3. Data Control

Der Honeypot sollte in einem abgeschotteten Netzwerksegment positioniert werden. Optimal wäre eine Anbindung über zwei dedizierte LAN-Ports.

- Ein Port mit Zugriffsmöglichkeit auf die Industriesteuerungen (im Optimalfall kontrolliert über eine Firewall).
- Ein zweiter Port mit Verbindung in einen abgeschotteten Netzbereich. Dieser Netzbereich ist für den Zugriff durch einen Angreifer vorgesehen und bietet keinen direkten Netzwerkverbindungen zu den echten Industriesteuerungen.

Dieses Netzdesign schützt zum einen bestmöglich die kritische Infrastruktur, bietet jedoch auch größtmögliche Flexibilität. Es ist beispielsweise denkbar, in dem “Angriffsbereich” viele verschiedene virtuelle Honeypots zu betreiben, die auf der anderen Seite über einen singulären Zugriffspunkt die Daten von den physischen Industriesteuerungen holen. Dabei ist es auch mit wenig Aufwand möglich, den

Netzwerkverkehr zu den physischen Systemen zu analyisieren bzw. zu kontrollieren - wodurch eine Kompromitierung des Honeypot-Systems sehr schnell bemerkt werden kann. Durch eine Virtualisierung der Honeypots können diese im Bedarfsfall schnell vom Netzwerk getrennt und bei Bedarf auf einen Ausgangszustand zurückgesetzt werden.

### 3.1.4. Data Capture und Data Collection

Wie in Kapitel 3.1.1 dargestellt, ist ein dedizierter Server zur Speicherung von relevanten Log-Informationen vorgesehen. Es müssen daher alle Modbus-Verbindungen im Zuge der Verarbeitung an diesen zentralen Log-Server gesendet werden. Auch hier ist es sinnvoll, dass diese Verbindung über den Netzwerk-Bereich erfolgt, auf den der/die Angreifer/in keinen direkten Zugriff hat. Die Übermittlung der Angriffs-Aktionen wird im Zuge der PoC-Implentierung direkt in das Modbus-Protokoll programmiert. Es sollten keine zusätzlichen Programme oder Services notwendig sein.
Folgende Informationen müssen erfasst werden:

- Timestamp
- IP und Port des/der Angreifers/Angreiferin
- IP und Port des Honeypots
- Modbus-Function-Code
- Angefragte Adressen und Werte
- Verbindungsaufbau und -abbau
- Modbus-Antwortpaket des Honeypots

## 3.2. Anforderungen Hardware & Software

### 3.2.1. Virtualisierung

Für die PoC-Implementierung muss eine Hardware verwendet werden, die für Virtualisierung geeignet ist. Je nach Anzahl an Honeypots, sowie der Last können unterschiedliche Hardwareanforderungen notwendig sein. In der Minimal-Konstellation werden zwei virtuelle Maschinen (Honeypot und Logserver) benötigt.

### 3.2.2. Betriebssystem

Der PoC wurde auf Ubuntu Server 16.04. durchgeführt. Es sollten grundsätzlich die meisten Linux-basierenden Betriebssysteme verwendet werden können. Es sind keine speziellen Betriebssystem-Anforderungen gegeben.

### 3.2.3. Zusatzsoftware

Je nach Rolle des Servers sind hier unterschiedliche Softwarepakete notwendig. Der **Honeypot** wurde in der Programmiersprache python programmiert. Folgende Pakete werden benötigt:

- python 2.7 oder python 3
- elasticsearch==6.3.0 (Schnittstelle zu Elasticsearch)
- pwn==1.0 (Command-Line Logging für Testzwecke)
- pymodbus==1.5.2 (Modbus-Protokoll)
- pytz==2018.4 (Timezone-Paket für Cross-Platform Speicherung von timestamps)
- Twisted==18.4.0 (“event-driven networking engine“ für asynchrone Server)

Als zentrale **Log-Server** wurde das Produkt Elasticsearch und Kibana verwendet. Grundsätzlich ist hier keine spezielle Software-Version notwendig, es ist nur darauf zu achten, dass das verwendete python-elasticsearch-Paket mit der Version von Elasticsearch kompatibel ist. Im Zuge der PoC Implementierung wurden verschiedenste Releases von Elasticsearch verwendet (6.x.x).

#### Visualisierung

Die Visualisierung ist auf Basis von Kibana-Dashboards vorgesehen. Auch hier sind keine speziellen Anforderungen vorhanden. Es muss nur darauf geachtet werden, einen geeigneten Web-Browser zu verwenden.

## 3.3. Spezielle Anforderungen für die Proof-of-Concept-Implementierung

Um eine Industriesteuerung für eine/n Angreifer/in so realistisch wie möglich zu simulieren, ist es sinnvoll vor der Entwicklung die Rolle des Angreifers/der Angreiferin einzunehmen. Aus diesem Grund wurde eine Industrie-Steueranlage der Firma Siemens betrachtet. Die Tabelle 3.1 enthält die technischen Daten des Testobjekts.

| Typ | Siemens S7-1200 |
|---|---|
| Modell | CPU 1214AC AC/DC/RLY |
| Software (Programmierung) | TIA Portal V14 |

Tabelle 3.1.: Technische Daten der Test-PLC

Auf der Steuerung wurde die Modbus-Funktionalität über TCP/IP aktiviert. Damit können über den Standardport TCP 502 sämtliche Modbus-Transaktionen durchgeführt werden. Neben der Applikations-Sicht soll auch bereits in der Phase des Information-Gatherings die Existenz einer Industriesteuerung glaubwürdig vorgetäuscht werden. Konkret geht es darum, dass einem Angreifer/einer Angreiferin Informationen zu offenen Ports, Betriebssystem-Version, Standard-Services usw. durch den Honeypot übermittelt werden, die analog zu den Informationen einer echten Industriesteuerung aussehen.
In den weiteren Kapiteln wird die Test-PLC mit typische Angriffswerkzeugen angegriffen. Es handelt sich um die folgenden Tools (die Beschreibung der Tools erfolgt in den dazugehörigen Kapiteln):

- nmap
- modbusdetect
- modbus_findunitid
- modbusclient

Die dabei erhaltenen Informationen werden bei der Implementierung berücksichtigt. In Kapitel 5 wird der Honeypot mit den selben Tools getestet und verifiziert, ob das Verhalten mit dem Verhalten der Test-PLC übereinstimmt.

### 3.3.1. Information Gathering mittels nmap

Sobald ein/e Angreifer/in Zugang zu einem Netzwerk erlangt, muss dieser im ersten Schritt Informationen über das Netzwerk sammeln - ähnlich zu dem Vorgehen eines Penetration Testers. [17]

Ein weit verbreitetes Tool für diesen Schritt ist nmap (Network Mapper). nmap bietet die Möglichkeit ganze Netzwerke zu scannen und für jedes verbundene System offene Netzwerkports zu erkennen. Darüber hinaus ist es mit nmap möglich das Betriebssystem anhand des Netzwerkverkehrs einzuschätzen. [18]. Um das Verhalten der Industriesteuerung glaubwürdig zu simulieren, muss der Honeypot auf die selbe Weise auf eingehende Netzwerkverbindungen reagieren. Im Listing 3.1 ist zu erkennen welches Ergebnis nmap mit der Test-PLC liefert.

```
arp:~ ap\$ nmap -sV -p 1-65535 192.168.1.120
Starting Nmap 7.70 ( https://nmap.org ) at 2018-06-26 18:39 CEST
Nmap scan report for 192.168.1.120
Host is up (0.0076s latency).
Not shown: 65531 closed ports
PORT    STATE SERVICE   VERSION
102/tcp open  iso-tsap  Siemens S7 PLC
502/tcp open  mbap?
```

Listing 3.1: NMAP-Enumerierung einer Siemens S7-1200

### 3.3.2. Modbus-Angriffe mittels metasploit

Metasploit bezeichnet sich selbst als meist genutztes Pentrationtesting-Framework.[19] Aufgrund der weiten Verbreitung und der guten Dokumentation wurde metasploit für die Angriffe auf Protokoll-Ebene herangezogen. Es existieren derzeit drei relevante Module (modbusdetect, modbus_findunitid und modbusclient), welche auf der Siemens-PLC getestet wurden.

#### modbusdetect

modbusdetect wird dazu verwendet um auf einfache Weise herauszufinden, ob auf einem System Modbus aktiviert ist bzw. korrekte Modbus-Kommunikation stattfindet. Listing 3.2 zeigt das Ergebnis für die Test-PLC. Es ist zu sehen, dass modbusdetect einen korrekten Modbus-TCP-Header empfangen hat.

```
msf > use auxiliary/scanner/scada/modbusdetect
msf auxiliary(scanner/scada/modbusdetect) > show options

Module options (auxiliary/scanner/scada/modbusdetect):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   RHOSTS                    yes       The target address range or
   CIDR identifier
   RPORT    502              yes       The target port (TCP)
   THREADS  1                yes       The number of concurrent
   threads
   TIMEOUT  10               yes       Timeout for the network probe
   UNIT_ID  1                yes       ModBus Unit Identifier,
   1..255, most often 1
```

```
msf auxiliary(scanner/scada/modbusdetect) > set RHOSTS 192.168.1.120
RHOSTS => 192.168.1.120
msf auxiliary(scanner/scada/modbusdetect) > run

[+] 192.168.1.120:502     - 192.168.1.120:502 - MODBUS - received
   correct MODBUS/TCP header (unit-ID: 1)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Listing 3.2: Durchführung modbusdetect zu einer Siemens S7-1200

### modbus_findunitid

modbus_findunitid wird dazu verwendet um zu testen auf welche Modbus-UnitIDs die angefragte PLC reagiert. Sollte eine PLC nicht auf die gewählte UnitID hören, führt sie die übermittelte Funktion nicht aus, sondern antwortet mit einer Exception. Listing 3.3 zeigt das Ergebnis für die Test-PLC. Es ist zu sehen, dass im Falle der Siemens-S7 auf alle UnitIDs gehört wird. Diese Information ist für das nächste Metasploit-Modul essentiell.

```
msf > use auxiliary/scanner/scada/modbus_findunitid
msf auxiliary(scanner/scada/modbus_findunitid) > show options

Module options (auxiliary/scanner/scada/modbus_findunitid):

   Name          Current Setting  Required  Description
   ----          ---------------  --------  -----------
   BENICE        1                yes       Seconds to sleep between
   StationID-probes
   RHOST                          yes       The target address
   RPORT         502              yes       The target port (TCP)
   TIMEOUT       2                yes       Timeout for the network
   probe, 0 means no timeout
   UNIT_ID_FROM  1                yes       ModBus Unit Identifier
   scan from value
   UNIT_ID_TO    254              yes       ModBus Unit Identifier
   scan to value

msf auxiliary(scanner/scada/modbus_findunitid) > set RHOST
   192.168.1.120
RHOST => 192.168.1.120
msf auxiliary(scanner/scada/modbus_findunitid) > run

[+] 192.168.1.120:502 - Received: correct MODBUS/TCP from stationID
   1
[+] 192.168.1.120:502 - Received: correct MODBUS/TCP from stationID
   2
[SNIP]
[+] 192.168.1.120:502 - Received: correct MODBUS/TCP from stationID
   254
[*] Auxiliary module execution completed
```

Listing 3.3: Durchführung modbus_findunitid zu einer Siemens S7-1200

### modbusclient

modbusclient stellt ein voll funktionstüchtiges Tool zum Lesen und Schreiben von Daten von und zu PLCs zur Verfügung. In Listing 3.4 ist zu erkennen, dass mit Hilfe von modbusclient mit der Siemens-S7 Daten ausgetauscht werden können. Sowohl das Lesen, als auch das Schreiben von Daten sind möglich.

