



ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
School of Electrical and Computer Engineering

**ΜΕΛΕΤΗ ΤΩΝ ICS/SCADA HONEYPOTS ΚΑΙ ΣΥΓΧΡΟΝΩΝ
ΜΕΘΟΔΩΝ ΑΥΤΟΜΑΤΗΣ ΑΝΑΠΤΥΞΗΣ
STUDY OF ICS/SCADA HONEYPOTS AND MODERN
AUTOMATIC DEPLOYMENT METHODS**

Author

Irodotos Karatsoris

Supervisor

Associate Professor Sotirios Ioannidis

Thesis Committee

Associate Professor Sotirios Ioannidis

Professor Apostolos Dollas

Associate Professor Eftychios Koutroulis

Chania 2022

Table of contents

Acknowledgements	4
Abstract	5
Περίληψη	6
1 Introduction to Honeypots	7
1.1 Generic Honeypot Model	7
1.2 Classification of Honeypots	9
1.2.1 Based on the purpose	9
1.2.2 Based on the level of interaction	9
1.2.3 Based on the role of the honeypot	11
1.2.4 Based on hardware deployment	12
1.3 Honeypot Deployment Methods on a Network	12
2 Background on Industrial Control Systems	14
2.1 Simple Network Management Protocol (SNMP)	14
2.2 Modbus TCP	15
2.3 S7comm	16
2.4 Telnet	19
2.4.1 Negotiation bytes	20
2.4.2 User data bytes	21
2.5 IEC 60870-5-104	22
2.6 IEC 61850 (GOOSE - MMS)	24
2.7 DNP3	25
3 SCADA Networks & Honeypots	29
3.1 SCADA Honeynet Project	29
3.2 Digital Bond's Honeynet	29
3.3 Conpot	30
3.4 Crysyst PLC Honeypot (CryPLH)	30
3.5 SHaPe	32
3.6 S7commTrace	33
3.7 HoneyPLC	33
3.8 DiPot	35
4 Modern Automatic Deployment Methods	37
4.1 Historical Background	37
4.2 Introduction to Container Technology	39
4.3 Introduction to Kubernetes	41

	3
4.4 The Architecture of a Kubernetes Cluster	42
4.5 Deployment Process	43
4.6 Helm Charts	44
5 HoneyChart	45
5.1 Home Page	45
5.2 Custom Honeypots Page	46
5.2.1 How It Works	49
5.2.2 The Creation of the Helm Chart	54
5.3 Prebuild Interfaces Page	56
5.4 Deployment Example	59
5.5 Log Management and Visualization	61
5.6 System Architecture	64
6 Conclusion	65
References	66

Acknowledgements

There are a lot of people that I want to give thanks to because without them I would have never managed to deal with my problems and achieve my goals. First and foremost, my family, that stood strong by me even after illness, losses, and pain that we went through. My dear friends that were always there for me and made me feel joy even in my darkest hours.

Of course, I want to thank my supervisor Sotirios Ioannidis for giving me the opportunity to work on an interesting subject and work with a great team from the Foundation for Research and Technology - Hellas (FORTH). Manos Athanatos and Giorgos Tsirantonakis, thank you for your guidance and your support. And last but not least, Giorgos Kokolakis thank you for being a great project partner.

Abstract

When Industrial Control Systems (ICS) started to be developed, their security was not considered to be a priority, because they were isolated from other networks. Nowadays ICS networks are connected to other networks, even to the internet. Making sure these networks are protected from malicious attacks is an important issue. One of the ways to protect an ICS network is with the use of honeypots. This dissertation describes, at first, the known classifications of honeypots and the communication protocols that are used mostly in ICS networks. Based on the bibliography there are honeypots that can simulate these protocols in various combinations. These honeypots are described and categorized according to certain attributes. Concerning the modern deployment methods, after the reference of all known deployment methods, containerized application deployment is the most time and cost efficient method, according to the bibliography. Kubernetes is a container orchestration system for managing, scaling, and deploying software. And by combining Kubernetes with HoneyChart, we were able to achieve fast and automatic deployment of containerized honeypots.

Περίληψη

Όταν άρχισαν να αναπτύσσονται τα Συστήματα Βιομηχανικού Ελέγχου (ΣΒΕ), η ασφάλειά τους δεν θεωρούταν προτεραιότητα, επειδή ήταν απομονωμένα από άλλα δίκτυα. Σήμερα τα δίκτυα ΣΒΕ είναι συνδεδεμένα με άλλα δίκτυα, ακόμη και με το διαδίκτυο. Η διασφάλιση της προστασίας αυτών των δικτύων από κακόβουλες επιθέσεις είναι ένα σημαντικό ζήτημα. Ένας από τους τρόπους προστασίας ενός δικτύου ΣΒΕ είναι με τη χρήση honeypots. Η παρούσα εργασία περιγράφει, αρχικά, τις γνωστές κατηγορίες των honeypots και τα πρωτόκολλα επικοινωνίας που χρησιμοποιούνται κυρίως σε δίκτυα ΣΒΕ. Με βάση τη βιβλιογραφία υπάρχουν honeypots που μπορούν να προσομοιώσουν αυτά τα πρωτόκολλα σε διάφορους συνδυασμούς. Αυτά τα honeypots περιγράφονται και κατηγοριοποιούνται σύμφωνα με ορισμένα χαρακτηριστικά. Όσον αφορά τις σύγχρονες μεθόδους ανάπτυξης, μετά την αναφορά όλων των γνωστών μεθόδων ανάπτυξης, η ανάπτυξη εφαρμογών σε μορφή container είναι η πιο αποδοτική μέθοδος από άποψη χρόνου και κόστους, σύμφωνα με τη βιβλιογραφία. Το λογισμικό Kubernetes είναι ένα σύστημα ενορχήστρωσης container για διαχείριση, κλιμάκωση και ανάπτυξη λογισμικού. Και συνδυάζοντας το Kubernetes με το HoneyChart, μπορέσαμε να επιτύχουμε γρήγορη και αυτόματη ανάπτυξη containerized honeypots.

1 Introduction to Honeypots

A useful component of an Intrusion Detection System (IDS) is the honeypot. Honeypot is a decoy system that attracts attackers away from critical systems. This system emulates an operating system (OS) and/or services and acts as a trap for hackers. This means that users who do not belong in the network have no real reason to interact with it. Any interaction with a honeypot is considered suspicious and gets logged for future analysis by the administrators (Stallings et al. 2012). These interactions can be access attempts, keystrokes, file access and modification, and process execution.

There are many configurations that can be deployed, depending on what is required. Honeypots can attain intrusion detection, countering spam, data collection, etc. Also, they can keep attackers interested in the system long enough for the administrators to understand the vulnerabilities of their system. With this information, they can patch the vulnerabilities themselves or inform the software or hardware provider about it. However, honeypots by themselves do not provide complete protection from attackers. And certainly, they do not replace other traditional security systems like intrusion detection systems. But if they are to be combined with other defending components, they can produce good results. For example, they can detect insider threats and attackers which have breached the defenses.

1.1 Generic Honeypot Model

On a network, a honeypot is set up solely to be attacked. It's built with intentional flaws, and it's exposed to a public network. A honeypot does not have a production value. Therefore, any traffic destined to a honeypot is considered suspicious and gets monitored. For this to be achieved, a honeypot consists of (Joshi & Sardana, 2011):

- Honeypot Production System: It is a fake production system, created to attract intruders. Depending on the emulated service, it can provide honey-files and fake system resources to the intruder. This system produces automatic responses for every interaction so that it appears to be functioning as a real production system.
- Firewall: Firewalls log how an intruder tries to break into a honeypot. The honeypot's firewall is configured to log all packets going to and from it.

- **Monitor Unit:** This unit evaluates threats by monitoring network and/or system activities for malicious activities or policy violations and creates reports for the administrators. These reports help identify the intruders' methods, methodology and intention, by providing information about the order, sequence, timestamps and type of packets used by an intruder to gain access to the honeypot and also the keystrokes, system accesses, files changed, etc.
- **Alert Unit:** This unit by creating alerts notifies the administration about traffic going to or from the honeypot.
- **Logging Unit:** This unit stores both firewall and system logs, as well as the traffic between the firewall and the honeypot system, efficiently.

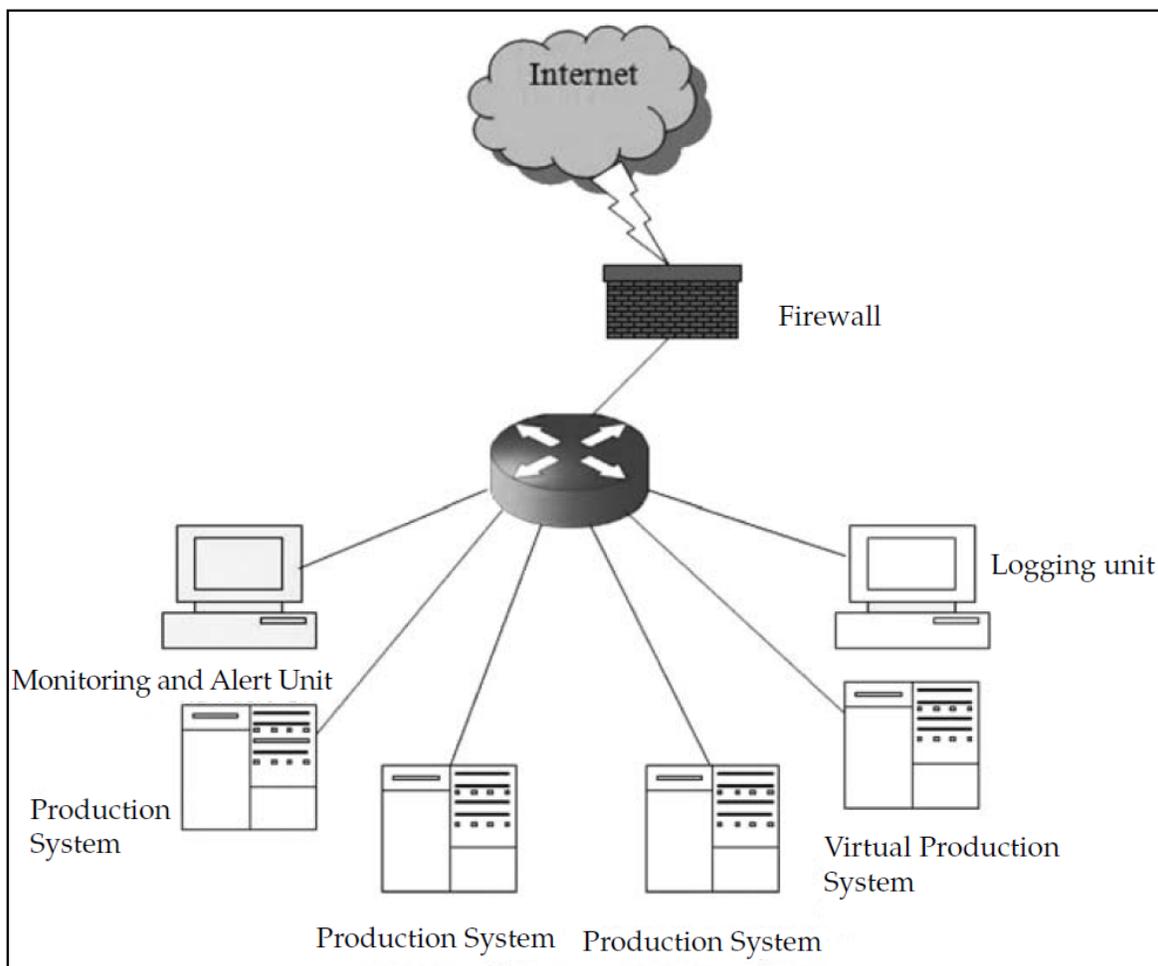


Figure 1.1 Generic Model of Honeypot (Joshi & Sardana, 2011, 9)

1.2 Classification of Honeypots

There are different types of honeypots, these types can be grouped into four categories.