```
msf > use auxiliary/scanner/scada/modbusclient
msf auxiliary(scanner/scada/modbusclient) > show options

Module options (auxiliary/scanner/scada/modbusclient):

   Name            Current Setting  Required  Description
   ----            ---------------  --------  -----------
   DATA                             no        Data to write (
   WRITE_COIL and WRITE_REGISTER modes only)
   DATA_ADDRESS    0                yes       Modbus data address
   DATA_COILS                       no        Data in binary to write
    (WRITE_COILS mode only) e.g. 0110
   DATA_REGISTERS                   no        Words to write to each
   register separated with a comma (WRITE_REGISTERS mode only) e.g.
   1,2,3,4
   NUMBER          10               no        Number of coils/
   registers to read (READ_COILS ans READ_REGISTERS modes only)
   RHOST           192.168.1.120    yes       The target address
   RPORT           502              yes       The target port (TCP)
   UNIT_NUMBER     1                no        Modbus unit number


Auxiliary action:

   Name            Description
   ----            -----------
   READ_REGISTERS  Read words from several registers
msf auxiliary(scanner/scada/modbusclient) > set RHOST 192.168.1.120
RHOST => 192.168.1.120
msf auxiliary(scanner/scada/modbusclient) > set NUMBER 10
NUMBER => 10
msf auxiliary(scanner/scada/modbusclient) > set DATA_ADDRESS 0
DATA_ADDRESS => 0
msf auxiliary(scanner/scada/modbusclient) > set action READ_COILS
action => READ_COILS
msf auxiliary(scanner/scada/modbusclient) > run

[*] 192.168.1.120:502 - Sending READ COILS...
[+] 192.168.1.120:502 - 10 coil values from address 0 :
[+] 192.168.1.120:502 - [0, 1, 1, 0, 0, 1, 0, 0, 0, 0]
[*] Auxiliary module execution completed
msf auxiliary(scanner/scada/modbusclient) > set action READ_REGISTERS
action => READ_REGISTERS
msf auxiliary(scanner/scada/modbusclient) > run

[*] 192.168.1.120:502 - Sending READ REGISTERS...
[+] 192.168.1.120:502 - 10 register values from address 0 :
[+] 192.168.1.120:502 - [3, 8, 10, 42, 55, 10, 2, 2, 80, 56]
[*] Auxiliary module execution completed
```

Listing 3.4: Beispiel zur Verwendung von modbusclient zu einer Siemens S7-1200

# 4. Implementierung des Proof of Concept

Der Honeypot wurde in der Programmiersprache Python (python 2.7.13) implementiert und auf einem Linux-Server unter Ubuntu 16.04 getestet. Für die Programmierung des Honeypots wurde das Modul `pymodbus` verwendet. `pymodbus` ist eine vollständige Implementierung des Modbus-Protokolls und wird unter der BSD-Lizenz als Opensource-Projekt aktiv weiterentwickelt. Weiterführende Informationen zu `pymodbus` können aus [20] entnommen werden.

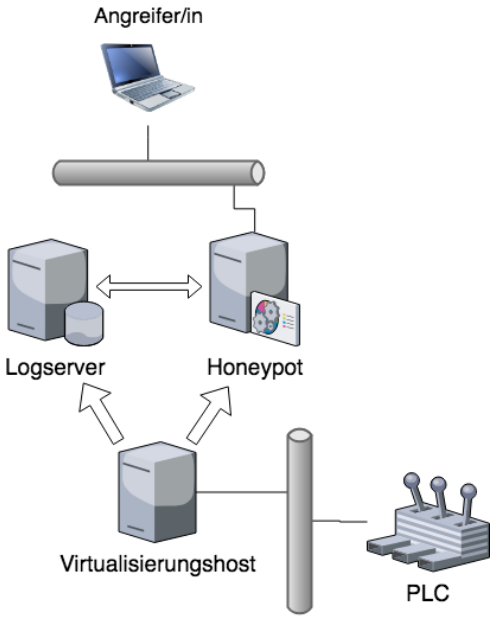



Abbildung 4.1.: Topologie des Proof-of-Concept

Die Abbildung 4.1 zeigt wie die einzelnen Komponenten miteinander interagieren. Auf dem Virtualisierungshost werden sowohl Logserver als auch Honeypot als virtuelle Maschienen betrieben. Die in den folgenden Kapiteln beschriebenen Softwareteile laufen alle auf dem Honeypotserver, welcher mit dem/der Angreifer/in interagiert und die gewonnenen Daten an den Logserver schickt. Über den zweiten Netzwerkanschluss wird eine Verbindung zu der Industriesteuerung (PLC) aufgebaut um die Daten zu synchronisieren.

## 4.1. Hauptprogramm

Im Proof of Concept wurden für den Honeypot zwei Betriebsmodi implementiert. Als primären Modus der “normale” Server und für Testzwecke ein “Demoserver”. Der Demoserver benötigt für die Ausführung keine Verbindung zu einem PLC. Um zwischen den beiden Modi wählen zu können und zusätzliche Optionen setzen zu können, wurde ein Kommandozeilenprogramm implementiert (main.py). Das Listing 4.1 zeigt die verfügbaren Optionen inklusive einer kurzen Beschreibung.

```
usage: main.py [-h] [-d] [-de] [-s] [-c C]

optional arguments:
  -h, --help    show this help message and exit
  -d, --demo    demo mode - no connection to PLC
  -de           enable elastic in demo mode
  -s, --silent  silent execution - no debug messages
  -c C          set path to configfile
```

Listing 4.1: Optionen des Kommandozeilenprogramms

Wie in Kapitel 3.2.3 beschrieben, wird als Logserver das Produkt Elasticsearch eingesetzt. Der oben beschriebene Demoserver loggt standardmäßig ausschließlich auf die Standardausgabe des Betriebssystems (meist die Kommandozeile). Um die Verbindung zwischen Honeypot und Logserver zu testen, wurde zusätzlich zum herkömmlichen Demoserver ein Demo-Modus (Option -de) implementiert, der die relevanten Informationen (siehe Kapitel 3.1.4) an den zentralen Logserver sendet.

### 4.1.1. Konfig-Load

Neben den Kommandozeilenoptionen kann der Honeypot über ein Konfig-File (JSON Notation) parametriert werden. Diese Parameter sind für den Normalbetrieb des Honeypots notwendig. Damit diese Informationen nicht bei jedem Start manuell über Kommandozeilenoptionen eingegeben werden müssen, wurde der Weg über ein Konfig-File gewählt. Das Listing 4.2 zeigt ein Beispiel eines Konfig-Files. Der Name der Datei muss immer config.json lauten.

```
{
  "delay": "10",
  "plc": "192.168.1.120:502",
  "max_requests": "125",
  "hp_address": "0.0.0.0:502",
  "elastic": ["192.168.1.231:9200"]
}
```

Listing 4.2: Beispiel eines Konfigfiles für den Honeypot

Die Konfiguration wird im Zuge Programmstarts geladen und als Dictionary gespeichert.

### 4.1.2. Sanity-Check

Das Konfigdictionary wird vor der Verwendung mit einem "Sanity-Check" verifiziert. Das Ziel ist, dass bereits vor dem Start des eigentlichen Honeypots Fehlkonfigurationen erkannt und korrigiert werden können.
Im Listing 4.3 ist die Hauptfunktion des Sanity-Checks zu sehen. Folgende Schritte werden sequentiell abgearbeitet:

1. Es wird überprüft ob das konfigurierte Delay größer-gleich Null ist. Das Delay ist notwendig um das Update-Intervall zu definieren, das für den Datenabgleich mit der PLC verwendet wird.

2. Im Anschluss muss die Erreichbarkeit der echten PLC verifiziert werden. Dazu wird IP und Port aus dem Konfigdictionary ausgelesen und an die Funktion `check_remote_service(ip, port)` übergeben. Die Funktion versucht mittels Socket-Connect zu der angegebenen IP und Port ein Socket zu öffnen. Im Erfolgsfall liefert die Funktion `True`, im Fehlerfall `False`.

3. Als nächstes wird überprüft, ob auf dem System mit den angegebenen IP-Informationen ein Service gestartet werden kann. Dazu wird die Funktion `check_local_service(ip, port)` verwendet. Es wird versucht mittels Socket-Bind ein lokales Service auf dem System zu erzeugen. Auch hier wird bei Erfolg `True` zurückgeliefert, ansonsten `False`.

4. Als letzter Schritt wird überprüft ob in der Konfiguration ein Wert für `max_requests` gesetzt ist. Bei diesem Wert handelt es sich um die Angabe, wie viele Adressen von der PLC innerhalb eines Function-Codes abgerufen werden können. Bei Der Test-PLC war dieser Wert zum Beispiel 125.

```
def sanity_check(config, silent):
    if not silent:
        p = log.progress("Checking configuration")
    sane = True
    hp_address = False
    plc = False
    max_requests = False

    if config["delay"] >= 0:
        if not silent:
            p.status("Checked if delay is set")
        if config["plc"]:
            if not silent:
                p.status("Checking if PLC-IP reachable")
            ip, port = config["plc"].split(":")
            plc = check_remote_service(ip, port)
        if config["hp_address"]:
            if not silent:
                p.status("Checking if Honeypot can be started on
  defined IP and/or Port")
```

```
20              ip, port = config["hp_address"].split(":")
21              hp_address = check_local_service(ip, port)
22          if config["max_requests"] >= 0:
23              if not silent:
24                  p.status("Checked if MAX-Requests is set")
25              max_requests = True
26      else:
27          if not silent:
28              log.critical("Delay must be >=0")
29          sane = False
30      if plc is False and sane is True:
31          if not silent:
32              log.critical("PLC-IP not reachable")
33          sane = False
34      if hp_address is False and sane is True:
35          if not silent:
36              log.critical("Honeypot can not be started on defined IP
    and/or Port")
37          sane = False
38      if max_requests is False and sane is True:
39          if not silent:
40              log.critical("MAX-Requests Option is not set")
41          sane = False
42
43      if sane:
44          p.success()
45      else:
46          p.failure()
47
48      return sane
```

Listing 4.3: Sanity-Check der Konfiguration

### Sanity-Check für Logserver-Verbindung

Zusätzlich zu dem grundsätzlichen Sanity-Check, der nur ausgeführt wird, wenn der Honeypot im Echtbetrieb gestartet wird, gibt es auch noch einen Check, ob die Verbindung zu dem zentralen Logserver hergestellt werden kann. Dieser Check ist in einer eigenen Funktion abgebildet, da der Honeypot einerseits auch ohne Verbindung zum zentralen Logserver betrieben werden kann (unter Einschränkungen). Zum anderen wird dieser Sanity-Check auch für den Demo-Modus benötigt. Die erwähnte Einschränkung besteht darin, dass ohne Verbindung zu einem Elasticsearch-Knoten, nur die Standardausgabe zur Darstellung der Logmeldungen verwendet wird. Im Listing 4.4 ist ein interessanter Auszug der Funktion zu sehen. Der Honeypot unterstützt nämlich nicht nur eine singuläre Elasticsearch-Instanz. In der Praxis kommt es häufig vor, dass mehrere Elasticsearch-Knoten zur horizontalen Skalierung verwendet werden. Daher können in der Konfiguration auch mehrere Knoten konfiguriert werden. Es wird daher in einer Schleife jeder konfigurierte Knoten mit der Funktion `check_elastic_service(ip, port)` überprüft. Sofern mindestens ein Knoten erreichbar ist wird ein gültiger Zustand der Elasticsearch-Konfiguration erreicht. Die Funktion `check_elastic_service(ip, port)` überprüft mittels der elasticsearch-Funktion `ping` ob der Cluster up&running ist.

```
[...]
for i in range(len(elastic_config)):
            ip, port = elastic_config[i].split(":")
            p.status("Try to reach Elastic-Node at %s:%s" % (ip, port
    ))
            elastic_config_sane[i] = check_elastic_service(ip, port)
        sane = any(item is True for item in elastic_config_sane)
[...]
```

Listing 4.4: Auszug Sanity-Check Logserver

## 4.2. Honeypot - Echtbetriebsmodus

In diesem Kapitel werden die essentiellen Teile der Implementation anhand von Sourcecode-Auszügen erläutert. Der vollständige Sourcecode des Honeypot-Echtbetriebsmoduses ist im Kapitel A.2 zu finden.

Das Modbus-Protokoll bietet die Möglichkeit einen Modbus-Server auf einer Industriesteueranlage zu betreiben. Ein Modbus-Server besteht aus 1-n Modbus-Slaves. Der Modbus-Server stellt die Daten der Slaves zur Verfügung und antwortet auf eingehende Anfragen. In unserem Fall bilden wir das Verhalten der verwendeten Industriesteueranlage ab. Die Anlage antwortet unabhängig davon welche “Unit-ID” angefragt wird, mit den verfügbaren Daten.

### 4.2.1. Datenspeicher

In Listing 4.5 wird ein neuer Modbus-Context angelegt. Der Master-Context bildet dabei die Datenstruktur eines Modbus-Servers ab. Der Slave-Context stellt die Datenstruktur eines Modbus-Slave dar. Für jede Datenart wird initial eine Datenadresse angelegt. Damit verfügt das Honeypot-Programm grundsätzlich über einen vollständigen Datenspeicher eines Modbus-Geräts. Die beiden Context-Objekte sind Klassen aus dem Paket `pymodbus.datastore`. Die Context-Ojekte werden mit je einem Datenspeicher (ModbusSequentialDataBlock) pro Modbus-Datentyp angelegt. Im Paket `pymodbus` sind für die Datastores und Context-Objekte set- und get-Methoden vorhanden, die die Modbus-Adressierung berücksichtigen. Die Option “single” sorgt dafür, dass der Master-Context unabhängig der angefragten “Unit-ID” auf Anfragen antwortet.

```
class Modpot:
    def __init__(self, log, delay, plc, max_requests):
        [...]
        self.__slave_store = ModbusSlaveContext( #Init a Context 1
   value per type
            di = ModbusSequentialDataBlock(0, [0]),
            co = ModbusSequentialDataBlock(0, [0]),
```

```
7               hr = ModbusSequentialDataBlock(0, [0]),
8               ir = ModbusSequentialDataBlock(0, [0]))
9           self.master_context = ModbusServerContext(slaves=self.
    __slave_store, single=True)
10          self.slave_context = self.master_context[self.unit_id]
11          [...]
```

Listing 4.5: Auszug: Initialisierung des Modbus-Context

Nachdem der Context angelegt und initial mit je einer Null befüllt wurde, wird eine Verbindung zum PLC aufgebaut, die gespiegelt werden soll. Dafür wird der synchrone TCP-Client des Pakets `pymodbus` verwendet. In dieser Klasse sind sämtliche Modbus-Function-Calls implementiert, die für den Verbindungsaufbau und die Abfrage von Daten notwendig sind.

### 4.2.2. Verbindung Honeypot zu PLC

Wie im Listing 4.6 zu sehen, wird für den Client die `pymodbus`-Klasse `ModbusTcpClient` verwendet. Mit dem Aufruf der Funktion `connect()` verbindet sich der Client mit der PLC. Ab diesem Zeitpunkt können Daten von der PLC gelesen und geschrieben werden. Im Zuge der Implementierung kam es jedoch hin und wieder zu Problemen, wenn der Honeypot zu schnell nach dem Verbindungsaufbau bereits versuchte Daten zu lesen. Daher wird nach dem initialen `connect()` ein `sleep(1)` ausgeführt. Dadurch wartet der Honeypot eine Sekunde, bevor er versucht Daten zu lesen. Dadurch kam es zu keinen Exceptions mehr.

```
1  class Modpot:
2      def __init__(self, log, delay, plc, max_requests):
3          [...]
4          try:
5              self.plclog = self.log.progress("Try to connect to PLC %s
       " % plc[0])
6              self.client = ModbusTcpClient(host=plc[0])
7              self.client.connect()
8              time.sleep(1)
9              self.client.read_coils(0,1)
10             self.plclog.success("Done")
11         except:
12             self.plclog.failure("Failed")
13         [...]
```

Listing 4.6: Auszug: Verbindungsaufbau zu PLC

Nachdem die Verbindung aufgebaut wurde, muss die Anzahl an Adressen pro Datentyp festgestellt werden. Zu diesem Zweck wurde die Funktion `__find_addresses()` implementiert. Dadurch kann sichergestellt werden, dass der Honeypot ausschließlich Anfragen auf Adressen beantwortet, die auch auf der PLC vorhanden sind. Des weiteren wird die Addressanzahl für die Aktualisierungs-Funktion benötigt. Wie in Listing 4.7 zu sehen, verhält sich die Funktion grundsätzlich wie eine binäre Suche. Sie sucht das Element an letzter Stelle, welches keine `Exception Response` zurückliefert. Der Algorithmus

wird für alle vier Datentypen durchgeführt und das Ergebnis in die Variablen der Honeypot-Klasse gespeichert.

```
def __find_addresses(self):
        for i in range(0, 4):
            highest = 9999
            nearest = 0
            self.addr_num[i] = highest
            addr_l = self.log.progress("Find %s-Addresses" % (self.
    addr_name[i]))
            while (highest - 1) != nearest:
                if i == 0: # CO
                    myresponse = self.client.read_coils(self.addr_num
    [i], 1)
                elif i == 1: # DI
                    myresponse = self.client.read_discrete_inputs(
    self.addr_num[i], 1)
                elif i == 2: # HR
                    myresponse = self.client.read_holding_registers(
    self.addr_num[i], 1)
                else: # IR
                    myresponse = self.client.read_input_registers(
    self.addr_num[i], 1)
                if myresponse.function_code >= self.EXCEPTION_OFFSET:
                    highest = self.addr_num[i]
                    self.addr_num[i] = highest - ((highest - nearest)
     / 2)
                else:
                    nearest = self.addr_num[i]
                    self.addr_num[i] = highest + ((highest - nearest)
     / 2)
            self.addr_num[i] -= 1
```

Listing 4.7: Auszug: Feststellung Anzahl an Datenadressen

Nachdem der Bedarf an Datenadressen festgestellt wurde, wird der Datenspeicher des Honeypots auf die notwendige Anzahl erweitert und mit Zufallswerten beschrieben. Dies geschieht in der Funktion `__randomize_context()`. Im Kapitel 4.2.3 wird das Vorgehen zur Aktualisierung der Daten beschrieben. Da das Vorgehen in der Funktion `__randomize_context()` sehr ähnlich ist, wird an dieser Stelle auf die Erklärung dieser Funktion verzichtet.

### 4.2.3. Laufender Datenabgleich Honeypot und PLC

Für den laufenden Datenabgleich zwischen Honeypot und PLC ist es notwendig die Daten der PLC über den gestarteten Modbus-Client einzulesen und den Modbus-Server zu aktualisieren. Dazu wird der Slave-Context mit den neuen Daten überschrieben. Der Update-Writer wird als dedizierter Thread ausgeführt und schläft nach der Aktualisierung für eine konfigurierbare Zeit (z.B. 10 Sekunden).

In Listing 4.8 ist die Funktion `__update_writer` beschrieben. Dieses auf den ersten Blick unnö-

tig komplizierte Vorgehen ist notwendig, da die PLC nur eine begrenzte Anzahl an Adressen innerhalb eines Function-Calls bedient (wie im Kapitel 4.1.2 bereits beschrieben im Fall der Test-PLC 125).
Daher können pro Datentyp mehrere Abfragen notwendig sein. Dabei ist jedoch darauf zu achten, dass bei dem letzten Request pro Datentyp die richtige Anzahl angefragt wird, da die PLC ansonsten mit einem `Exception Response` antwortet. Gelöst wurde dieser Umstand durch die Verwendung von mehreren Abfrage-Runden sowie einer Schluss-Abfrage mit dem Rest der Adressen (Modulo-Berechnung).
Um in der Kommandozeile erkennen zu können, wann die Werte aktualisiert wurden, wird aus allen Daten ein SHA256-Hash berechnet und ausgegeben. Dadurch kann überprüft werden, ob der Honeypot nach wie vor Aktualisierungen durchführt.

```
def __update_writer(self, logger):
    while 1:
        logger.status("Values are now updated")
        valuehash = hashlib.sha256()
        for i in range(0, 4):
            modulo = self.addr_num[i] \% self.MAX_REQUESTS
            rounds = self.addr_num[i] / self.MAX_REQUESTS
            plc_data = []
            x = 0
            if i == 0: # CO
                while x < rounds:
                    plc_data.extend((self.client.read_coils(x * self.
MAX_REQUESTS, self.MAX_REQUESTS)).bits)
                    x += 1
                plc_data.extend((self.client.read_coils(x * self.
MAX_REQUESTS, modulo)).bits)
                self.slave_context.setValues(self.FC_CO, 0, plc_data)
            elif i == 1: # DI
                while x < rounds:
                    plc_data.extend((self.client.read_discrete_inputs
(x * self.MAX_REQUESTS, self.MAX_REQUESTS)).bits)
                    x += 1
                plc_data.extend((self.client.read_discrete_inputs(x *
 self.MAX_REQUESTS, modulo)).bits)
                self.slave_context.setValues(self.FC_DI, 0, plc_data)
            elif i == 2: # HR
                while x < rounds:
                    plc_data.extend((self.client.
read_holding_registers(x * self.MAX_REQUESTS, self.MAX_REQUESTS)).
registers)
                    x += 1
                plc_data.extend((self.client.read_holding_registers(x
 * self.MAX_REQUESTS, modulo)).registers)
                self.slave_context.setValues(self.FC_HR, 0, plc_data)
            else: # IR
                while x < rounds:
                    plc_data.extend((self.client.read_input_registers
(x * self.MAX_REQUESTS, self.MAX_REQUESTS)).registers)
                    x += 1
                plc_data.extend((self.client.read_input_registers(x *
 self.MAX_REQUESTS, modulo)).registers)
                self.slave_context.setValues(self.FC_IR, 0, plc_data)
```

```
34          valuehash.update(str(self.slave_context.getValues(self.FC_DI,
35                               0, count=self.addr_num[1])) +
36                           str(self.slave_context.getValues(self.FC_CO,
37                               0, count=self.addr_num[0])) +
38                           str(self.slave_context.getValues(self.FC_HR,
39                               0, count=self.addr_num[2])) +
40                           str(self.slave_context.getValues(self.FC_IR,
41                               0, count=self.addr_num[3])))
42          logger.status("Values updated - SHA256-Hash: " + valuehash.
    hexdigest())
43          time.sleep(self.delay)
```

Listing 4.8: Update-Funktion des Honeypots

### 4.2.4. Start der Server-Komponente

Da nun alle notwendigen Teile und Funktionen für den Honeypot vorhanden sind, wird eine Start-Funktion benötigt um die notwendigen Schritte in der richtigen Reihenfolge auszuführen. Der Honeypot wird aus dem Hauptprogramm heraus gestartet, indem eine Instanz der Modpot-Klasse erzeugt wird. In diesem Schritt wird der im Kapitel 4.2.1 erwähnte Datenspeicher erzeugt und initialisiert. Anschließend wird die Klassen-Funktion `start_modpot` aufgerufen. Im Listing 4.9 ist die Funktion beschrieben. Wie zu sehen ist, ruft diese Funktion sequentiell die bereits erläuterten Funktionen `__find_addresses` und `__randomize_context` auf.

Im Anschluss wird die Funktion `__start_server` aufgerufen. Hier wird nun die Funktion `__update_writer` in einem Thread ausgeführt und `startModpotTcpServer` aufgerufen. Diese Hilfsfunktion startet einen asynchronen TCP-Server als Daemon-Thread. Verwendet wird dabei das `Twisted`-Paket. [21]

Das Modbus-Applikationsprotokoll wird dabei durch die beiden Klassen `ModbusSocketFramer` und `ModpotServerFactory` bereitgestellt. Der Framer hat dabei die Aufgabe den MBAP-Header (siehe Kapitel 2.2.2) abzubilden. Die `ModpotServerFactory` kapselt das Modbus-Protokoll und dekodiert die Byte-Ströme in die Datenstrukturen. Die Grund-Klassen werden durch das `pymodbus`-Paket zur Verfügung gestellt. [20]

```
1  class Modpot:
2      [...]
3      def start_modpot(self, hp_addr, elastic=False, e_addr=None):
4          self.p.status("Starting Modpot")
5          try:
6              self.__find_addresses()
7              self.vars = self.log.progress("Modbus Data-Update")
8              self.__randomize_context(self.vars)
9              self.conns = self.log.progress("Last request/response")
10             self.__start_server(elastic, self.p, e_addr, hp_addr)
11         except Exception as e:
12             self.p.failure("Failed - %s" % e)
```

```
    [...]
    def __start_server(self, elastic, logger, e_addr=None, address=("
0.0.0.0:50002")):
        try:
            ip, port = address.split(":")
            address = ip, int(port)
            self.t = threading.Thread(target=self.__update_writer,
args=[self.vars])
            self.t.daemon = True
            self.t.start()
            startModpotTcpServer(self.master_context, elastic, self.
conns, e_addr, address=address)
            logger.status("TCP-Server started")
        except:
            logger.failure("Can\'t start TCP-Server")
    [...]

def startModpotTcpServer(context, elastic, conns_logger, e_addr=None,
    identity=None, address=None, **kwargs):
    from twisted.internet import reactor
    from pymodbus.transaction import ModbusSocketFramer

    address = address
    framer = ModbusSocketFramer
    factory = protocols.ModpotServerFactory(elastic, e_addr,
conns_logger, context, framer, identity, **kwargs)
    reactor.listenTCP(address[1], factory, interface=address[0])
    reactor_thread = threading.Thread(target=reactor.run, args=(False
,))
    reactor_thread.daemon = True
    reactor_thread.start()
```

Listing 4.9: server.py: Start des Honeypots

### 4.2.5. ModpotServerFactory - Logging der Vorgänge

Da für die Server-Komponente ein asynchroner TCP-Server auf Basis des `Twisted`-Pakets verwendet wird, ist für die Abbildung des Modbus-Protokolls eine sogenannte `ServerFactory` erforderlich. `pymodbus` bietet für das Modbus-Protokoll bereits die Implementation `ModbusServerFactory`. Das Honeypot-Konzept sieht jedoch vor, dass jede Modbus-Interaktion mit dem Honeypot an den Logserver übermittelt werden muss. Daher wurden die beiden Klassen `ModbusServerFactory` sowie `ModbusTcpProtocol` erweitert.

Im Listing 4.10 ist die Klasse `ModpotServerFactory` ersichtlich. Die wichtigste Änderung in der abgeleiteten Klasse ist das Setzen der Variable `protocol`. Diese Variable zeigt auf eine Klasse, die von der ServerFactory zum Erzeugen des Protokolls verwendet wird. Der Server benötigt das Protokoll für eingehende Verbindungen.