1.2.1 Based on the purpose

Honeypots can be classified into Production and Research honeypots according to their purpose.

Production Honeypot

They are used by organizations as an additional countermeasure for possible attacks (Aliyev, 2010). Production honeypot's purpose is to emulate the production network of a company. Depending on the level of interaction, the company can collect information about the attacks that take place in their network. By analyzing this information they can discover vulnerabilities in their systems and be ready to face similar attacks. They add value to a company's security measures. But they are unable to prevent hackers from entering. To keep malicious actors out of the company's IT infrastructure, the organization must continue to rely on its security policies, processes and best practices, such as disabling unused services, patch management and implementing security measures such as firewalls, intrusion detection systems, anti-virus and reliable authentication mechanisms. (Joshi & Sardana, 2011)

Research Honeypot

They are deployed mainly by non-profit research organizations or educational institutes. Research honeypots are used for a more in-depth analysis of threats. By collecting information on the attacker's motive, methods and tools, researchers are able to design newer and better systems for intrusion detection and malware protection. They can be compared with a "counter-intelligence" body. In addition, research honeypots are great for capturing automated attacks like auto-rooters and worms. Since these attacks target entire network blocks, research honeypots can easily catch and analyze these attacks.

1.2.2 Based on the level of interaction

According to their services and the interaction level that a honeypot provides, it can be classified as a Low, High, or Medium Interaction Honeypot.

Low-interaction Honeypots

Low-interaction honeypots emulate IT services that are usually requested by attackers. They do not provide an actual OS, only a limited subset of the functionality attackers would expect from a server (Peter & Schiller, 2011). Usually, this kind of honeypot monitors certain ports that a server would use for known services, like FTP, SQL, HTTP, etc. Because of the emulation of a limited amount of services and the lack of an OS, low-interaction honeypots are easy to install, deploy and maintain. They log only a limited amount of information regarding the hacker's activities.

High-interaction Honeypots

A high-interaction honeypot is a complete system, which contains a fully functional OS and all the services that it could provide (Stallings et al., 2012). High-interaction honeypots are used to capture the maximum amount of information concerning new and old ways of attacking. Researchers are using this kind of honeypot to investigate the attacker's methodology, by observing their interactions with the system. High-interaction honeypots are complex systems, which makes their installation, deployment and maintenance difficult. Also, they increase the risk of intrusion into the real system.

Medium-interaction Honeypots

They attempt to mix the benefits of both low and high-interaction honeypots. They are more advanced than low-interaction honeypots, but they are not as advanced as high-interaction honeypots. When they are questioned, they will react in a specific way. These honeypots don't have a real OS or follow every aspect of the application protocol. They have a virtualization layer in place. They simply respond to the hacker's requests. These responses are designed to entice hackers into sending their payload. As security personnel receive the payload, they remove the shellcode for further inspection and forensic examination. These honeypots are then used to mimic the behavior that shellcode was intended to perform after a thorough review. Following these steps to download malware from a serving site, it is either saved locally or sent somewhere else for review. These honeypots are very complex, time-consuming to install and necessitate a significant amount of effort as well as a detailed understanding of protocols, application services and protection to build. They are also more vulnerable to dangers.

Medium-interaction honeypots are custom-made honeypots that have been customized to meet the needs of organizations or individuals. (Joshi & Sardana, 2011)

Factors	Low-interaction	Medium-interaction	High-interaction
Degree of involvement	Low	Medium	High
Real operating system	No	No	Yes
Installation	Easy	Difficult	More difficult
Maintenance	Easy	Easy	Time-consuming
Risk	Low	Medium	High
Compromised wished	No	No	Yes
Need control	No	No	Yes
Knowledge to run	Low	Low	High
Knowledge to develop	Low	High	Mid-high
Data gathering	Limited	Medium	Extensive
Interaction	Emulated services	Requests	Full control

Table 1.1 Low-interaction vs. Medium-interaction vs. High-interaction (Joshi & Sardana, 2011, 17)

1.2.3 Based on the role of the honeypot

Additionally, honeypots can be classified as server or client honeypots. This classification is based on the direction of the interaction.

Server Honeypots

Server honeypots are designed to be passive and do not initiate traffic unless they are compromised. Server-side honeypots aid in the detection of new exploits, the collection of malware and the enrichment of threat analysis research. (Joshi & Sardana, 2011)

Client Honeypots

Unlike Server Honeypots, Client Honeypots actively search for threats and possible malicious entities and they initiate the interaction. Usually, they emulate web browsers or components of theirs. When a client interacts with malicious servers, client-side attacks target vulnerable client applications. These honeypots are used to search for and detect malicious servers. (Joshi & Sardana, 2011)

1.2.4 Based on hardware deployment

Finally, honeypots can be classified as Physical or Virtual depending on their physicality.

Physical Honeypots

Physical Honeypots are real entities on the network with their own IP address. They are often high-interaction honeypots and they are expensive to install and maintain. Due to the limited view of their single IP address and the high cost of maintaining a farm of physical honeypots, they are less practical in real-world scenarios. (Joshi & Sardana, 2011)

Virtual Honeypots

They are usually implemented on a single physical machine that serves as a host for multiple virtual honeypots. Virtual honeypots are more cost-effective when it comes to monitoring large IP address spaces while also simulating large IP addresses. (Joshi & Sardana, 2011)

1.3 Honeypot Deployment Methods on a Network

Honeypots can be deployed in a variety of locations on a network, depending on the goal of the organization using it. There are three main locations: Outside the external firewall, Inside the DMZ (Demilitarized Zone) and Inside the internal network.

A honeypot located outside the external firewall detects connection attempts on unused IP addresses. It does not increase the risk for the internal network. It also reduces the number of alerts that are produced by the external firewall, and by internal IDS sensors. However, it cannot trap internal attackers.

DMZ allows an organization to access untrustworthy networks while ensuring its private network remains secure. On the DMZ are stored external-facing services and resources like FTP,

mail server, web server, etc. A honeypot deployed in the DMZ can emulate the servers that connect to the public domain and provide early warning of threats located there. This means that certain ports will intentionally be opened so that the honeypot can attract attackers.

The most advantageous way to detect attacks in the internal network is the internal honeypot. It alerts if any external exploits have made it past other network defenses and catches internal threats at the same time. (Stallings et al., 2012)

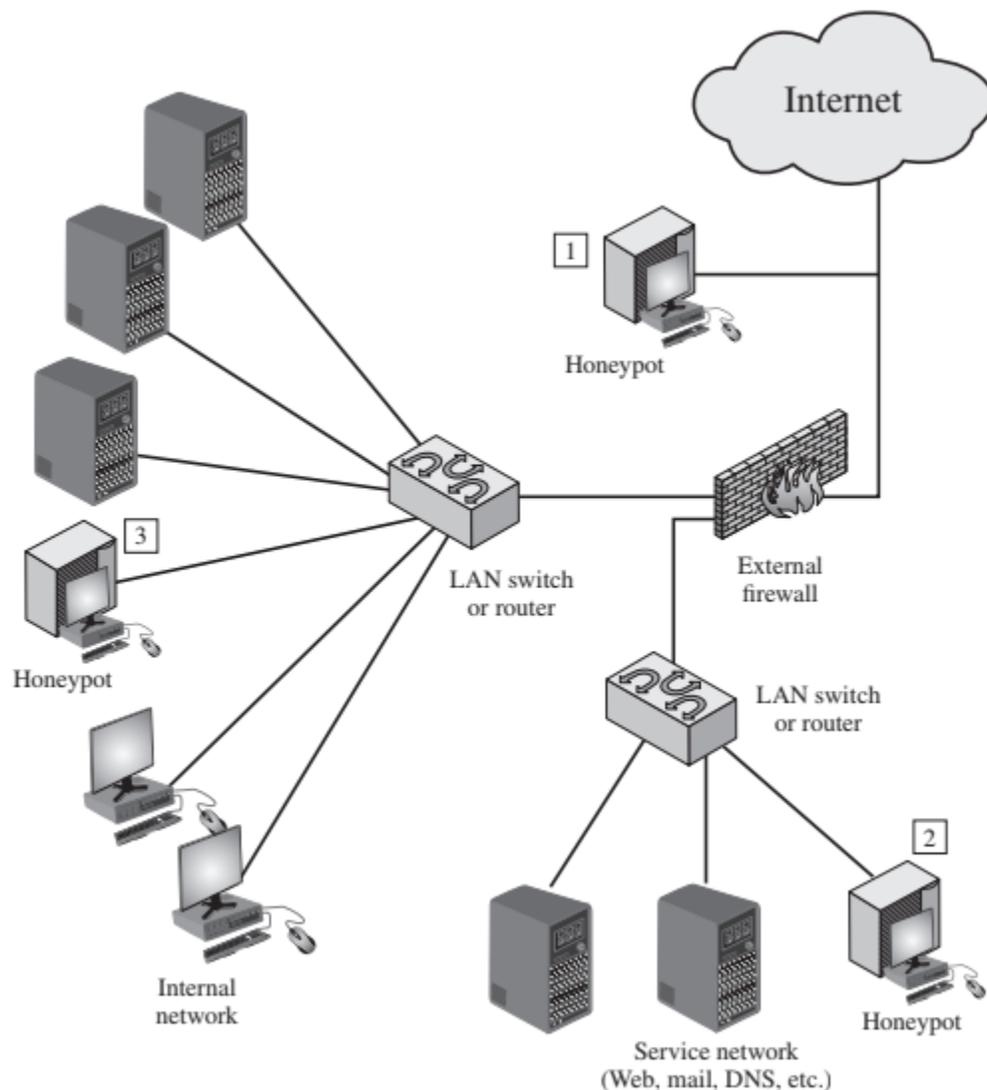


Figure 1.2 Example of Honeypot Deployment (Stallings et al., 2018, 301)

2 Background on Industrial Control Systems

Industrial Control Systems (ICS) are the systems that manage an industrial process. These systems usually include Supervisory Control and Data Acquisition (SCADA), Distributed Control Systems (DCS) and Programmable Logic Controllers (PLC). The above systems are used by industries to monitor and control energy production, water and electricity distribution and manufacturing. They could also be a significant part of a country's Critical National Infrastructure (CNI).

Specifically, SCADA systems are distributed systems that are used to control assets on a large geographical area, usually thousands of square kilometers, where centralized data acquisition and control are critical to system operation. Data is collected from field devices that control local operations, like valve controls, sensors and monitoring systems.

DCS are used to control industrial processes like electric power generation, oil refineries, water and wastewater treatment, as well as chemical, food and automotive manufacturing. DCS are part of a control architecture that includes a supervisory level of control that oversees various, integrated sub-systems that control the specifics of a localized process.

The PLC is a small industrial computer that was initially intended to perform the logic functions performed by electrical hardware (relays, switches and mechanical timer/counters). PLCs have developed into controllers capable of managing complex processes and they are widely used in SCADA and DCS systems.

Much like Web Applications that use protocols like TCP/IP, HTTP/HTTPS, FTP, etc. to transfer information, ICSs also use communication protocols. The most used protocols are described below.

2.1 Simple Network Management Protocol (SNMP)

For monitoring the health of networked devices, the Simple Network Management Protocol (SNMP) is used. Administrators can access a wide range of information from devices, including interface statistics, software versions and internal component temperatures. As a form of automated administration, SNMP also allows you to remotely write certain values (Lewis & Peterson). Because it provides a simple and easy way to monitor and manage the device, the SNMP service is commonly installed on intermediate and end network devices such as PLCs. A

typical SNMP conversation involves two parties: a manager who queries the requests and a managed device who responds to them (Buza et al., 2014).