```
class ModpotServerFactory(ModbusServerFactory):
    protocol = ModpotTcpProtocol

```

```
4     def __init__(self, elastic, e_addr, var_logger, store, framer=
    None, identity=None, **kwargs):
5         ModbusServerFactory.__init__(self, store, framer, identity,
    **kwargs)
6         if elastic:
7             self.protocol.elastic_address=e_addr
8         if var_logger:
9             self.protocol.var_logger = var_logger
```

Listing 4.10: ModpotServerFactory

Die Klasse `ModpotTcpProtocol` wurde von `ModbusTcpProtocol` abgeleitet und um die Funktionalität erweitert, eingehende und ausgehende Nachrichten an einen zusätzlichen Server weiterzuleiten. Die erste relevante Aktion ist der Verbindungsaufbau zum Honeypot. Daher wurde die Funktion `connectionMade()` überschrieben. Diese Funktion wird aufgerufen, sobald ein Client versucht eine Verbindung zum Server aufzubauen.

Listing 4.11 zeigt die relevanten Codestellen. Da die Daten zum zentralen Logserver geschickt werden müssen, wird als erster Schritt die Verbindung zu Elasticsearch konfiguriert. Dabei werden die IP-Daten hinterlegt und der Elasticsearch-Index gesetzt. Im Anschluss werden die IP-Daten von Honeypot und Client gespeichert, da diese Informationen in nachfolgenden Verbindungs-Stati nicht mehr ausgelesen werden können. Die Informationen werden zusammen mit der Aktion (also in diesem Fall “Client connected”) und einem Timestamp in ein Dictionary gespeichert. Dieses Dictionary wird in JSON-Format mittels der Funktion `log_Event(my_doc)` an den Elasticsearch-Knoten geschickt. Anschließend wird die Funktion der “parent class” aufgerufen, welche die Daten an den Client sendet.

```
def connectionMade(self):
    if self.elastic_address:
        self.setupElastic()
    self._honeypot_ip = self.transport.getHost().host
    self._honeypot_port = self.transport.getHost().port
    self._client_ip = self.transport.getPeer().host
    self._client_port = self.transport.getPeer().port

    my_doc = {
        'Honeypot-IP' : self._honeypot_ip,
        'Honeypot-Port' : self._honeypot_port,
        'Client-IP' : self._client_ip,
        'Client-Port' : self._client_port,
        'Action' : 'Client connected',
        'timestamp' : vienna.localize(datetime.now()),
    }
    self.log_Event(my_doc)
    ModbusTcpProtocol.connectionMade(self)

def setupElastic(self):
    try:
        self.es = Elasticsearch([{"host": str(self.elastic_address
   [0]), "port": str(self.elastic_address[1])}])
        self.my_index = "modpot" #Index for Elasticsearch
        self.ELASTIC_ENABLED = True
    except:
        self.ELASTIC_ENABLED = False

def sendtoElastic(self, my_doc):
    try:
        self.es.index(index=self.my_index, doc_type='modbus', body=
   my_doc) #write the event to Elasticsearch
    except Exception as e:
        self.ELASTIC_ENABLED = False
        self.log_Event(my_doc)
```

Listing 4.11: ModpotTcpProtocol: connectionMade() und setupElastic()

Neben der Funktion `connectionMade()` wurden die Funktionen `connectionLost(reason)`, `_execute(request)` und `_send(message)` überschrieben. Da sich `connectionLost(reason)` kaum von `connectionMade()` unterscheidet, wird an dieser Stelle verzichtet auf den Code näher einzugehen. Interessant sind die anderen beiden Funktionen.

Im Listing 4.12 sind in gekürzter Form die relevanten Code-Stellen ersichtlich (die Daten im Dictionary sind mit `[...]` abgekürzt und können im Anhang A.3 nachgelesen werden). Aufgrund der Tatsache, dass die Variablen-Namen je nach Function-Code unterschiedlich sind, bzw. in verschiedenen Zuständen der Verbindung nur gewisse Informationen verfügbar sind, mussten viele Spezialfälle implementiert werden bzw. Informationen wie Modbus-Adressen, oder IP-Adressen zwischengespeichert werden.

`_execute` wird aufgerufen, sobald ein Modbus-Function-Call vom Client übermittelt wird. Dies stellt einen Client-Request dar. Somit sind in diesem Zustand unterschiedliche Informationen möglich, da zwischen lesenden und schreibenden Function-Calls unterschieden werden muss und zusätzlich noch unterschiedliche Datentypen vorhanden sein können (values und value). `_send` wird nach der Ausfüh-

rung von `_execute` aufgerufen, sobald der Server Daten an den Client senden muss. Um nicht nur die eingehenden Requests auf dem Logserver zu speichern werden in dieser Funktion auch die ausgehende Nachrichten mit den relevanten Verbindungsinformationen an den Logserver gesendet. Das Modbus-Paket liegt zu diesem Zeitpunkt bereits in strukturierter Form ohne String-Darstellung vor. Mit Hilfe der Funktion `b2a_hex` wird der Inhalt des Speicherbereichs in einen String in Hexadezimal-Darstellung konvertiert und gespeichert. `b2a_hex` ist im Paket `binascii` der Python-Standardbibliothek enthalten.

```
def _execute(self, request):
    try:
        my_address = request.address
    except AttributeError:
        my_address = 'None'
    try:
        my_count = request.count
    except AttributeError:
        my_count = 'None'
    try:
        my_byte_count = request.byte_count
    except AttributeError:
        my_byte_count = 'None'
    try:
        my_values = request.values
    except AttributeError:
        my_values = 'None'
    try:
        my_value = request.value
    except AttributeError:
        my_value = 'None'
    if request.function_code < 5:
        my_doc = { [...] }
    elif request.function_code == 16:
        my_doc = { [...] }
    [...]
    elif request.function_code == 5 or request.function_code == 6:
        my_doc = { [...] }
    else:
        my_doc = { [...] }
    self.log_Event(my_doc)
    ModbusTcpProtocol._execute(self, request)

def _send(self, message):
    pdu = self.framer.buildPacket(message)
    modbus_packet = str(b2a_hex(pdu))
    if message.function_code <= 2:
        my_doc = { [...] }
    elif message.function_code == 3 or message.function_code == 4:
        my_doc = { [...] }
    [...]
    elif message.function_code == 15 or message.function_code == 16:
        my_doc = { [...] }
    else:
        my_doc = { [...] }
    self.log_Event(my_doc)
    ModbusTcpProtocol._send(self, message)
```

Listing 4.12: ModpotTcpProtocol Auzug: _execute und _send

## 4.3. Honeypot - Demomodus

Der Demomodus entspricht im Großen und Ganzen der Implementierung des Echtbetriebsmodus, wie im Kapitel 4.2 beschrieben. Dieser Modus unterscheidet sich nur in folgenden Punkten:

- Der Demomodus startet auf dem fest konfiguriertem Port tcp 502 und antwortet auf allen verhandenen IP-Adressen des Servers (IP-Adresse 0.0.0.0).
- Es wird keine Verbindung zu einer PLC aufgebaut.
- Es werden pro Datentyp zehn Adressen angelegt.
- Die Datenadressen werden alle zehn Sekunden mit zufälligen Werten überschrieben.

Da alle relevanten Punkte der Implementierung inklusive Code-Stellen bereits im Kapitel 4.2 beschrieben wurden, wird an dieser Stelle auf weitere Details verzichtet. Der vollständige Code kann dem Anhang A.4 entnommen werden.

Die Abbildung 4.2 zeigt die Ausführung des Honeypots im Demomodus.

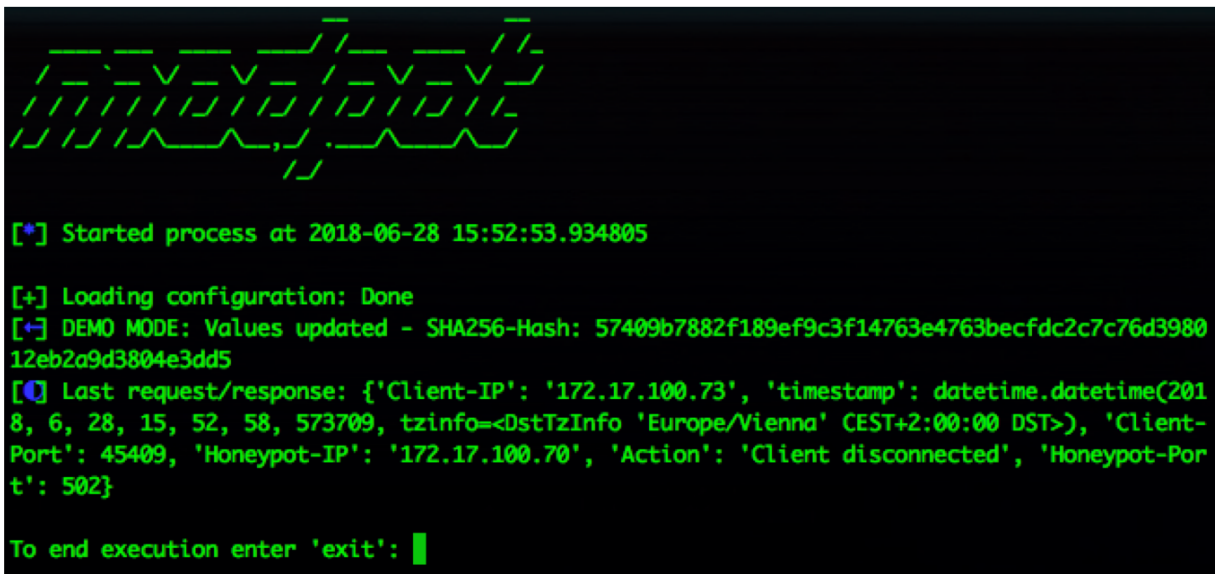



Abbildung 4.2.: Ausführung des Honeypots im Demomodus

# 5. Test des Honeypots

Um überprüfen zu können, ob sich der Honeypot genauso verhält wie eine echte Industriesteuerung, wurde der Proof-of-Concept mit den selben Methoden überprüft die in Kapitel 3.3 bereits beschrieben wurden.

## 5.1. Vergleich mittels nmap

Bei der Überprüfung der echten Test-PLC (siehe Kapitel 3.3.1) lieferte nmap das Ergebnis wie in Listing 5.1 angeführt.

```
arp:~ ap\$ nmap -sV -p 1-65535 192.168.1.120
Starting Nmap 7.70 ( https://nmap.org ) at 2018-06-26 18:39 CEST
Nmap scan report for 192.168.1.120
Host is up (0.0076s latency).
Not shown: 65531 closed ports
PORT    STATE SERVICE   VERSION
102/tcp open  iso-tsap  Siemens S7 PLC
502/tcp open  mbap?
```

Listing 5.1: NMAP-Enumerierung einer Siemens S7-1200

Da im Zuge der Honeypot-Implementierung ausschließlich das Protokoll Modbus implementiert wurde, ist absehbar, dass der Port tcp 102 auf dem Honeypot als `open` angezeigt wird. In einer Produktivumgebung sollte daher zumindest auch das Protokoll iso-tsap implementiert werden. Das Listing 5.2 zeigt das Ergebnis für den Honeypot.

```
arp:~ ap\$ nmap -sV -p 1-65535 172.17.100.70
Starting Nmap 7.70 ( https://nmap.org ) at 2018-06-28 16:26 CEST
Nmap scan report for 172.17.100.70
Host is up (0.000064s latency).
Not shown: 65534 closed ports
PORT    STATE SERVICE VERSION
502/tcp open  mbap?
```

Listing 5.2: NMAP-Enumerierung des Honeypots

## 5.2. Vergleich mittels metasploit

An dieser Stelle wird darauf verzichtet, die Kommandozeilenausgaben der abgesetzten Befehle gegenüber der Test-PLC nochmals anzuführen. Diese sind im Kapitel 3.3.2 angeführt.
In Listing 5.3 ist zu erkennen, dass die Ausführung des metasploit-Moduls `modbusdetect` das gleiche Ergebnis liefert, wie bei der Ausführung auf der Test-PLC.

```
msf auxiliary(scanner/scada/modbusdetect) > set RHOSTS 172.17.100.70
RHOSTS => 172.17.100.70
msf auxiliary(scanner/scada/modbusdetect) > run

[+] 172.17.100.70:502     - 172.17.100.70:502 - MODBUS - received
   correct MODBUS/TCP header (unit-ID: 1)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Listing 5.3: Durchführung modbusdetect zum Honeypot

In Listing 5.4 wird das metasploit-Modul `modbus_findunitid` ausgeführt. Der Honeypot antwortet, wie erwünscht auf sämtliche angefragte stationIDs bzw. UNIT-IDs mit den erwarteten Antworten.

```
msf > use auxiliary/scanner/scada/modbus_findunitid
msf auxiliary(scanner/scada/modbus_findunitid) > set RHOST
   172.17.100.70
RHOST => 172.17.100.70
msf auxiliary(scanner/scada/modbus_findunitid) > run

[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   1
[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   2
[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   3
[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   4
[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   5
[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   6
[SNIP]
[+] 172.17.100.70:502 - Received: correct MODBUS/TCP from stationID
   254
[*] Auxiliary module execution completed
```

Listing 5.4: Durchführung modbus_findunitid zum Honeypot

In Listing 5.5 und 5.6 wird mittels dem metasploit-Modul `modbusclient` mit dem Honeypot interagiert. Dabei werden die Aktionen `READ_COILS` sowie `READ_REGISTERS` ausgeführt um die beiden Datentypen abzufragen. Die Ergebnisse sehen für den Angreifer oder die Angreiferin in beiden Fällen identisch mit den Ergebnissen der Test-PLC aus.

```
msf > use auxiliary/scanner/scada/modbusclient
msf auxiliary(scanner/scada/modbusclient) > set RHOST 192.168.1.228
RHOST => 192.168.1.228
msf auxiliary(scanner/scada/modbusclient) > set DATA_ADDRESS 0
DATA_ADDRESS => 0
msf auxiliary(scanner/scada/modbusclient) > set NUMBER 10
NUMBER => 10
msf auxiliary(scanner/scada/modbusclient) > set ACTION READ_COILS
ACTION => READ_COILS
msf auxiliary(scanner/scada/modbusclient) > run

[*] 192.168.1.228:502 - Sending READ COILS...
[+] 192.168.1.228:502 - 10 coil values from address 0 :
[+] 192.168.1.228:502 - [1, 1, 0, 0, 1, 0, 1, 0, 1, 0]
[*] Auxiliary module execution completed
```

Listing 5.5: Durchführung READ_COILS zum Honeypot

```
msf > use auxiliary/scanner/scada/modbusclient
msf auxiliary(scanner/scada/modbusclient) > set RHOST 192.168.1.228
RHOST => 192.168.1.228
msf auxiliary(scanner/scada/modbusclient) > set DATA_ADDRESS 0
DATA_ADDRESS => 0
msf auxiliary(scanner/scada/modbusclient) > set NUMBER 10
NUMBER => 10
msf auxiliary(scanner/scada/modbusclient) > set ACTION READ_REGISTERS
ACTION => READ_REGISTERS
msf auxiliary(scanner/scada/modbusclient) > run

[*] 192.168.1.228:502 - Sending READ REGISTERS...
[+] 192.168.1.228:502 - 10 register values from address 0 :
[+] 192.168.1.228:502 - [327, 704, 815, 692, 760, 730, 5, 185, 998,
   412]
[*] Auxiliary module execution completed
```

Listing 5.6: Durchführung READ_REGISTERS zum Honeypot

# 6. Fazit

Das Ziel dieser Arbeit war die folgenden beiden Forschungsfragen (siehe Kapitel 1) zu beantworten:

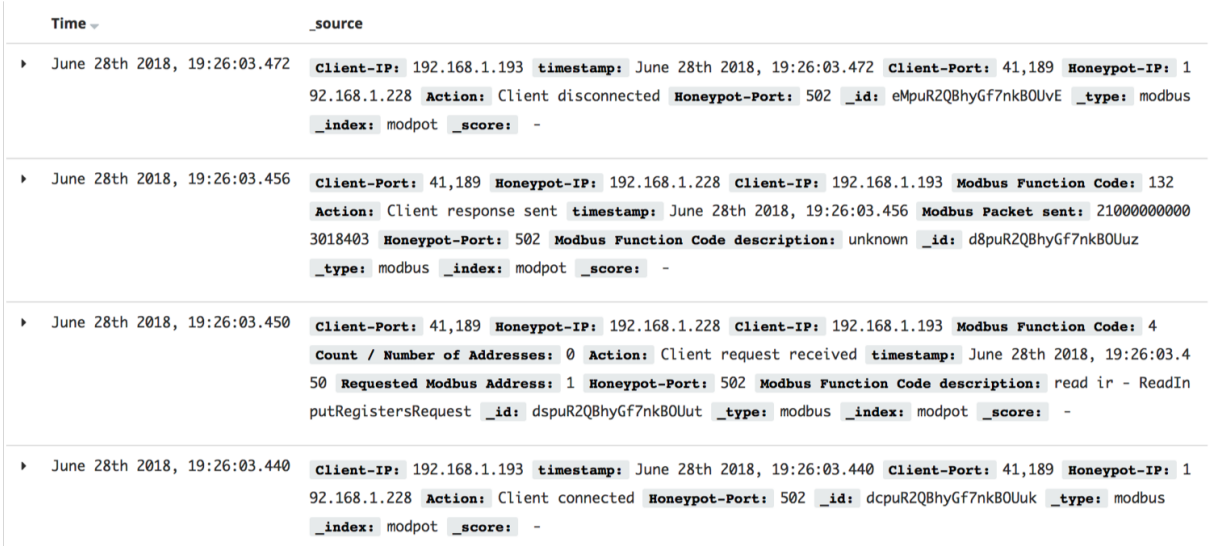
1. Wie sollte ein Honeypot für Industriesteuerungen konzipiert werden?
   Dabei werden Erkenntnisse und State-of-the-Art-Empfehlungen für herkömmliche Honeypots auf Industrie-Honeypots angewendet.

2. Wie ist ein solches Konzept technisch realisierbar?
   Dies wird anhand einer Proof-of-Concept-Implementierung (für das weit verbreitete Protokoll Modbus) und anschließenden Tests validiert.

Ein Industrie-Honeypot sollte die Empfehlungen bzw. Anforderungen des Honeynet-Projekts (State-of-the-Art-Empfehlungen) erfüllen und daneben großen Augenmerk auf eine realitätsnahe Simulation eines PLC richten. Die in Kapitel 3 definierten Anforderungen erfüllen diese Kriterien.

Die Realisierbarkeit über das Modbus-Protokoll ist grundsätzlich gegeben. Mit Standard-Angriffswerkzeugen kann die Simulation nicht "bemerkt" werden. Im Kapitel 6.1 wird aufgezeigt, in welchen Details die Simulation noch verbessert werden kann. Im Kapitel 4 wurde dargestellt, wie das Konzept in einem Proof-of-Concept umgesetzt wurde.

Im Konzept wurde eine zentrale Log-Komponente vorgesehen, welche die generierten Logs entgegennimmt, speichert und in einem Kibana-Dashboard darstellt. Dies bietet den Vorteil, dass auf Basis dieser Softwarelösung das Verhalten bei Angriffen analysiert, korreliert und bei Bedarf alarmiert werden kann. Die Abbildung 6.1 zeigt die gespeicherten Daten bei der Ausführung des metasploit-Modus `modbusdetect`.



| Time | _source |
|---|---|
| June 28th 2018, 19:26:03.472 | Client-IP: 192.168.1.193 timestamp: June 28th 2018, 19:26:03.472 Client-Port: 41,189 Honeypot-IP: 192.168.1.228 Action: Client disconnected Honeypot-Port: 502 _id: eMpuR2QBhyGf7nkBOUvE _type: modbus _index: modpot _score: - |
| June 28th 2018, 19:26:03.456 | Client-Port: 41,189 Honeypot-IP: 192.168.1.228 Client-IP: 192.168.1.193 Modbus Function Code: 132 Action: Client response sent timestamp: June 28th 2018, 19:26:03.456 Modbus Packet sent: 21000000000 3018403 Honeypot-Port: 502 Modbus Function Code description: unknown _id: d8puR2QBhyGf7nkBOUuz _type: modbus _index: modpot _score: - |
| June 28th 2018, 19:26:03.450 | Client-Port: 41,189 Honeypot-IP: 192.168.1.228 Client-IP: 192.168.1.193 Modbus Function Code: 4 Count / Number of Addresses: 0 Action: Client request received timestamp: June 28th 2018, 19:26:03.4 50 Requested Modbus Address: 1 Honeypot-Port: 502 Modbus Function Code description: read ir - ReadIn putRegistersRequest _id: dspuR2QBhyGf7nkBOUut _type: modbus _index: modpot _score: - |
| June 28th 2018, 19:26:03.440 | Client-IP: 192.168.1.193 timestamp: June 28th 2018, 19:26:03.440 Client-Port: 41,189 Honeypot-IP: 192.168.1.228 Action: Client connected Honeypot-Port: 502 _id: dcpuR2QBhyGf7nkBOUuk _type: modbus _index: modpot _score: - |

Abbildung 6.1.: Analyse über Kibana - Netzwerkverkehr von modbusdetect

## 6.1. Aufgetretene Schwierigkeiten und weiterführende Forschung

Die größte Schwierigkeit im Zuge der Arbeit bestand darin, die Test-PLC korrekt zu konfigurieren, um das Modbus-Protokoll zu aktivieren und Daten über dieses Protokoll abfragen zu können.

Ein zweites großes Problem trat im Zuge der Implementierung auf. Zuvor funktionierender Code führte zu einem späteren Zeitpunkt zu Ausführungsfehlern. Nach einer Reihe von Tests stellte sich heraus, dass es bei der Verbindung zu der Test-PLC und darauf folgenden Datenabfragen zu einer Race-Condition kam. Anscheinend führte der Honeypot die Anfragen zu schnell aus, so dass die Test-PLC auf die ersten Anfragen mit Exception-Responses antwortete. Gelöst würde das Problem dadurch, dass nach dem Verbindungsaufbau ein sleep für eine Sekunde ausgeführt wird. Nach Umsetzung dieser Maßnahme kam es zu keinen Exception-Responses mehr.

Um den Proof-of-Concept zu einem Honeypot für den Produktiveinsatz weiterzuentwickeln sind zahlreiche Protokoll-Implementierung notwendig. Bei Siemens-PLCs ist jedenfalls das Protokoll iso-tsap notwendig, da es standardmäßig verwendet wird. Je nach Hersteller und Anwendungsfall sind weitere Protokoll-Implementationen notwendig.