2.2 Modbus TCP

Modicon invented the Modbus protocol in 1979. Modbus is a basic communication protocol that is widely used in the industry. It is a protocol that is universal, open and simple to use. Although new industrial products like PLCs, PACs, I/O devices and instruments may have Ethernet, serial, or even wireless interfaces, Modbus remains the preferred protocol. Modbus protocol's main advantage is that it works with any type of communication medium, including twisted pair wires, wireless, fiber optics, Ethernet, and so on. The plant data is stored in the memory of the Modbus devices. Discrete input, discrete coil, input register and holding register are the four parts of this memory. The discrete input and coil are of 1 bit while the input register and holding register are of 16 bits. Modbus TCP is a server-client communication protocol that runs over an Ethernet TCP/IP network. The Modbus TCP messaging cycle is divided into four steps, as shown in Figure 2.1. The client sends a query (connection request) to the server in the first step, the server acknowledges or accepts the query in the second step, the server sends responses for function code in the third step, and the client sends a confirmation signal to the server in the fourth step, which may be a disconnected TCP connection. Figure 2.2 depicts the Modbus TCP message format.

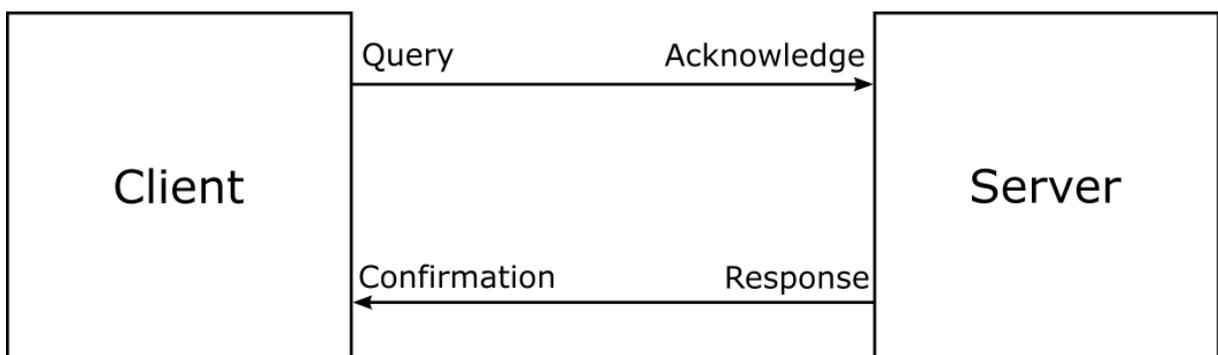


Figure 2.1 Message Cycle in Modbus TCP (Tamboli et al., 2015, 3)



Figure 2.2 Message Format of Modbus TCP (Tamboli et al., 2015, 3)

MBAP (Modbus application header) is a seven-byte message format that includes transaction ID, protocol ID, message length and client ID. In Modbus TCP, the server's ID and port number are required to establish communication, while the message format for a client requires the server's IP address, client ID and port number (Tamboli et al., 2015). Modbus is the most popular communication protocol for SCADA applications, because of its simplicity and ease of use. However, Modbus has several well-known vulnerabilities, such as lack of encryption (Samanis, 2018).

2.3 S7comm

The S7 communications protocol (S7comm) is a Siemens S7-300/400 PLC family proprietary protocol. The Siemens S7-1200/1500 PLC family also partially supports the protocol. PLC programming, data exchange with PLCs, PLC data access by SCADA systems and diagnostics are all possible with it.

S7comm's Ethernet implementation is based on ISO TCP (RFC 1006), a block-oriented protocol. Each block is called a protocol data unit (PDU). Each transmission in the S7comm protocol is function-oriented or command-oriented, meaning it contains a command or a response to a command. If a command or reply does not fit inside a single PDU, it is split across multiple PDUs.

Each command has four components:

- Header
- Parameters
- Parameter data
- Data block

The header and parameters are required components of an S7comm command; the remaining components are optional. The first byte in the parameters field represents the S7 function code. S7's optional function codes are listed in Table 2.1. To create an S7 connection, the

Communication Setup code is used. Read code assists the host computer in reading data from the PLC, while Write code assists the host computer in writing data to the PLC. The Request Download, Download Block, Download End, Download Start, Upload, and Upload End codes are used to perform block downloading and uploading operations. The operations of Hot Run and Cool Run are covered by PLC Control code, while PLC Stop is used to turn off the device. The 0x00 function code denotes a system function that is used to verify system settings or status. The 4-bit function group code and 1-byte subfunction code in the parameters field describe the details. Table 2.2 shows how system functions are further divided into seven groups. The Block function reads the block, while the Time function checks or sets the device clock (Xiao et al., 2017).

Code	Function	Code	Function	Code	Function
0x00	System functions	0x1b	Download block	0x1f	Upload end
0x04	Read	0x1c	Download end	0x28	PLC control
0x05	Write	0x1d	Download start	0x29	PLC stop
0x1a	Request download	0x1e	Upload	0xf0	Communication setup

Table 2.1 System function codes of S7comm (Xiao et al., 2017, 415)

Function group code	Function	Subfunction code	Subfunction
1	Programmer commands	1	Request diag data
		2	VarTab
2	Cyclic data	1	Memory
3	Block function	1	List blocks
		2	List blocks of type
		3	Get block info
4	CPU function	1	Read SZL

		2	Message service
5	Security	1	PLC password
6	PBC BSEND/BRECV	None	None
7	Time function	1	Read clock
		2,3	Set clock
		4	Read clock (following)

Table 2.2 System function group and corresponding subfunction (Xiao et al., 2017, 416)

The protocol encapsulation structure is shown in Figure 2.3, followed by S7 Telegram, ISO on TCP and TCP/IP (Yau et al., 2018).

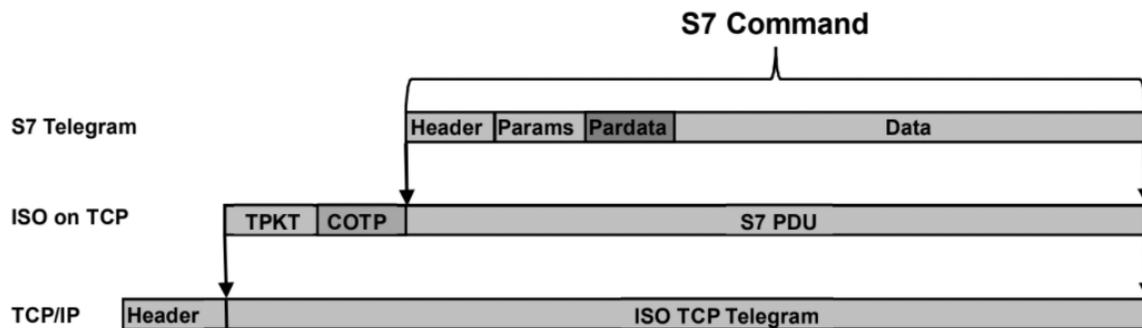


Figure 2.3 Protocol Encapsulation (Yau et al., 2018, 339)

Lastly, The S7 protocol's communication procedure is divided into three stages, as shown in Figure 2.4. The first stage is to establish a COTP connection, the second is to set up S7 communication, and the third is to exchange the request and response for the function code.

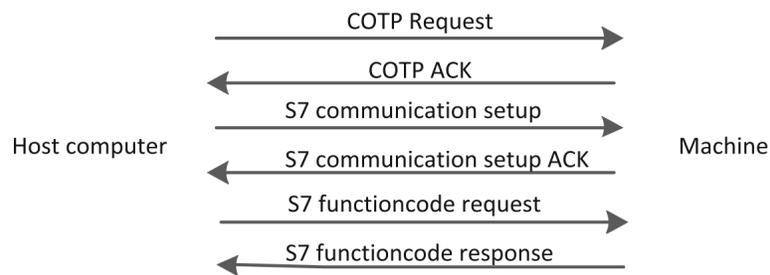


Figure 2.4 Communication procedure of S7 protocol (Xiao et al., 2017, 415)

2.4 Telnet

Telnet is a TCP-based text-oriented interaction protocol that allows bidirectional communication with a device. It's most commonly used to send commands to a terminal server. Although it is no longer used on servers and workstations, it is still widely used on embedded systems due to its simplicity. After authentication, access to the command line is granted; however, owners of IoT devices frequently do not change the default credentials (Metongnon & Sadre, 2018).

The Mirai botnet is one of the most well-known botnets on the Internet. Mirai was first discovered attacking Telnet ports 23/TCP and 2323/TCP in August 2016. Mirai used a list of 68 default usernames and passwords to gain access to the targeted device. In September 2016, the Mirai botnet was also responsible for a massive DDoS attack against KrebsOnSecurity's servers. Because the Telnet protocol does not encrypt the payload, all of the communication between devices can be read from network traffic (Musilová, 2020). The Telnet protocol's bytes can be divided into two groups:

- **Negotiation bytes:** Dedicated non-printable bytes used to configure and exchange information between the two terminals. The majority of the negotiation bytes are sent at the start of the communication, but they can be sent at any point during the communication.
- **User data:** Any byte in the Telnet communication that isn't the negotiation byte. These bytes hold everything the user typed on the keyboard as well as all the data displayed in the terminal.

2.4.1 Negotiation bytes

Specifically, there are three bytes in the negotiation process. The first byte is always 0xFF, which indicates the beginning of the negotiation byte. 0xFF is called Interpret As Command (IAC) byte. The second byte is the option code, which indicates whether you want to start or stop performing a particular option. The option byte is the third byte, and it specifies a terminal function or setting. Four different bytes can be used to turn on or off an option during the negotiation process:

- 0xFE: Demand to stop (DON'T)
- 0xFD: Request to start using specified option (DO)
- 0xFC: Reject the proposed option (WON'T)
- 0xFB: Accept the proposed option (WILL)

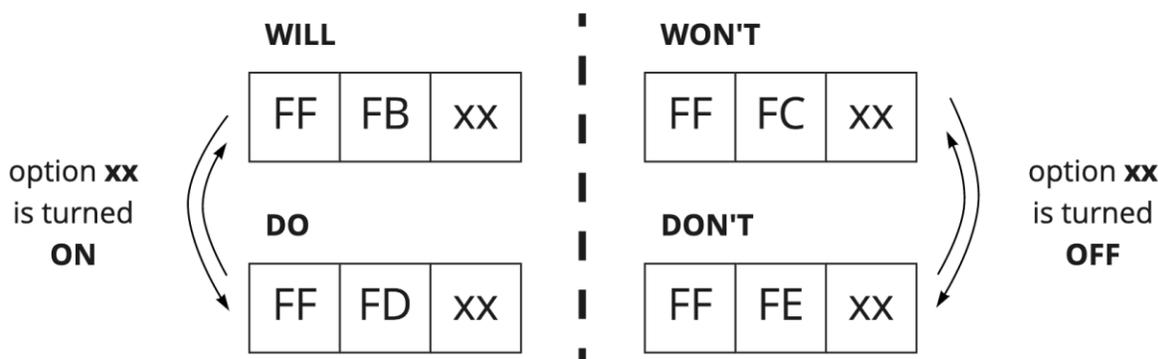


Figure 2.5 The structure of the Telnet negotiation bytes (Musilová, 2020, 12)

Only if both devices agree to use the specified option is it enabled. By sending the 0xFD byte, also known as the DO byte, to the other device, any device can propose to use any option. If the other device wishes to use this option, it responds with the 0xFB byte, also known as the WILL byte, indicating that it accepts the proposal. If the second device wants to reject the proposed option, it sends the WON'T byte, meaning the 0xFC byte. By sending the 0xFE byte, also known as the DON'T byte, to the client, any device can demand this option to turn off any previously agreed upon option. The other device must confirm this by sending the 0xFC byte, also known as the WON'T byte, in order to disable the option. The structure of the negotiation byte is shown in Figure 2.5 (Musilová, 2020).