# A. Sourcecode Modpot

## A.1. main.py

```
#!/usr/bin/python

from pwn import *
import random
from datetime import datetime
import argparse
import json
import threading
import Queue
import sys
import time
import demoserver
import server
import socket
from elasticsearch import Elasticsearch

def main():
    silent = False
    config_path = "./config.json"

    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--demo", help="demo mode - no
   connection to PLC", action="store_true")
    parser.add_argument("-de", help="enable elastic in demo mode",
   action="store_true")
    parser.add_argument("-s", "--silent", help="silent execution - no
    debug messages", action="store_true")
    parser.add_argument("-c", help="set path to configfile")
    parser.parse_args()
    opts = parser.parse_args()

    if opts.silent:
        silent = True
    if opts.c is not None:
        config_path = opts.c
    if not silent:
        starttime = start()
    try:
        config = load_config(config_path, silent)
    except:
        config = None
    # Check if Demo Mode should be started
    if opts.demo:
        addr="0.0.0.0", 502
        e_addr = None
        elastic = False
```

```
        if opts.de:
            if config is None:
                ip, addr = "localhost", 9200
                if check_elastic_service(ip, addr) is True:
                    e_addr = ip, addr
                else:
                    log.critical("Elasticsearch not reachable -
    Fallback to stdout")
            else:
                elastic = sanity_elastic_check(config, silent)
                e_addr = config["elastic"][0].split(":")
        demopot = demoserver.Demopot(log)
        demopot.start_demomode(addr, elastic, e_addr)
        userkill()
    # Start the real Honeypot - no Demo Mode
    else:
        sane = sanity_check(config, silent)
        elastic = sanity_elastic_check(config, silent)
        if sane and elastic:
            start_emulation_mode()
            userkill()
        else:
            if sane:
                modpot = server.Modpot(log, config["delay"], config["
    plc"].split(":"), config["max_requests"])
                modpot.start_modpot(config["hp_address"])
                userkill()
            else:
                log.critical("Can't start Honeypot - Config-Failure")

    if not silent:
        end(starttime)


def load_config(confpath, silent):
    if not silent:
        p = log.progress("Loading configuration")
    config = json.load(open(confpath))
    if not silent:
        p.success()
    return config

def sanity_elastic_check(config, silent):
    if not silent:
        p = log.progress("Checking Elastic configuration")
    sane = False
    if config["elastic"]:
        if not silent:
            p.status("")
        elastic_config = config["elastic"]
        elastic_config_sane = [False] * len(elastic_config)
        for i in range(len(elastic_config)):
            ip, port = elastic_config[i].split(":")
            p.status("Try to reach Elastic-Node at %s:%s" % (ip, port
    ))
            elastic_config_sane[i] = check_elastic_service(ip, port)
        sane = any(item is True for item in elastic_config_sane)
    if sane is False:
        if not silent:
            p.failure("Done")
            log.critical("Elasticsearch not reachable - Fallback to
    stdout")
```

```
102     p.success("Done")
103     return sane
104 
105 def sanity_check(config, silent):
106     if not silent:
107         p = log.progress("Checking configuration")
108     sane = True
109     hp_address = False
110     plc = False
111     max_requests = False
112 
113     # Check ob config okay - ips & Ports erreichbar
114     if config["delay"] >= 0:
115         if not silent:
116             p.status("Checking if delay ist set")
117         if config["plc"]:
118             if not silent:
119                 p.status("Checking if PLC-IP reachable")
120             ip, port = config["plc"].split(":")
121             plc = check_remote_service(ip, port)
122         if config["hp_address"]:
123             if not silent:
124                 p.status("Checking if Honeypot can be started on
    defined IP and/or Port")
125             ip, port = config["hp_address"].split(":")
126             hp_address = check_local_service(ip, port)
127         if config["max_requests"] >= 0:
128             if not silent:
129                 p.status("Checking if MAX-Requests is set")
130             max_requests = True
131 
132     if plc is False:
133         if not silent:
134             log.critical("PLC-IP not reachable")
135         sane = False
136     if hp_address is False:
137         if not silent:
138             log.critical("Honeypot can not be started on defined IP
    and/or Port")
139         sane = False
140     if max_requests is False:
141         if not silent:
142             log.critical("MAX-Requests Option is not set")
143         sane = False
144 
145     if sane:
146         p.success()
147     else:
148         p.failure()
149 
150     return sane
151 
152 def check_elastic_service(ip, port):
153     try:
154         es = Elasticsearch([{"host": ip, "port": int(port)}])
155         if not es.ping():
156             return False
157         return True
158     except:
159         return False
160 
161 
```

```
162 def check_local_service(ip, port):
163     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
164     try:
165         sock.bind((ip, int(port)))
166         sock.listen(1)
167     except socket.error:
168         return False
169     sock.close()
170     return True
171 
172 def check_remote_service(ip, port):
173     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
174     try:
175         sock.connect((ip, int(port)))
176     except socket.error:
177         return False
178     sock.close()
179     return True
180 
181 def start():
182     print_banner()
183     starttime = datetime.now()
184     log.info("Started process at %s" % starttime)
185     log.indented("")
186     return starttime
187 
188 def start_emulation_mode():
189     p = log.progress("Emulation MODE")
190     counter = 0
191     while counter < 20000:
192         if random.randint(1, 10) == 2:
193             counter += 1
194             p.status("Count - %s" % counter)
195     p.success("Done")
196 
197 
198 def userkill():
199     a = ""
200     print("")
201     sys.stdout.write("To end execution enter 'exit': ")
202     input_queue = Queue.Queue()
203     input_thread = threading.Thread(target=add_input, args=(
    input_queue,))
204     input_thread.daemon = True
205     input_thread.start()
206 
207     while "exit" not in a.lower():
208         if not input_queue.empty():
209             a = input_queue.get()
210             if "exit" not in a.lower():
211                 sys.stdout.write("To end execution enter 'exit': ")
212         time.sleep(0.1)
213 
214 
215 def add_input(input_queue):
216     while True:
217         input_queue.put(sys.stdin.readline())
218 
219 
220 def end(starttime):
221     endtime = datetime.now()
222     log.indented("")
```

```
    log.info("Finished at %s" % endtime)
    log.info("Runtime was %s" % (endtime - starttime))


def print_banner():
    print "                        __            __ "
    print "   ____ ___  ____  ____/ /___  ____  / /_"
    print "  / __ `__ \/ __ \/ __  / __ \/ __ \/ __/"
    print " / / / / / / /_/ / /_/ / /_/ / /_/ / /_  "
    print "/_/ /_/ /_/\____/\__,_/ .___/\____/\__/  "
    print "                     /_/                 "
    print ""


if __name__ == "__main__":
    main()
```

Listing A.1: Sourcecode Modpot - main.py

## A.2. server.py

```
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext,
   ModbusServerContext
from pymodbus.client.sync import ModbusTcpClient
import threading
import time
import random
import hashlib
import protocols

#This Class is the Honeypot; reads data from PLC and presents them to
    a attacker
class Modpot:
    def __init__(self, log, delay, plc, max_requests):
        self.EXCEPTION_OFFSET = 0x80  # this is the exception_offset,
    see pymodbus pdu.py - ExceptionResponse
        self.unit_id=0x00  # We only have one unit, so this does not
   matter
        self.delay = float(delay)  # Delay for update-loop in seconds
        self.addr_name = ("CO", "DI", "HR", "IR")
        self.addr_num = [0]*4
        self.MAX_REQUESTS = int(max_requests)
        self.FC_CO = 1  # Function-Code for Coils
        self.FC_DI = 2  # Function-Code for Dicrete Inputs
        self.FC_HR = 3  # Function-Code for Holding Registers
        self.FC_IR = 4  # Function-Code for Input Registers
        self.__slave_store = ModbusSlaveContext( #Init a Context with
    10 values per type
            di = ModbusSequentialDataBlock(0, [0]),
            co = ModbusSequentialDataBlock(0, [0]),
            hr = ModbusSequentialDataBlock(0, [0]),
            ir = ModbusSequentialDataBlock(0, [0]))
        self.master_context = ModbusServerContext(slaves=self.
   __slave_store, single=True)
        self.slave_context = self.master_context[self.unit_id]
        self.log = log
        self.p = self.log.progress("Honeypot")
        self.client = ModbusTcpClient(host=plc[0])  # TODO: try catch
    hier einbauen
        self.client.connect()
        self.vars = None
        self.conns = None

    def __del__(self):
        try:
            self.client.close()
            self.p.success("Done")
        except:
            self.p.failure("Done")

    def start_modpot(self, hp_addr, elastic=False, e_addr=None):
        self.p.status("Starting Modpot")
        self.__find_addresses()
        self.vars = self.log.progress("Modbus Data-Update")
        self.__randomize_context(self.vars)
        self.conns = self.log.progress("Last request/response")
        self.__start_server(elastic, self.p, e_addr, hp_addr)

    def __find_addresses(self):
```

```
53          fa = self.log.progress("Find Modbus-Address-Counts on PLC")
54          for i in range(0, 4):
55              highest = 9999
56              nearest = 0
57              self.addr_num[i] = highest
58              addr_l = self.log.progress("Find %s-Addresses" % (self.
    addr_name[i]))
59              while (highest - 1) != nearest:
60                  if i == 0: # CO
61                      myresponse = self.client.read_coils(self.addr_num
    [i], 1)
62                  elif i == 1: # DI
63                      myresponse = self.client.read_discrete_inputs(
    self.addr_num[i], 1)
64                  elif i == 2: # HR
65                      myresponse = self.client.read_holding_registers(
    self.addr_num[i], 1)
66                  else: # IR
67                      myresponse = self.client.read_input_registers(
    self.addr_num[i], 1)
68                  if myresponse.function_code >= self.EXCEPTION_OFFSET:
69                      highest = self.addr_num[i]
70                      self.addr_num[i] = highest - ((highest - nearest)
     / 2)
71                  else:
72                      nearest = self.addr_num[i]
73                      self.addr_num[i] = highest + ((highest - nearest)
     / 2)
74              self.addr_num[i] -= 1
75              addr_l.success("Done -> %s" % (self.addr_num[i]))
76          fa.success("Done")
77
78      def __randomize_context(self, logger):
79          random.seed()
80          valuehash = hashlib.sha256()
81          di_randoms = [random.randint(0, 1) for p in range(0, self.
    addr_num[1])]
82          co_randoms = [random.randint(0, 1) for p in range(0, self.
    addr_num[0])]
83          hr_randoms = [random.randint(0, 999) for p in range(0, self.
    addr_num[2])]
84          ir_randoms = [random.randint(0, 999) for p in range(0, self.
    addr_num[3])]
85          self.slave_context.setValues(self.FC_CO, 0, co_randoms)
86          self.slave_context.setValues(self.FC_DI, 0, di_randoms)
87          self.slave_context.setValues(self.FC_HR, 0, hr_randoms)
88          self.slave_context.setValues(self.FC_IR, 0, ir_randoms)
89          valuehash.update(str(self.slave_context.getValues(self.FC_DI,
     0, count=self.addr_num[1])) +
90                           str(self.slave_context.getValues(self.FC_CO,
     0, count=self.addr_num[0])) +
91                           str(self.slave_context.getValues(self.FC_HR,
     0, count=self.addr_num[2])) +
92                           str(self.slave_context.getValues(self.FC_IR,
     0, count=self.addr_num[3])))
93          logger.status("Values updated - SHA256-Hash: " + valuehash.
    hexdigest())
94
95      def __start_server(self, elastic, logger, e_addr=None, address=("
    0.0.0.0:50002")):
96          try:
97              ip, port = address.split(":")
```

```
98              address = ip, int(port)
99              self.t = threading.Thread(target=self.__update_writer,
    args=[self.vars])
100             self.t.daemon = True
101             self.t.start()
102             startModpotTcpServer(self.master_context, elastic, self.
    conns, e_addr, address=address)
103             logger.status("TCP-Server started")
104         except:
105             logger.failure("Can\'t start TCP-Server")
106
107     # this function is executed as a thread
108     def __update_writer(self, logger):
109         while 1:
110             logger.status("Values are now updated")
111             valuehash = hashlib.sha256()
112             for i in range(0, 4):
113                 modulo = self.addr_num[i] % self.MAX_REQUESTS
114                 rounds = self.addr_num[i] / self.MAX_REQUESTS
115                 plc_data = []
116                 x = 0
117                 if i == 0: # CO
118                     while x < rounds:
119                         plc_data.extend((self.client.read_coils(x *
    self.MAX_REQUESTS, self.MAX_REQUESTS)).bits)
120                         x += 1
121                     plc_data.extend((self.client.read_coils(x * self.
    MAX_REQUESTS, modulo)).bits)
122                     #self.log.critical("Update Values for CO: " + str
    (plc_data))
123                     self.slave_context.setValues(self.FC_CO, 0,
    plc_data)
124                 elif i == 1: # DI
125                     while x < rounds:
126                         plc_data.extend((self.client.
    read_discrete_inputs(x * self.MAX_REQUESTS, self.MAX_REQUESTS)).
    bits)
127                         x += 1
128                     plc_data.extend((self.client.read_discrete_inputs
    (x * self.MAX_REQUESTS, modulo)).bits)
129                     #self.log.critical("Update Values for DI: " + str
    (plc_data))
130                     self.slave_context.setValues(self.FC_DI, 0,
    plc_data)
131                 elif i == 2: # HR
132                     while x < rounds:
133                         plc_data.extend((self.client.
    read_holding_registers(x * self.MAX_REQUESTS, self.MAX_REQUESTS)).
    registers)
134                         x += 1
135                     plc_data.extend((self.client.
    read_holding_registers(x * self.MAX_REQUESTS, modulo)).registers)
136                     #self.log.critical("Update Values for HR: " + str
    (plc_data))
137                     self.slave_context.setValues(self.FC_HR, 0,
    plc_data)
138                 else: # IR
139                     while x < rounds:
140                         plc_data.extend((self.client.
    read_input_registers(x * self.MAX_REQUESTS, self.MAX_REQUESTS)).
    registers)
141                         x += 1
```

```
142                     plc_data.extend((self.client.read_input_registers
    (x * self.MAX_REQUESTS, modulo)).registers)
143                     #self.log.critical("Update Values for IR: " + str
    (plc_data))
144                     self.slave_context.setValues(self.FC_IR, 0,
    plc_data)
145             valuehash.update(str(self.slave_context.getValues(self.
    FC_DI, 0, count=self.addr_num[1])) +
146                              str(self.slave_context.getValues(self.
    FC_CO, 0, count=self.addr_num[0])) +
147                              str(self.slave_context.getValues(self.
    FC_HR, 0, count=self.addr_num[2])) +
148                              str(self.slave_context.getValues(self.
    FC_IR, 0, count=self.addr_num[3])))
149             logger.status("Values updated - SHA256-Hash: " +
    valuehash.hexdigest())
150             time.sleep(self.delay)
151
152 # Helper Function to start the Modpot Class as a Server; see the
    twisted documentation
153 def startModpotTcpServer(context, elastic, conns_logger, e_addr=None,
     identity=None, address=None, **kwargs):
154     from twisted.internet import reactor
155     from pymodbus.transaction import ModbusSocketFramer
156
157     address = address
158     framer = ModbusSocketFramer
159     factory = protocols.ModpotServerFactory(elastic, e_addr,
    conns_logger, context, framer, identity, **kwargs)
160     reactor.listenTCP(address[1], factory, interface=address[0])
161     reactor_thread = threading.Thread(target=reactor.run, args=(False
    ,))
162     reactor_thread.daemon = True
163     reactor_thread.start()
```

Listing A.2: Sourcecode Modpot - server.py

## A.3. protocols.py

```
from pymodbus.server.async import ModbusServerFactory,
   ModbusTcpProtocol
from pytz import timezone
from datetime import datetime
from binascii import b2a_hex
from elasticsearch import Elasticsearch
#Setup timezone for logging
vienna = timezone('Europe/Vienna')

#This Class implements the ModbusTcpProtocol for our Modpot. It's
   needed because we want to log what the attacker does.
class ModpotTcpProtocol(ModbusTcpProtocol):

    #These are the functions we implemented for logging to a
   Elasticsearch-Cluster
    possible_functions = {1: 'read co - ReadCoilsRequest', 2: 'read
   di - ReadDiscreteInputsRequest',
                          3: 'read hr - ReadHoldingRegistersRequest',
    4: 'read ir - ReadInputRegistersRequest',
                          5: 'write co - WriteSingleCoilRequest', 6:
   'write hr - WriteSingleRegisterRequest',
                          15: 'write co - WriteMultipleCoilsRequest',
    16: 'write hr - WriteMultipleRegistersRequest'}

    ELASTIC_ENABLED = False
    var_logger = None
    elastic_address = None

    def setupElastic(self):
        try:
            self.es = Elasticsearch([{"host": str(self.
   elastic_address[0]), "port": str(self.elastic_address[1])}])
            self.my_index = "modpot" #Index for Elasticsearch
            self.ELASTIC_ENABLED = True
        except:
            self.ELASTIC_ENABLED = False

    def sendtoElastic(self, my_doc):
        try:
            self.es.index(index=self.my_index, doc_type='modbus',
   body=my_doc) #write the event to Elasticsearch
        except Exception as e:
            self.ELASTIC_ENABLED = False
            self.log_Event(my_doc)


    def log_Event(self, my_doc):
        if self.ELASTIC_ENABLED:
            self.sendtoElastic(my_doc)
        elif self.var_logger:
            self.var_logger.status(str(my_doc))
        else:
            print(my_doc)


    def connectionMade(self):
        if self.elastic_address:
            self.setupElastic()
        self._honeypot_ip = self.transport.getHost().host
```

```
51          self._honeypot_port = self.transport.getHost().port
52          self._client_ip = self.transport.getPeer().host
53          self._client_port = self.transport.getPeer().port
54
55          my_doc = {
56              'Honeypot-IP' : self._honeypot_ip,
57              'Honeypot-Port' : self._honeypot_port,
58              'Client-IP' : self._client_ip,
59              'Client-Port' : self._client_port,
60              'Action' : 'Client connected',
61              'timestamp' : vienna.localize(datetime.now()),
62          }
63          self.log_Event(my_doc)
64          ModbusTcpProtocol.connectionMade(self)
65
66      def connectionLost(self, reason):
67          my_doc = {
68              'Honeypot-IP': self._honeypot_ip,
69              'Honeypot-Port': self._honeypot_port,
70              'Client-IP': self._client_ip,
71              'Client-Port': self._client_port,
72              'Action': 'Client disconnected',
73              'timestamp': vienna.localize(datetime.now()),
74          }
75          self.log_Event(my_doc)
76          ModbusTcpProtocol.connectionLost(self,reason)
77
78      def _execute(self, request):
79          try:
80              my_address = request.address
81          except AttributeError:
82              my_address = 'None'
83          try:
84              my_count = request.count
85          except AttributeError:
86              my_count = 'None'
87          try:
88              my_byte_count = request.byte_count
89          except AttributeError:
90              my_byte_count = 'None'
91          try:
92              my_values = request.values
93          except AttributeError:
94              my_values = 'None'
95          try:
96              my_value = request.value
97          except AttributeError:
98              my_value = 'None'
99
100         if request.function_code < 5:
101             my_doc = {
102                 'Honeypot-IP': self._honeypot_ip,
103                 'Honeypot-Port': self._honeypot_port,
104                 'Client-IP': self._client_ip,
105                 'Client-Port': self._client_port,
106                 'Action': 'Client request received',
107                 'timestamp': vienna.localize(datetime.now()),
108                 'Modbus Function Code': request.function_code,
109                 'Modbus Function Code description': self.
    possible_functions.get(request.function_code),
110                 'Requested Modbus Address': my_address,
111                 'Count / Number of Addresses': my_count,
```

```
112             }
113         elif request.function_code == 16:
114             my_doc = {
115                 'Honeypot-IP': self._honeypot_ip,
116                 'Honeypot-Port': self._honeypot_port,
117                 'Client-IP': self._client_ip,
118                 'Client-Port': self._client_port,
119                 'Action': 'Client request received',
120                 'timestamp': vienna.localize(datetime.now()),
121                 'Modbus Function Code': request.function_code,
122                 'Modbus Function Code description': self.
    possible_functions.get(request.function_code),
123                 'Requested Modbus Address': my_address,
124                 'Count / Number of Addresses': my_count,
125                 'Data to write': my_values,
126             }
127         elif request.function_code == 15:
128             my_doc = {
129                 'Honeypot-IP': self._honeypot_ip,
130                 'Honeypot-Port': self._honeypot_port,
131                 'Client-IP': self._client_ip,
132                 'Client-Port': self._client_port,
133                 'Action': 'Client request received',
134                 'timestamp': vienna.localize(datetime.now()),
135                 'Modbus Function Code': request.function_code,
136                 'Modbus Function Code description': self.
    possible_functions.get(request.function_code),
137                 'Requested Modbus Address': my_address,
138                 'Count / Number of Addresses': my_byte_count,
139                 'Data to write': my_values,
140             }
141         elif request.function_code == 5 or request.function_code ==
    6:
142             my_doc = {
143                 'Honeypot-IP': self._honeypot_ip,
144                 'Honeypot-Port': self._honeypot_port,
145                 'Client-IP': self._client_ip,
146                 'Client-Port': self._client_port,
147                 'Action': 'Client request received',
148                 'timestamp': vienna.localize(datetime.now()),
149                 'Modbus Function Code': request.function_code,
150                 'Modbus Function Code description': self.
    possible_functions.get(request.function_code),
151                 'Requested Modbus Address': my_address,
152                 'Data to write': my_value,
153             }
154         else:
155             my_doc = {
156                 'Honeypot-IP': self._honeypot_ip,
157                 'Honeypot-Port': self._honeypot_port,
158                 'Client-IP': self._client_ip,
159                 'Client-Port': self._client_port,
160                 'Action': 'Client request received',
161                 'timestamp': vienna.localize(datetime.now()),
162                 'Modbus Function Code': request.function_code,
163                 'Modbus Function Code description': 'unknown',
164             }
165         self.log_Event(my_doc)
166         ModbusTcpProtocol._execute(self, request)
167 
168     def _send(self, message):
169         pdu = self.framer.buildPacket(message)
```

```
170         modbus_packet = str(b2a_hex(pdu))
171
172         if message.function_code <= 2:
173             my_doc = {
174                 'Honeypot-IP': self._honeypot_ip,
175                 'Honeypot-Port': self._honeypot_port,
176                 'Client-IP': self._client_ip,
177                 'Client-Port': self._client_port,
178                 'Action': 'Client response sent',
179                 'timestamp': vienna.localize(datetime.now()),
180                 'Modbus Function Code': message.function_code,
181                 'Modbus Function Code description': self.
    possible_functions.get(message.function_code),
182                 'Value-Data sent': message.bits,
183                 'Modbus Packet sent': modbus_packet,
184             }
185         elif message.function_code == 3 or message.function_code ==
    4:
186             my_doc = {
187                 'Honeypot-IP': self._honeypot_ip,
188                 'Honeypot-Port': self._honeypot_port,
189                 'Client-IP': self._client_ip,
190                 'Client-Port': self._client_port,
191                 'Action': 'Client response sent',
192                 'timestamp': vienna.localize(datetime.now()),
193                 'Modbus Function Code': message.function_code,
194                 'Modbus Function Code description': self.
    possible_functions.get(message.function_code),
195                 'Value-Data sent': message.registers,
196                 'Modbus Packet sent': modbus_packet,
197             }
198         elif message.function_code == 5 or message.function_code ==
    6:
199             my_doc = {
200                 'Honeypot-IP': self._honeypot_ip,
201                 'Honeypot-Port': self._honeypot_port,
202                 'Client-IP': self._client_ip,
203                 'Client-Port': self._client_port,
204                 'Action': 'Client response sent',
205                 'timestamp': vienna.localize(datetime.now()),
206                 'Modbus Function Code': message.function_code,
207                 'Modbus Function Code description': self.
    possible_functions.get(message.function_code),
208                 'Value-Data sent': message.value,
209                 'Modbus Packet sent': modbus_packet,
210             }
211         # response to function code 15 and 16 do not send data back
    to client
212         # look at http://www.simplymodbus.ca/FC15.htm
213         elif message.function_code == 15 or message.function_code ==
    16:
214             my_doc = {
215                 'Honeypot-IP': self._honeypot_ip,
216                 'Honeypot-Port': self._honeypot_port,
217                 'Client-IP': self._client_ip,
218                 'Client-Port': self._client_port,
219                 'Action': 'Client response sent',
220                 'timestamp': vienna.localize(datetime.now()),
221                 'Modbus Function Code': message.function_code,
222                 'Modbus Function Code description': self.
    possible_functions.get(message.function_code),
223                 'Modbus Packet sent': modbus_packet,
```

```
224             }
225         else:
226             my_doc = {
227                 'Honeypot-IP': self._honeypot_ip,
228                 'Honeypot-Port': self._honeypot_port,
229                 'Client-IP': self._client_ip,
230                 'Client-Port': self._client_port,
231                 'Action': 'Client response sent',
232                 'timestamp': vienna.localize(datetime.now()),
233                 'Modbus Function Code': message.function_code,
234                 'Modbus Function Code description': 'unknown',
235                 'Modbus Packet sent': modbus_packet,
236             }
237         self.log_Event(my_doc)
238         ModbusTcpProtocol._send(self,message)
239
240
241
242 #Implement the ModpotTcpProtocol as a ServerFactory (see twisted
    documentation)
243 class ModpotServerFactory(ModbusServerFactory):
244
245     protocol = ModpotTcpProtocol
246
247     def __init__(self, elastic, e_addr, var_logger, store, framer=
    None, identity=None, **kwargs):
248         ModbusServerFactory.__init__(self, store, framer, identity,
    **kwargs)
249         if elastic:
250             self.protocol.elastic_address=e_addr
251         if var_logger:
252             self.protocol.var_logger = var_logger
```