The subnegotiation process can begin once the specified option is enabled. The terminals exchange detailed information or settings about each other during this process. The subnegotiation process always begins with the 0xFF byte, which indicates the start of the subnegotiation bytes, followed by the 0xFA byte. The specified option that was previously turned on is the next byte, followed by any number of bytes. The meaning of those bytes is determined by the option selected. The 0xFF byte is always followed by the 0xF0 byte in the subnegotiation bytes. The subnegotiation bytes structure is shown in Figure 2.6 (Musilová, 2020).

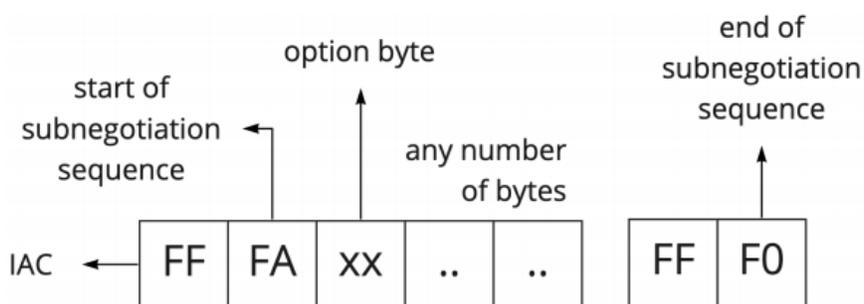


Figure 2.6 The Structure of the Telnet Subnegotiation Bytes (Musilová, 2020, 12)

2.4.2 User data bytes

The user data bytes are all the bytes that do not belong to the negotiation or subnegotiation byte structures, and they represent the interaction between two hosts. We can find everything the client-side user typed on the keyboard, edited, and sent to the server, as well as all the server's responses, among those bytes in the server-client oriented connection. The user on the client-side interacts with the server by typing commands in the command line in server-client-oriented communication. Every key pressed by the user is sent to the server to be processed. The majority of the sent bytes are returned to the client, and they are only displayed on the screen of the user's terminal window after they have been delivered to the client, as shown in Figure 2.7. The echo bytes are all of the returned bytes. This is why, when a user is connected to a server located far away from them, there may be a delay between pressing a key and seeing the character appear on the terminal screen. There are some exceptions when the server does not send back the echo byte, such as when typing a password or when sending special bytes (Musilová, 2020).

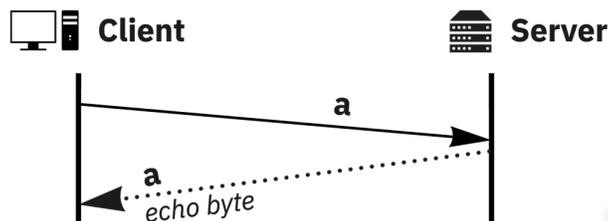


Figure 2.7 Telnet Echo Bytes

Many PLCs are still managed by Telnet, despite the security issues with cleartext password transmission are well known. In addition to the cleartext password issue, large character strings passed to the PLCs username or password fields can cause the PLC to freeze or reload (Lewis & Peterson).

2.5 IEC 60870-5-104

IEC 60870-5-104 provides network access for IEC60870-5-101 based on TCP/IP, which can be used for basic remote-controlled tasks between control centers and substations. The IEC 60870-5-104 protocol, on the other hand, sends messages in plain text with no authentication mechanism. In addition, the IEC 60870-5-104 protocol is based on TCP/IP, which has its own set of cyber-security issues. As a result of the IEC/104 protocol, a proliferation of cyber vulnerabilities in SCADA systems has emerged.

In Europe, China and many other non-US countries, the IEC/104 protocol is widely used in SCADA systems. The Enhanced Performance Architecture (EPA) model, which belongs to the application layer protocol, is given a transport and a network layer by IEC/104. A port number is assigned to a TCP/IP-based application layer protocol. The IEC/104's standard port number is 2404, which can be used to write detection rules.

As shown in Figure 2.8, the IEC/104 application layer sends an Application Service Data Unit (ASDU). The IEC/104 protocol defines Application Protocol Control Information (APCI) to detect the start and end of the ASDUs because the transport interface does not define a start or stop mechanism for the ASDUs of IEC 60870-5-101. The start character (68H), the Application Protocol Data Units (APDUs) length field and the control field make up the APCI. Figure 2.9 shows how the APCI and ASDU combine to form the APDU. The APDU has a maximum length

of 253 bytes, and the control field has a length of 4 bytes. The three types of control field formats are I format, S format and U format, which are used to perform numbered information transfer, numbered supervisory functions and unnumbered control functions, respectively (Yang et al., 2013).

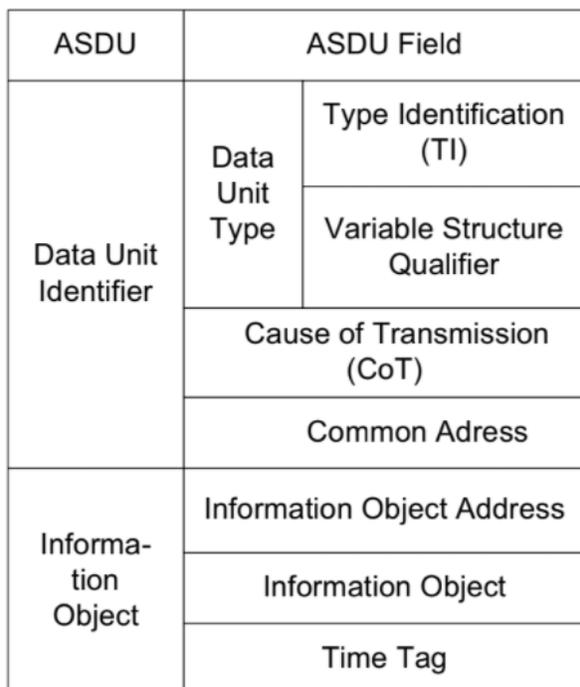


Figure 2.8 ASDU Structure

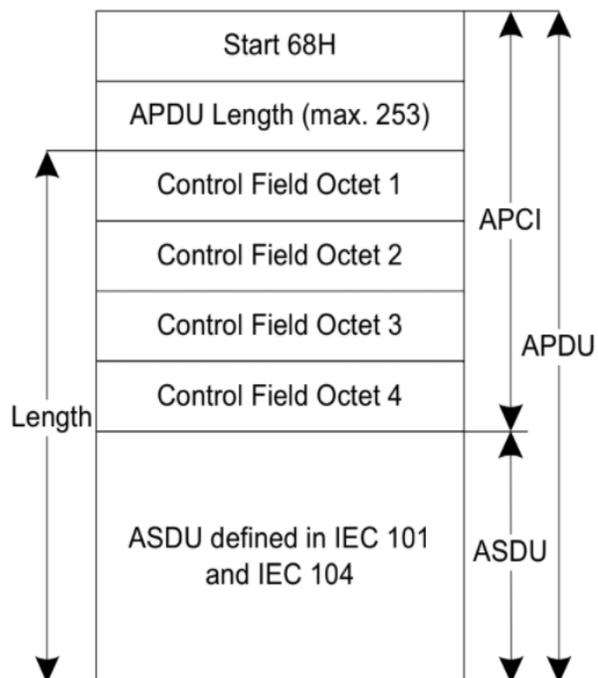


Figure 2.9 APDU Structure

Potential cyber vulnerabilities and attacks from the physical layer to the application layer in the IEC/104 protocol are as follows:

- Plaintext Mode Message Transmission:** Information transmission between the control center and substations is potentially vulnerable to eavesdropping, sniffing and tampering as a result of data transmission in clear text in legacy SCADA systems. In each case, they could be modified and then re-injected onto the communications infrastructure to jeopardize the SCADA system's stability or security, possibly to facilitate further intrusion at a later date.
- Lack of Authentication Mechanism:** Malicious attackers could gain unauthorized access to SCADA systems, compromise information integrity, and availability, and launch spoofing, replay, and MITM attacks due to a lack of authentication for

interrogation commands, remote control commands and remote adjustment commands. This is a critical vulnerability because the lack of authentication allows for relatively easy access to vulnerable points, potentially resulting in catastrophic damage and compromised power system operation and safety.

2.6 IEC 61850 (GOOSE - MMS)

The IEC 61850 standard is frequently used in electric substation automation systems. Generic Object Oriented Substation Events (GOOSE) and Manufacturing Message Specification (MMS) are the two main protocols used by an IEC 61850 substation. The GOOSE protocol is an Ethernet-based protocol. It uses the Generic Substation Event (GSE) service, which is part of the IEC61850-7-2 communications standard, to provide fast and reliable multicast messaging. The GOOSE protocol, which is based on Ethernet, does not include IP addresses.

When broadcast messages are sent from logical devices, the GOOSE protocol has a static periodic transmission behavior. When no new events are triggered, the retransmission time between GOOSE messages from a single logical device is usually set. When a new event occurs, however, a new message is created and sent to the subscribed destination devices right away. The subsequent transmission time increases exponentially until it reaches the usual rate again, restoring the regular periodic retransmission status. When a new event occurs, a new message is generated with the following actions: 1) GOOSE stNum value is incremented by one; 2) GOOSE sqNum value is reset to zero; 3) GOOSE time value record is set to the current time; and 4) GOOSE datSet value is modified.

Manufacturing Message Specification (MMS), an IP-based protocol, is the second communications protocol commonly used in substation automation. To support peer-to-peer communications in substation automation, MMS messages override the TCP/IP protocol; however, IP addresses are contained in the headers of MMS packets. There are two message transmission modes in MMS. Request/response communication between a client and server is the first mode. The second mode is unsolicited communications, in which the server sends periodic status updates or new event reports (Lahza et al., 2018).

Figure 2.10 depicts the IEC 61850 architecture and the range of uses of MMS and GOOSE protocols.

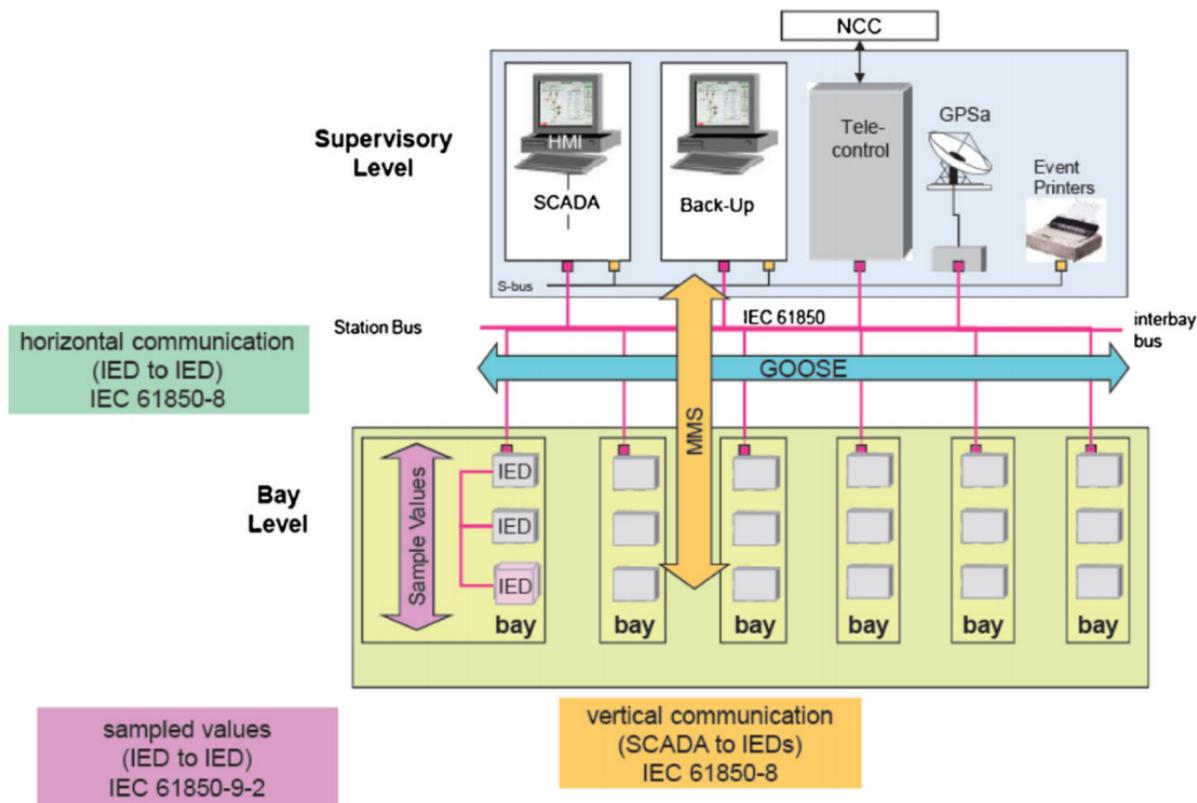


Figure 2.10 IEC 61850 substation automation architecture (Yoo & Shon, 2015, 306)

2.7 DNP3

DNP3 was created in the early 1990s by Westronic, Inc. (now GE Harris). The protocol specifies how SCADA devices exchange control commands and process data.