Listing A.3: Sourcecode Modpot - protocols.py

## A.4. demoserver.py

```
#!/usr/bin/env python
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext,
   ModbusServerContext
import threading
import time
import random
import hashlib
import protocols

#This Class is the Honeypot; reads data from PLC and presents them to
    a attacker
class Demopot:
    def __init__(self, log, co_nr=10, di_nr=10, hr_nr=10, ir_nr=10,
   delay=10):
        self.EXCEPTION_OFFSET = 0x80  # this is the exception_offset,
    see pymodbus pdu.py - ExceptionResponse
        self.unit_id=0x00   # We only have one unit, so this does not
    matter
        self.delay = delay  # Delay for update-loop in seconds
        self.co_nr = co_nr  # Nulloffset - dh 0 heisst 1
        self.di_nr = di_nr  # Nulloffset - dh 0 heisst 1
        self.hr_nr = hr_nr  # Nulloffset - dh 0 heisst 1
        self.ir_nr = ir_nr  # Nulloffset - dh 0 heisst 1
        self.FC_CO = 1      # Function-Code for Coils
        self.FC_DI = 2      # Function-Code for Dicrete Inputs
        self.FC_HR = 3      # Function-Code for Holding Registers
        self.FC_IR = 4      # Function-Code for Input Registers
        self.__slave_store = ModbusSlaveContext(
            di = ModbusSequentialDataBlock(0, [0]*self.di_nr),
            co = ModbusSequentialDataBlock(0, [0]*self.co_nr),
            hr = ModbusSequentialDataBlock(0, [0]*self.hr_nr),
            ir = ModbusSequentialDataBlock(0, [0]*self.ir_nr))
        self.master_context = ModbusServerContext(slaves=self.
   __slave_store, single=True)
        self.slave_context = self.master_context[self.unit_id]
        self.log = log
        self.p = self.log.progress("DEMO MODE")
        self.vars = self.log.progress("Last request/response")

    def __del__(self):
        try:
            self.client.close()
            self.p.success("Done")
        except:
            self.p.failure("Done")

    def start_demomode(self, address, elastic=False, e_addr=None):
        self.__randomize_context(self.p)
        self.__start_server(elastic, self.p, e_addr, address)

    def __randomize_context(self, logger):
        random.seed()
        valuehash = hashlib.sha256()
        di_randoms = [random.randint(0,1) for p in range(0,self.di_nr
   )]
        co_randoms = [random.randint(0,1) for p in range(0,self.co_nr
   )]
```

```
51          hr_randoms = [random.randint(0,999) for p in range(0,self.
    hr_nr)]
52          ir_randoms = [random.randint(0,999) for p in range(0,self.
    ir_nr)]
53          self.slave_context.setValues(self.FC_CO, 0, co_randoms)
54          self.slave_context.setValues(self.FC_DI, 0, di_randoms)
55          self.slave_context.setValues(self.FC_HR, 0, hr_randoms)
56          self.slave_context.setValues(self.FC_IR, 0, ir_randoms)
57          valuehash.update(str(self.slave_context.getValues(self.FC_DI,
     0, count=self.di_nr)) +
58                       str(self.slave_context.getValues(self.FC_CO, 0, count
    =self.co_nr)) +
59                       str(self.slave_context.getValues(self.FC_HR, 0, count
    =self.hr_nr)) +
60                       str(self.slave_context.getValues(self.FC_IR, 0, count
    =self.ir_nr)))
61          logger.status("Values updated - SHA256-Hash: " + valuehash.
    hexdigest())
62
63      def __start_server(self, elastic, logger, e_addr=None, address=("
    0.0.0.0", 50002)):
64          try:
65              self.t = threading.Thread(target=self.__update_writer,
    args=[logger])
66              self.t.daemon = True
67              self.t.start()
68              startModpotTcpServer(self.master_context, elastic, self.
    vars, e_addr, address=address)
69              logger.status("TCP-Server started")
70          except:
71              logger.failure("Can\'t start TCP-Server")
72
73      #this function is executed as a thread
74      def __update_writer(self, logger):
75          while 1:
76              self.__randomize_context(logger)
77              time.sleep(self.delay)
78
79 #Helper Function to start the Modpot Class as a Server; see the
    twisted documentation
80 def startModpotTcpServer(context, elastic, var_logger, e_addr=None,
    identity=None, address=None, **kwargs):
81      from twisted.internet import reactor
82      from pymodbus.transaction import ModbusSocketFramer
83
84      address = address
85      framer = ModbusSocketFramer
86      factory = protocols.ModpotServerFactory(elastic, e_addr,
    var_logger, context, framer, identity, **kwargs)
87      reactor.listenTCP(address[1], factory, interface=address[0])
88      reactor_thread = threading.Thread(target=reactor.run, args=(False
    ,))
89      reactor_thread.daemon = True
90      reactor_thread.start()
```

Listing A.4: Sourcecode Modpot - demoserver.py

# Abbildungsverzeichnis

2.1. Mögliche Positionierung von Honeypots [3, p. 287] . . . . . 6
2.2. Buses and processors in industrial plants[7] . . . . . 8
2.3. Modbus request/response over TCP/IP . . . . . 10

3.1. Topologie des Honeypots . . . . . 12

4.1. Topologie des Proof-of-Concept . . . . . 21
4.2. Ausführung des Honeypots im Demomodus . . . . . 34

6.1. Analyse über Kibana - Netzwerkverkehr von modbusdetect . . . . . 39

# Tabellenverzeichnis

2.1. Vor- und Nachteile der Honeypot-Interaktionsklassen [3, p. 77] . . . . . 6
2.2. Überblick über die wichtigsten Function Codes . . . . . 10
2.3. Darstellung der MBAP-Header-Felder . . . . . 11
2.4. Darstellung der Modbus-Datentypen . . . . . 11

3.1. Technische Daten der Test-PLC . . . . . 17

# Listings


3.1. NMAP-Enumerierung einer Siemens S7-1200 . . . . . 18
3.2. Durchführung modbusdetect zu einer Siemens S7-1200 . . . . . 18
3.3. Durchführung modbus_findunitid zu einer Siemens S7-1200 . . . . . 19
3.4. Beispiel zur Verwendung von modbusclient zu einer Siemens S7-1200 . . . . . 20

4.1. Optionen des Kommandozeilenprogramms . . . . . 22
4.2. Beispiel eines Konfigfiles für den Honeypot . . . . . 22
4.3. Sanity-Check der Konfiguration . . . . . 23
4.4. Auszug Sanity-Check Logserver . . . . . 25
4.5. Auszug: Initialisierung des Modbus-Context . . . . . 25
4.6. Auszug: Verbindungsaufbau zu PLC . . . . . 26
4.7. Auszug: Feststellung Anzahl an Datenadressen . . . . . 27
4.8. Update-Funktion des Honeypots . . . . . 28
4.9. server.py: Start des Honeypots . . . . . 29
4.10. ModpotServerFactory . . . . . 30
4.11. ModpotTcpProtocol: connectionMade() und setupElastic() . . . . . 32
4.12. ModpotTcpProtocol Auzug: _execute und _send . . . . . 33

5.1. NMAP-Enumerierung einer Siemens S7-1200 . . . . . 35
5.2. NMAP-Enumerierung des Honeypots . . . . . 35
5.3. Durchführung modbusdetect zum Honeypot . . . . . 36
5.4. Durchführung modbus_findunitid zum Honeypot . . . . . 36
5.5. Durchführung READ_COILS zum Honeypot . . . . . 37
5.6. Durchführung READ_REGISTERS zum Honeypot . . . . . 37

A.1. Sourcecode Modpot - main.py . . . . . 40
A.2. Sourcecode Modpot - server.py . . . . . 45
A.3. Sourcecode Modpot - protocols.py . . . . . 49

A.4. Sourcecode Modpot - demoserver.py . . . . . 54

# Literaturverzeichnis


[1] “Richtlinie (eu) 2016/1148 des europÄischen parlaments und des rates über maßnahmen zur gewährleistung eines hohen gemeinsamen sicherheitsniveaus von netz- und informationssystemen in der union,” https://eur-lex.europa.eu/legal-content/DE/TXT/HTML/?uri=CELEX:32016L1148&from=DE, 07 2016.

[2] “Industroyer: Größte bedrohung für kritische infrastruktur seit stuxnet,” https://www.welivesecurity.com/deutsch/2017/06/12/industroyer-bedroht-kritische-infrastrukturen/, 06 2017.

[3] L. Spitzner, *Honeypots: Tracking Hackers*. Addison Wesley, 2002.

[4] R. Baumann and C. Plattner, “Honeypots,” Master’s thesis, Eidgenössische Technische Hochschule Zürich, 2002.

[5] F. Pouget, M. Dacier, and H. Debar, “Honeypot, honeynet, honeytoken: Terminological issues,” Institut Eurécom, Tech. Rep., 2003.

[6] “Honeynet definitions, requirements, and standards,” http://old.honeynet.org/alliance/requirements.html, 06 2018.

[7] T. Brandstetter, “Industrial security & safety; part 1: Industrial control system security & critical infrastructure protection,” 2018.

[8] “List of automation protocols,” https://en.wikipedia.org/wiki/List_of_automation_protocols, 07 2018.

[9] “Siemens industry online support,” https://support.industry.siemens.com/cs/document/8970169/which-ports-are-used-by-the-various-services-for-data-transfer-via-tcp-and-udp-and-what-should-you-watch-out-for-when-using-routers-and-firewalls-?dti=0&lc=en-WW, 05 2017.

[10] “Rfc 1006 - iso transport service on top of the tcp,” https://tools.ietf.org/html/rfc1006, 05 1987.

[11] *SIMATIC S7, S7-1200 Automatisierungssystem Systemhandbuch*, Siemens.

[12] “Iso/iec 8072 - information technology - open systems interconnection - transport service definition,” 08 1996.

[13] “Modbus application protocol specification v1.1b3,” http://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf.

[14] “Modbus messaging on tcp/ip implementation guide,” http://modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf.

[15] “Conpot project webpage,” http://conpot.org/, 10 2017.

[16] “Conpot documentation 0.3.1,” http://mushorg.github.io/conpot/index.html, 10 2017.

[17] “Owasp testing guide v3,” https://www.owasp.org/index.php/OWASP_Testing_Guide_v3_Table_of_Contents, 03 2018.

[18] “nmap dokumenation auf tools.kali.org,” https://tools.kali.org/information-gathering/nmap, 03 2018.

[19] “Metasploit webseite,” https://www.metasploit.com/, 08 2018.

[20] “Projekt pymodbus auf github,” https://github.com/riptideio/pymodbus, 10 2017.

[21] “Twisted documentation - writing servers,” http://twistedmatrix.com/documents/current/core/howto/servers.html, 02 2018.