Between a control center (master unit) and outstation devices, DNP3 supports three simple communication modes.

- Unicast transaction

A master sends a request message to an addressed outstation device, which responds with a reply message in a unicast transaction.

- Broadcast transaction

The master sends a message to all outstations in the network in a broadcast transaction, but the outstation devices do not respond to the broadcast message.

- Unsolicited responses

Unsolicited responses from outstation devices are the third mode of communication; these responses are typically used to provide periodic updates or alerts.

The DNP3 protocol is compatible with a wide range of network configurations. Figure 2.11 depicts three common configurations.

- One-on-One

One master and one outstation device share a dedicated connection, such as a dial-up telephone line, in a "one-on-one" configuration.

- Multi-Drop

One master communicates with multiple outstations in the popular "multi-drop" configuration. The master sends all requests to each outstation, but each outstation only responds to messages addressed to it.

- Hierarchical

A device that serves as an outstation in one segment and a master in another is referred to as a "sub-master" in a "hierarchical" configuration.

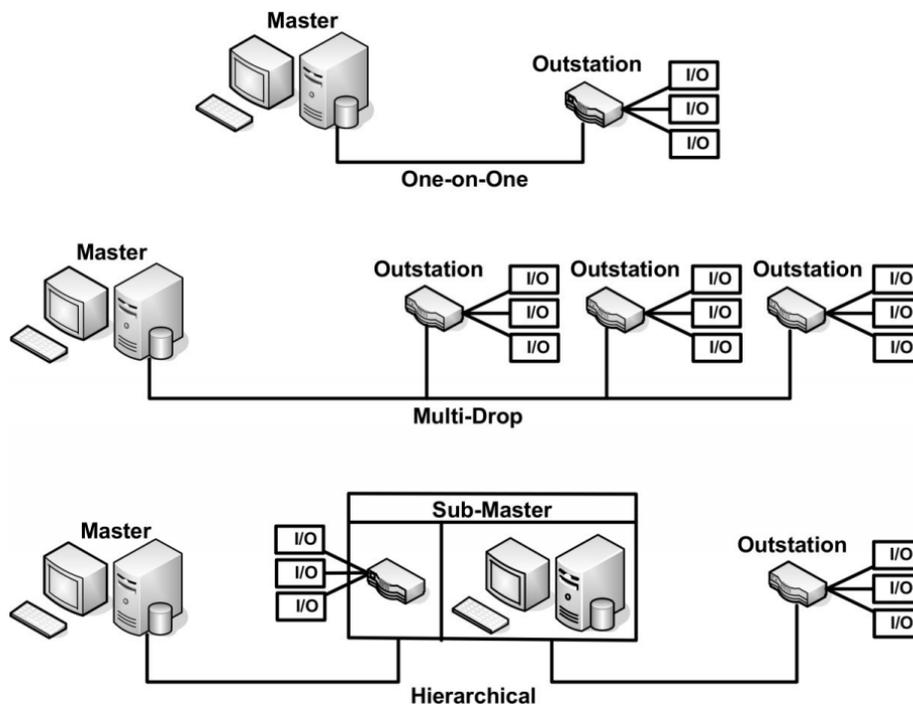


Figure 2.11 DNP3 Network Configurations (East et al., 2009, 69)

Early SCADA architectures relied heavily on noisy and distorted communication circuits. As a result, DNP3 was created with multiple protocol layers in mind. Based on the Open Systems Interconnection (OSI) model, the International Electrotechnical Commission (IEC) proposed the IEC 870 standard for telemetry data transmission in SCADA systems. The three-layer Enhanced Performance Architecture (EPA) was created by removing unnecessary layers from the seven-layer OSI model (from the perspective of SCADA systems) (Figure 2.12). EPA, on the other hand, did not support application layer messages that were longer than a data link frame's maximum length. DNP3 addressed this problem by including a pseudo-transport layer that allowed for message fragmentation.

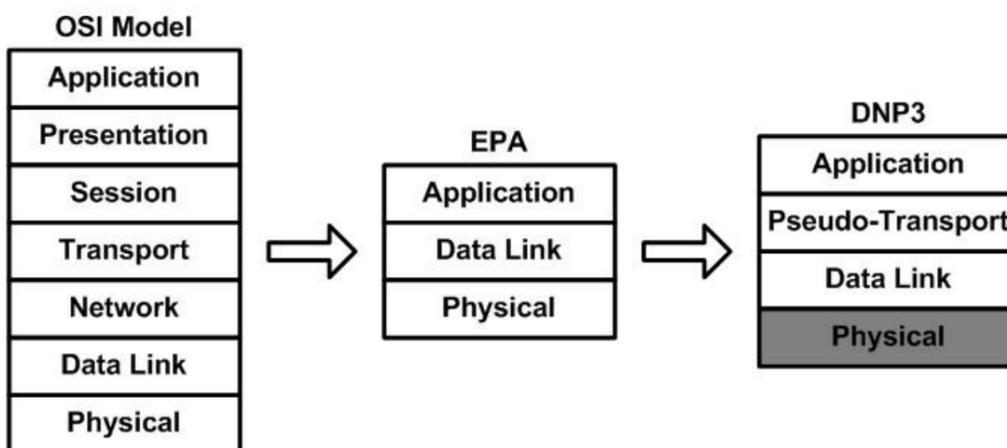


Figure 2.12 Design Progression From OSI to DNP3 (East et al., 2009, 70)

The DNP3 protocol layers sit on top of a physical layer that handles message transmission over physical media like radio, satellite, copper and fiber. The electrical settings, voltage and timing, as well as other properties required to send signals between devices, are determined by the physical layer specification. Send data, receive data, connect, disconnect and status update are the five services provided by the physical layer. Because the physical layer is not specified in the DNP3 standard, it is shaded in Figure 2.12.

DNP3 can be transmitted using a variety of physical media, including serial links. Modern SCADA systems, on the other hand, typically use DNP3 in IP networks. In IP-based implementations, the DNP Users Group has mandated that the three layers of DNP3 remain

unchanged. As a result, the three DNP3 layers in the protocol stack are placed directly above the TCP/IP or UDP/IP layers (East et al., 2009).

Situational awareness and forensics are difficult to implement, because of the operational requirements and reliance on embedded systems. Organizations that use ICS or Operational Technology (OT) have high operational demands. That means system security and data integrity are required for reliable operation.

3 SCADA Networks & Honeypots

When the SCADA systems started to be developed security was not a concern, mainly because the network was isolated from all other networks. As the industry grows, the demand for more connectivity also increases. This resulted in SCADA systems connecting to other networks. This makes them vulnerable to malicious attacks.

Introduction

Honeypots can be used on SCADA networks to identify attackers that threaten the integrity of the system and analyze their purpose and behavioral pattern. In the following sections are referenced honeypots and honeynets that have been developed for the purpose of diverting attacks from SCADA systems.

3.1 SCADA Honeynet Project

Cisco Critical Infrastructure Assurance Group (CIAG) began the SCADA Honeynet Project in 2004 and ended it in 2005. It is made up of a series of Python scripts, each of which implements a different service of the simulated PLC. Honeyd, a small daemon that creates virtual hosts on a network, is heavily used in this project. The hosts can be set up to run any number of services, and their personalities can be tweaked to make them appear to be running specific OSs. The Honeyd daemon can be configured to act as a computer with a PLC's OS fingerprint and run Cisco scripts on the appropriate ports. The Honeyd PLC simulates TCP/IP stack, Telnet, FTP, HTTP, and Modbus TCP with the help of these scripts. In summary, these scripts appear incomplete, the services are only partially implemented, and the implemented functionality is neither realistic nor interactive. (Buza et al., 2014)

3.2 Digital Bond's Honeynet

Digital Bond created this honeynet in 2006. It consists of two Linux-based virtual machines (VMs). The first VM works as a PLC (Modicon Quantum PLC) honeypot and the second VM runs the Honeynet Project's Generation III honeywall, which is enhanced with Digital Bond's Quickdraw IDS signatures. The goal of the second VM is to keep track of all the network activity

in order to detect and log any malicious attacks against the simulated PLC. The first VM, that emulates a PLC, runs five services: HTTP, FTP, SNMP, Telnet and Modbus TCP. The FTP, HTTP, and Modbus services are implemented using Java applications, while the Telnet and SNMP services are implemented using Python scripts. The core of the VM is Honeyd, which is responsible for the routing of the virtual host's network traffic (the data streams and datagrams). The Digital Bond's SCADA Honeynet is a significant improvement over the Cisco Honeynet Project. It is possible to fool scanning and information-gathering tools (such as nmap or nessus) into thinking it is a real PLC using the returned service banners and OS fingerprint, making it effective against automated attacks and tools. The simulated services, on the other hand, provide very little interaction and may not be able to keep an attacker interested long enough to uncover new targeted PLC attacks. Digital Bond's Honeynet scripts are available on GitHub. (Buza et al., 2014)

3.3 Conpot

In 2013 the Honeynet group released a honeypot for SCADA named Conpot. Conpot is a low-interaction server-side ICS honeypot and supports a variety of SmartGrid use cases. Conpot's ease of use, combined with simulated PLCs, is one of its most appealing features. Conpot supports protocols such as Modbus TCP, HTTP, IEC104, FTP, TFTP, S7Comm, BACnet, and SNMP, and it, like other honeypots, can keep track of attacks. It also supports a Human-Machine Interface (HMI) via the HTTP server on TCP port 80 and can be configured to display a page of one's choosing (Scott & Carbone, 2014). Conpot is open source and can be extended to emulate more complex SCADA systems. It can be installed using Docker or Virtualenv. (*Conpot Documentation*, 2018)

3.4 Crysyst PLC Honeypot (CryPLH)

CryPLH is a high-interaction honeypot that emulates a Siemens S7-300 PLC, with HTTP/HTTPS, S7comm, and SNMP services running on a Linux host modified to accept connections on specific ports. An abstract model of the CryPLH is shown in Figure 3.1. The TCP/IP Stack is simulated using the Linux kernel, and the S7comm protocol is simulated by showing an incorrect password response.

HTTP/HTTPS implementation

HTTP is one of the most common Internet protocols, and it's also supported by the Siemens Simatic 300 PLC. HTTPS (Secure HTTP) is an extension of the HTTP protocol. It is not a stand-alone protocol; rather, it is a layering of HTTP over the SSL/TLS protocol. The device makes use of a web server from the MiniWeb project, which was written in C to provide a small HTTP server that is both efficient and portable. It also has a login procedure for gaining access to the device (Buza et al., 2014).

SNMP implementation

Buza, Dániel István, et al. created a custom SNMP Agent on the honeypot device after a thorough exploration of the Siemens PLC SNMP database and implementation. The Agent, like the original SNMP service, listens on UDP port 161 and accepts and responds to SNMP requests and responses. Rather than using real MIBs (Management Information Databases), it parses an XML file containing a list of records found on the real PLC. These records all have an Object Identifier (OID) and a type attribute. They either contain the static data they represent or a special mark and string that instructs the interpreter on how to create or retrieve dynamic data (Buza et al., 2014).

S7comm protocol implementation

A simulation of the S7comm protocol is implemented on the PLC honeypot with a python script. The communication is going through PLC's TCP port 102. In (Buza et al., 2014) it is described that the PLC requires a password to make changes to it. And they decided to simulate this behavior with the exception that every time a password is entered, the PLC would respond with an invalid message (Buza et al., 2014).

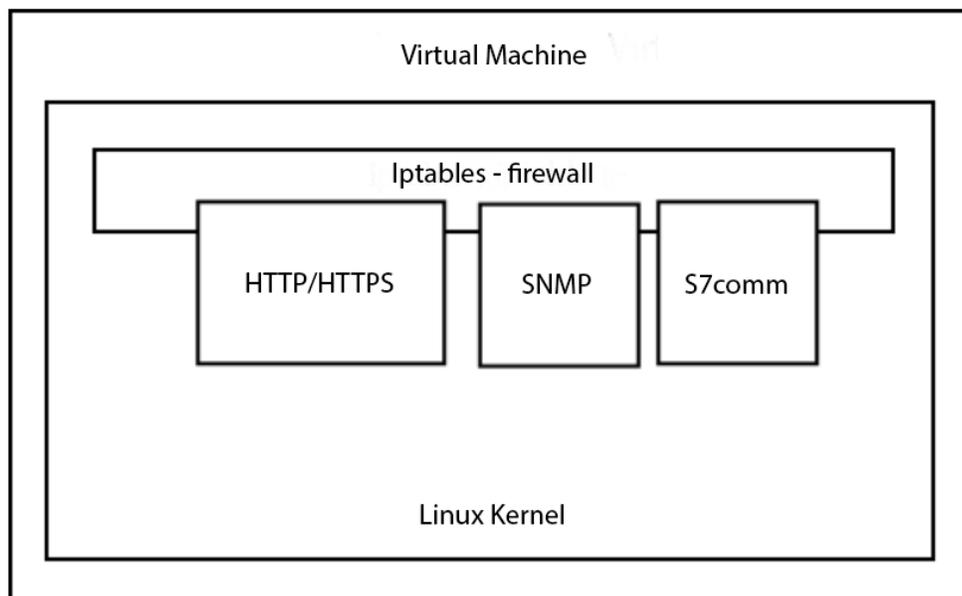


Figure 3.1 The abstract model of the CryPLH honeypot (Buza et al., 2014)

3.5 SHaPe

SHaPe (Scada HoneyPot) is a low-interaction honeypot that can be used on substation automation systems. Specifically, it can simulate Intelligent Electronic Devices (IEDs) that are compatible with the IEC-61850 MMS communication over TCP/IP connection. The SHaPe honeypot does not handle generic substation events based on GOOSE or transmission of sampled values that are mapped to other protocol stacks according to IEC 61850. SHaPe, on the other hand, supports and executes all IEC 61850 services mapped to MMS. The state of the emulated IED will be updated if a service creates, modifies, or deletes an object.

SHaPe can be used to simulate a specific type of IED. SHaPe, on the other hand, can run multiple copies of the emulated IED, each with its own IP address. In this way, SHaPe makes it simple to increase the number of monitored IP addresses and, as a result, the attack surface associated with a specific type of IED (Kołtyś & Gajewski, 2015).

SHaPe is a GPLv3-licensed open-source project. It is linked to the open-source projects libiec61850 and Dionaea. libiec61850 is a C library that implements the IEC 61850/MMS and IEC 61850/GOOSE communication protocols as a server and client. To handle IEC 61850 communication, SHaPe uses a modified version of libiec61850. Dionaea is a versatile honeypot

that can be easily customized with third-party modules. SHaPe is implemented as a Dionaea module (*The SHaPe Project Website*).

3.6 S7commTrace

S7commTrace is a honeypot based on the S7 protocol. This protocol is running on PLCs of Siemens S7-300, 400, 1200 and 1500 series. S7commTrace consists of four modules, TCP Communication module, S7 Communications Protocol Simulation module, Data Storage module and User Template, as shown in Figure 3.2. The TCP communication module listens on port 102, transmitting incoming data to the S7 protocol simulation module and responding to the remote peer. The simulation module parses the received data and creates a reply according to the User Template. And the reply is submitted to the TCP Communication module to be packaged. The Data Storage module handles the data storage requests (Xiao et al., 2017).

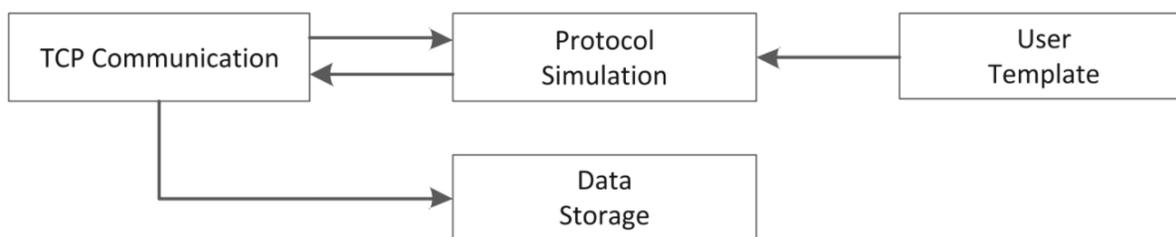


Figure 3.2 Module of S7commTrace (Xiao et al., 2017, 416)

3.7 HoneyPLC

HoneyPLC is a high-interaction, malware-collecting honeypot for ICS. It simulates TCP/IP, HTTP, SNMP and S7comm protocols. HoneyPLC consists of four modules, as shown in Figure 3.3. The PLC Profile Repository, the Integration Framework, the Network Services and the Interaction Data modules.

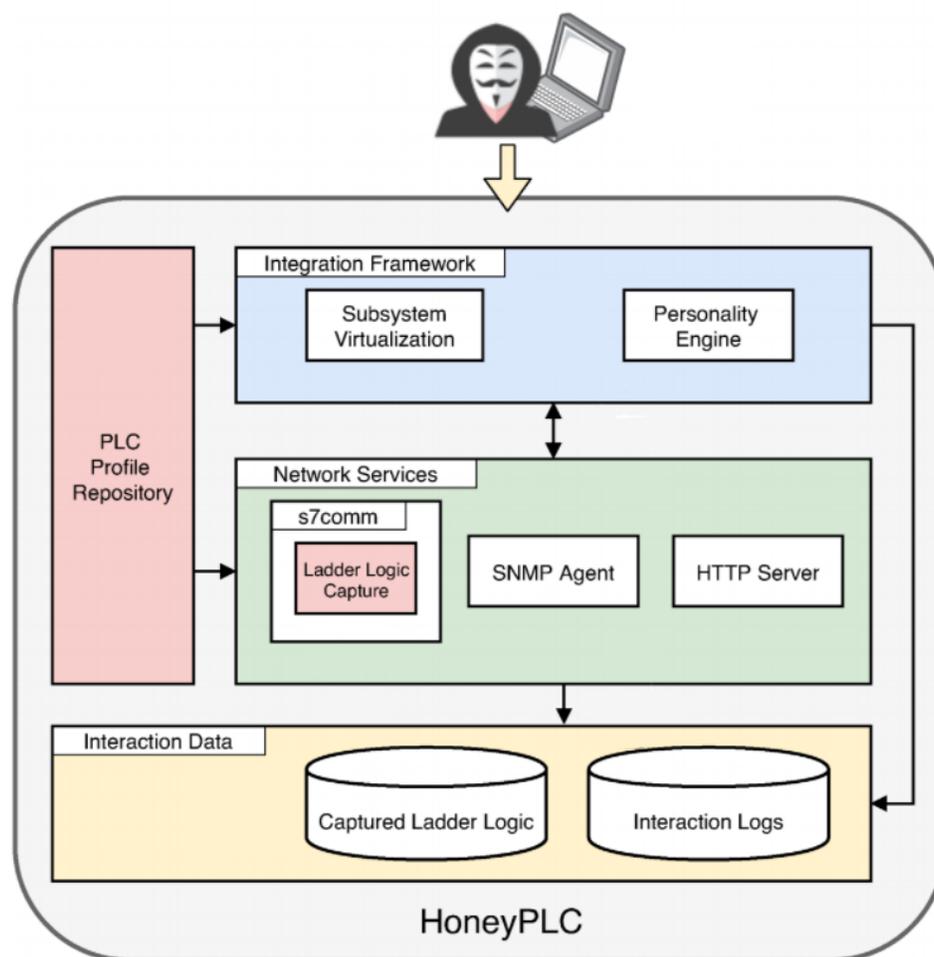


Figure 3.3 The architecture of HoneyPLC (López-Morales et al., 2020, 283)

The PLC Profile Repository contains several PLC profiles. These profiles have information on real-life PLCs. In the initial setup of the honeypot, a profile is chosen to be simulated. After the initialization of the HoneyPLC, when initial contact is established, the HoneyPLC's Personality Engine (in Integration Framework module), which is based on features provided by the Honeyd tool, will handle all TCP/IP requests.

Depending on the attacker's approach, the appropriate simulated protocol, in the Network Services module, responds to them. For example, if they try to initiate an S7comm connection, the S7comm server replies with the requested information, which is forwarded to the attacker by the Integration Framework. Meanwhile, the S7comm server records all communications,

including the attacker's source IP address and memory block requests to the HoneyPLC repository, which is managed by the Interaction Data module. (López-Morales et al., 2020)

HoneyPLC is available for download on GitHub.

3.8 DiPot

DiPot is a distributed industrial honeypot system. Its architecture consists of three components: Honeypot Node (HN), Data Processing Node (DPN) and Management Node (MN).

Honeypot nodes are based on an extended version of Conpot, and it is used as an interface with the outside world and potential attackers.

Raw log files are forwarded by HNs to the Data Processing Node (DPN), which filters them, removes duplicate data, and applies a custom format. The output is saved in the DPN's central database. Meanwhile, the node clusters the formatted logs for each HN separately using the k-means algorithm based on timestamp, source IP, protocol, and function/slave Identifier.

Finally, the DPN sends its output to the Management Node, which displays data in an easy-to-understand format. HNs are plotted on a world map based on their location, and information about each HN is organized according to the classification criteria mentioned above (Dalamagkas et al., 2019).

In Table 3.1 is shown the summary of the mentioned honeypots.

Name	Supported Protocols	Open-source	Extensibility	Dockerized	Support
Low-interaction Honeypots					
SCADA Honeynet Project	HTTP, FTP, TCP/IP, Modbus, TCP, Telnet	Yes	No	No	No, Last updated in 2005
Digital Bond's Honeynet	HTTP, FTP, Modbus, TCP, Telnet, SNMP	Yes	No	No	No
Conpot	HTTP, FTP, TFTP, Modbus, TCP, SNMP, S7comm, IEC/104, BACnet	Yes	Medium	Yes	Yes

SHaPe	IEC 61850-MMS	Yes	Medium	No	No, Last updated in 2015
DiPot	HTTP, Modbus TCP, SNMP, S7comm	No	Medium	No	Unknown
High-interaction Honeypots					
CryPLH	HTTP/HTTPS, SNMP, S7comm	No	No	No	Unknown
S7commTrace	S7comm	No	Medium	No	Unknown
HoneyPLC	HTTP, TCP/IP, SNMP, S7comm	Yes	High	No	No, Last updated in 2020

Table 3.1 Summary of the mentioned honeypots

4 Modern Automatic Deployment Methods

4.1 Historical Background

Software deployment has improved over the last three decades, which is essential in a world where people are increasingly living and working online. Chronologically deployment methods can be divided into three eras. In the traditional deployment era, the virtualized deployment era, and the container deployment era.

Traditional deployment era

Organizations used to run applications on physical servers. Developers would program multiple versions of the same application if they wanted to create applications that worked on different setups, and they would often need direct access to the machine to install the software. It's easy to see how this could be time and money-consuming. Users could only access the software from the physical machine or, in some cases, from a network connection even after a successful deployment. Another disadvantage of apps that run directly on the physical machine is the potential for the software to have a negative impact on the machine, such as using excessive amounts of resources or causing software problems due to bugs or other unforeseen interactions with the operating system.

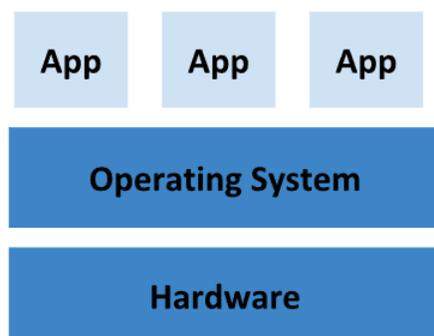


Figure 4.1 Traditional Deployment

Virtualized deployment era

The above problems were partially solved with the use of Virtual Machines (VMs). With virtualization, physical hardware turned into software components enabling us to run multiple

VMs on a single physical server's processor. Each VM has its own hardware and operating system. This is possible thanks to Hypervisor, which converts physical resources into virtual counterparts. Hypervisor is a piece of software that allows one host computer to support multiple guest VMs by sharing its resources virtually.

Virtualization isolates applications between VMs and increases security by preventing one application's information from being freely accessed by another. Virtualization also improves resource utilization on a physical server, compared to the traditional deployment method, improves scalability by allowing applications to be easily added or updated, and lowers hardware costs. But, as it is shown in Figure 4.2, an OS must be installed in every VM to be operational, which means that a lot of physical resources are utilized to deploy a limited amount of applications.

Container deployment era

Containerization was created to address the drawbacks that were mentioned above. Containerization is a type of OS virtualization in which applications are run in isolated spaces called containers while all sharing the same OS. Compared to VMs, containers have less strict isolation properties, allowing applications to share the same OS. As a result, containers are considered light and portable across OS and cloud distributions, because they are decoupled from the underlying infrastructure, as shown in Figure 4.3.

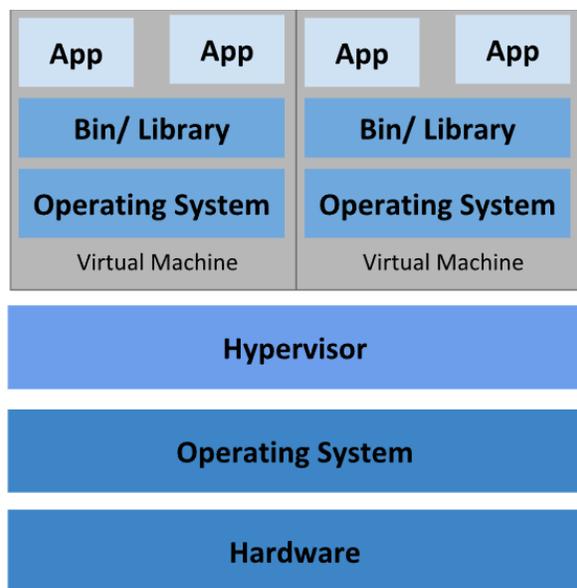


Figure 4.2 Virtualized Deployment

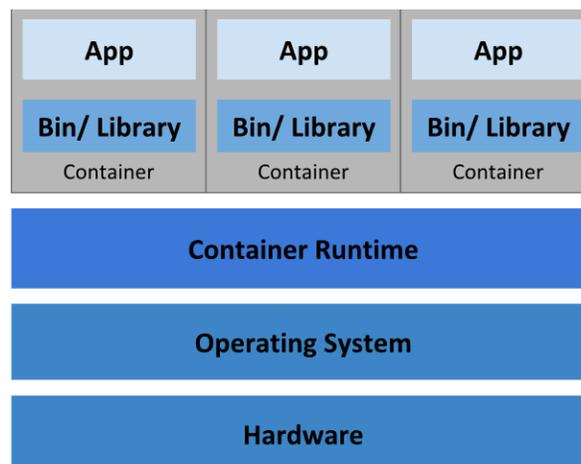


Figure 4.3 Container Deployment

4.2 Introduction to Container Technology

An application usually consists of a number of components. If the components are small in number and large then it is acceptable to run each component to a dedicated VM and isolate their environments by giving each one its own instance of the OS. But when the components become smaller and larger in number, it requires more VMs in order to run an application, which increases the hardware cost and likely the amount of staff needed to operate a large amount of VMs.

Rather than using virtual machines to isolate each microservice's environment, developers are using Linux container technologies. Containers enable us to run multiple services on the same host machine while isolating them from each other, but with far less cost.

A containerized process, like all other processes, runs inside the host's operating system. However, the container process remains isolated from other processes. From the process' perspective, it appears to be the one running on the machine and in the OS.

Docker Container Platform

The Docker container platform is the most widely known and used by developers. Docker was the first container system that allowed containers to be easily moved between machines. It made

it easier to package not only the application but also all of its libraries and other dependencies, as well as the entire OS file system, into a simple, portable package that could be used to provision the application to any other Docker-enabled machine.

When a Docker-based application is executed, it sees the exact filesystem contents that its developer has included with it. It sees the same files whether it's on the development machine or a production machine, even if the production server is running a different Linux distribution.

This is analogous to generating a VM image by installing an OS, installing a program in it, and then distributing and running the entire VM image. Instead of employing VMs to accomplish application isolation, Docker employs Linux container technologies to provide almost the same level of isolation as VMs.

The process of creating and deploying a Docker image is as follows. At first, the developer builds an image of their application, containing all of its dependencies in the container filesystem, and then pushes it to the Docker registry. Anyone who has access to this specific image can pull it to any Docker-enabled system. Docker uses the image to create an isolated container that runs the binary executable specified in the image. In Figure 4.4 is shown the above process.

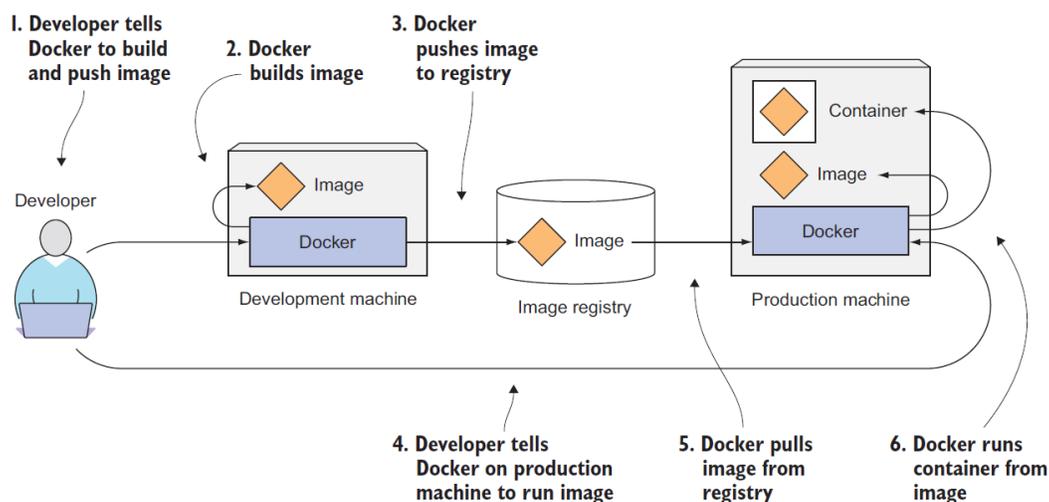


Figure 4.4 Development and Deployment of Docker images (Luksa, 2017, 13)

4.3 Introduction to Kubernetes

Kubernetes is a software platform for easily deploying and managing containerized applications. It uses Linux container features to run heterogeneous applications without having to know any internal details about them or manually deploying them on each host. Kubernetes allows you to run containerized applications across thousands of computer nodes as if they were all part of a single massive computer. It abstracts away the underlying infrastructure, making development, deployment, and management easier for development and operations teams alike. Using Kubernetes to deploy applications is the same whether your cluster has a few nodes or thousands. It does not matter how big the cluster is. Adding cluster nodes simply increases the amount of resources accessible to applications that have been deployed.

A Kubernetes system consists of a master node and any number of worker nodes. Kubernetes deploys applications to the cluster of worker nodes after the developer submits a list of applications to the master. It makes no difference to the developer or the system administrator which node a component lands on. If it is specified by the developer many applications can be deployed on the same worker node. Otherwise, the applications will be dispersed across the cluster.

Kubernetes functions as a cluster's OS. It frees application developers from needing to build infrastructure-related services within their apps. And with this developers can focus on the application's functionality and features.

The infrastructure-related services that Kubernetes provide are the following.

- **Service discovery and load balancing:** It can expose a container to a network or the open internet using an IP address or a DNS name. Kubernetes also load balances automatically the traffic between the deployed containers, to achieve system stability.
- **Storage orchestration:** A variety of storage systems can be mounted on Kubernetes like local storage, or cloud storage.
- **Automated rollouts and rollbacks:** Kubernetes allows you to specify the desired state for your deployed containers, and it can change the actual state to the desired state at a set rate.
- **Automatic bin packing:** Depending on the system's resources and a cluster of nodes that are provided to Kubernetes, it can deploy containers efficiently.

- **Self-healing:** The user can define health checks that work as instructions to Kubernetes to restart, replace or kill containers and divert the traffic from them, in case they fail.
- **Secrets and configuration management:** Kubernetes can be used to store and manage sensitive data like passwords, OAuth tokens, and SSH keys. Without rebuilding container images or exposing secrets in the stack configuration, the user can deploy and update secrets and application configuration.

4.4 The Architecture of a Kubernetes Cluster

A Kubernetes cluster consists of two types of nodes. The master node and the worker nodes. The master node hosts the Kubernetes Control Plane, which is responsible for controlling and managing the whole Kubernetes system and the worker nodes run the applications that have been deployed.

The Control Plane

The Control Plane is in charge of managing and operating the cluster. It is made up of several components that can run on a single master node or be distributed across numerous nodes and duplicated for high availability. Control Plane's components are

- The Kubernetes API Server: It enables the administrators to communicate with the other Control Plane components.
- The Scheduler: It schedules the applications.
- The Controller Manager: It is in charge of cluster-level functions like replicating components, keeping track of worker nodes, and handling node failures.
- etcd: It is a dependable distributed data storage that stores the cluster configuration indefinitely.

The Worker Nodes

The worker nodes, as stated above, run the containerized applications. The following components are responsible for running, monitoring, and providing services to the deployed applications.

- The container runtime: It runs the containers.
- The Kubelet: It manages containers on its node and communicates with the API Server.

- The Kubernetes Service Proxy: It is in charge of the load-balancing between application components.

4.5 Deployment Process

Deploying an application in Kubernetes is generally achieved in three steps. First, the developer needs to package one or more container images, then push those images to an image registry. And finally, post the application description file to the Kubernetes API server. This file is usually a JSON or a YAML file.

The description includes details like the container image or images that contain your application components, how those components are related to one another, and which ones must be run on the same node and which do not. We can also specify how many replicas we want to run for each component. Furthermore, the description specifies which components provide a service to either internal or external clients and should be exposed via a single IP address and made discoverable by the other components.

The Scheduler schedules the specified groups of containers onto the available worker nodes when the API server processes the application's description, based on the computational resources required by each group and the unallocated resources on each node at the time. The Container Runtime (e.g. Docker) is then instructed by the Kubelet on those nodes to pull the required container images and run the containers.

Figure 4.5 shows a simplified overview of the Kubernetes architecture and an example of application deployment. In this example, the application descriptor lists four containers, grouped into three sets, that are called pods. A pod is a co-located group of containers and represents the basic building block in Kubernetes. Each of the first two pods has only one container, whereas the last one has two. That means both containers must run in the same location and should not be isolated from one another. A number appears next to each pod, indicating the number of replicas of that pod that must run in parallel. Kubernetes will schedule the specified number of replicas of each pod to the available worker nodes after receiving the descriptor. The Kubelets on the worker nodes will then instruct Docker to download and run the container images from the image registry.

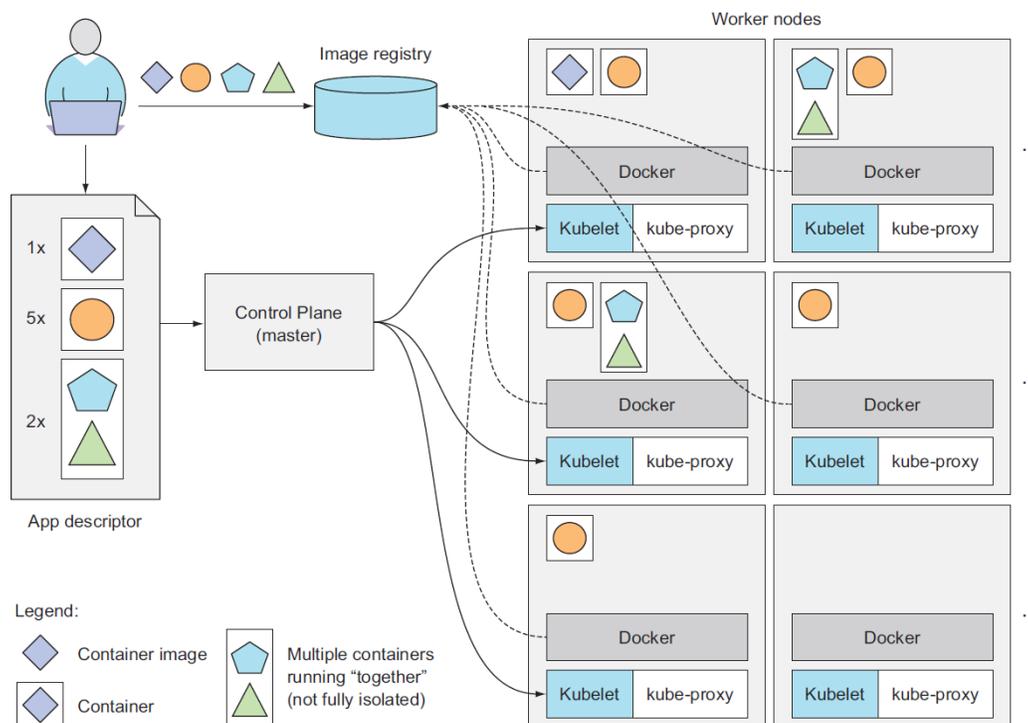


Figure 4.5 A Basic Overview of the Kubernetes Architecture and an Application Running on Top of It. (Luksa, 2017, 20)

4.6 Helm Charts

At this point Helm Charts should be mentioned because they will be referred later. Helm is a package manager for Kubernetes. Helm is an open-source project that was originally developed by Deis Labs and is now maintained by Cloud Native Computing Foundation (CNCF). Helm was created with the intention of giving users a better way to manage all of the Kubernetes YAML (description) files that they create on Kubernetes projects. Helm developed Helm Charts as a means of resolving the description file management problem. Each chart is a collection of one or more Kubernetes manifests; a chart can also have child and dependent charts. When we run the install command for the top-level chart, Helm installs the entire project's dependency tree.

The Kubernetes project's goal is to manage your containers, but it cannot use template files. Helm gives us the ability to create template files and add to them variables and functions. These files are truly generic and can be used across large teams or organizations to deploy scalable applications where their parameters can be changed at any time.

5 HoneyChart

The use of cloud-based services and IoT devices in an industrial environment has increased significantly. This might result in an increased exposure of industrial networks to the internet, with an increased likelihood of an attack to the network. As mentioned previously ([Section 1, p. 7](#)), honeypots can provide early detection of unauthorized network activity and this along with the Kubernetes orchestration system, we are able to deploy containerized honeypots timely, efficiently and with low resource cost.

To make the deployment process easier and faster, we developed the HoneyChart solution. HoneyChart is a tool for creating Helm Charts ([Section 4.6, p. 44](#)) of containerized honeypots. The charts are able to contain descriptions and deployment instructions for a single honeypot or multiple ones. The creation of the honeypot charts is made via the HoneyChart web application. The application's front-end was developed in HTML, CSS and Javascript and the back-end was built with Node.js (*About Node.js*, 2022).

5.1 Home Page

When entering the home page of the application, the user has two options to pick from. Custom Honeypots and Prebuild Interfaces. The Custom Honeypots option lets us create custom Helm Charts by using the available honeypots on the platform. The Prebuild Interfaces gives us the option to select prebuilt honeypot Helm Charts that can simulate for example PLC communication protocols or Microsoft Windows protocols. In Figure 5.1 is shown the application's home page.



Figure 5.1 Home Page

5.2 Custom Honey pots Page

On the Custom Honey pots page, the user first chooses which honeypot is going to use. At the moment there are three available honeypots on the platform. Conpot (Honeynet/conpot - Docker Image), Cowrie (Cowrie/cowrie - Docker Image) and Dionaea (Dinotools/dionaea - Docker Image), which are open-source, containerized and available for download from the Docker platform ([Section 4.2, p. 39](#)). After choosing the honeypot, the user can enable/disable the available protocols of the honeypot as shown in Figure 5.2. Also, they can select which Kubernetes ServiceType they prefer for their deployment (Nodeport or LoadBalancer). Table 5.1 details all the available Kubernetes ServicesTypes. And in the two final fields the user fills in the number of replicas and the file path for the honeypot logs. Figure 5.2 shows the form that was described above.

After filling the form and pressing the “Add” button the application performs a series of validations to make sure that the user’s options are in order. If there is an error, the application will inform the user, an example is shown in Figure 5.3. Otherwise, the user has the option to add more honeypots to the chart, or finish the process by pressing the “Create” button. The final step is depicted on Figure 5.4. If they choose to add more honeypots to the chart, they just have to follow the previous steps again. Otherwise, the chart gets downloaded to the user’s computer and they can now deploy it on their Kubernetes cluster.

With the options that HoneyChart provides, the user can combine services from different honeypots to build certain profiles. In an industrial environment, it is possible to create a honeypot profile that emulates an industrial device, e.g. PLC, to attract attackers away from

critical industrial devices. And by combining HoneyChart and Kubernetes, the honeypot deployment becomes a fast and easy process.

Service	Enabled	Port/Value
Name		myChart
HTTP	<input type="checkbox"/>	80
FTP	<input checked="" type="checkbox"/>	21
TFTP	<input type="checkbox"/>	69
Modbus	<input checked="" type="checkbox"/>	502
SNMP	<input checked="" type="checkbox"/>	161
S7comm	<input checked="" type="checkbox"/>	102
BACnet	<input checked="" type="checkbox"/>	47808
IPMI	<input type="checkbox"/>	623
Ethernet/IP(enip)	<input checked="" type="checkbox"/>	44818
Nodeport	<input type="checkbox"/>	
LoadBalancer	<input checked="" type="checkbox"/>	25.25.25.25
Replicas		2
Logs path (optional)		/chart-logs

Figure 5.2 Custom Honeypots Form

Service	Enabled	Port/Value
Name		myChart
HTTP	<input type="checkbox"/>	80
FTP	<input checked="" type="checkbox"/>	21
TFTP	<input type="checkbox"/>	69
Modbus	<input checked="" type="checkbox"/>	502
SNMP	<input checked="" type="checkbox"/>	161
S7comm	<input checked="" type="checkbox"/>	102
BACnet	<input checked="" type="checkbox"/>	47808
IPMI	<input type="checkbox"/>	623
Ethernet/IP(enip)	<input checked="" type="checkbox"/>	44818
Nodeport	<input type="checkbox"/>	
LoadBalancer	<input checked="" type="checkbox"/>	266.52.120
Replicas		1
Logs path (optional)		/chart-logs

False IP Address

Figure 5.3 Example With False IP

Figure 5.4 Final Step Before Chart Creation

Kubernetes ServiceTypes

Kubernetes ServiceTypes allow you to specify the type of Service you want. With ClusterIP being the default value.

These ServiceTypes are

- **ClusterIP:** Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.
- **Nodeport:** Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- **LoadBalancer:** Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- **ExternalName:** Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

Table 5.1 Kubernetes ServiceTypes (Service, 2022)

5.2.1 How It Works

Overall the Custom Honey pots page is divided into three sections. The dropdown menu where the user selects a honeypot, the honeypot options and the information panel (Figure 5.5).

The screenshot displays the HoneyChart interface for creating a custom honeypot. It is divided into three main sections:

- Dropdown Menu:** Located at the top left, it features a dropdown menu currently set to "Cowrie".
- Honeypot Options:** Located in the middle left, it includes:
 - SSH: A toggle switch (off), a port input field set to "22", and a help icon (?).
 - Telnet: A toggle switch (off), a port input field set to "23", and a help icon (?).
 - Logs path (optional): An input field containing "/logs".
 - An "Add" button at the bottom.
- Information Panel:** Located on the right side, it displays details for the selected chart:
 - Your Chart:** Name: myChart, Service: LoadBalancer IP: 25.25.25.25, Replicas: 2.
 - Conpot settings:** Services: ftp Port: 21, tftp Port: 69, modbus Port: 502, s7comm Port: 102, bacnet Port: 47808, enip Port: 44818. Path: /chart-logs.
 - A "Create" button at the bottom.

Figure 5.5 Custom Honey pots Page Sections

Initially the only section that is visible is the dropdown menu. After the user selects a honeypot, the page loads on the honeypot options section the corresponding options. The honeypot options section is divided into six sections. The name, the services, the Kubernetes ServiceTypes, replicas, the logs path and the “Add” button (Figure 5.6). If the user is in the process of adding the first honeypot to the Helm Chart, then all six sections are shown. Otherwise the name, replicas and ServiceTypes sections are hidden. The reason for this design is that the user needs to fill these sections only once.

In the name section, the user can choose the name for their Helm Chart. Although some constraints were added. They can only type 3 to 20 letters, numbers and hyphens (-). The user can mouse-over on the question mark, next to the name field, to get informed about this constraint.

The screenshot shows a form titled "Add honeypots to your chart" with a dropdown menu set to "Cowrie". The form is divided into several sections, each highlighted with a colored border and labeled on the right:

- Name:** A text input field containing "myChart" with a question mark icon.
- Services:** A section containing two rows. The first row has a toggle switch for "SSH" (disabled), a dropdown menu with "22", and a question mark icon. The second row has a toggle switch for "Telnet" (disabled), a dropdown menu with "23", and a question mark icon.
- ServiceType:** A section with two radio button options: "Nodeport" (selected) and "LoadBalancer". The "LoadBalancer" option has a text input field containing "19.34.54.63" and a question mark icon.
- Replicas:** A section with a dropdown menu set to "1" and a question mark icon.
- Logs Path:** A section with a text input field containing "/logs" and a question mark icon.

At the bottom of the form is a blue "Add" button.

Figure 5.6 Honeypot Options Sections

In the services section, the user can enable or disable the services that they need from the selected honeypot. They can do this by clicking on the corresponding switches. By enabling a service, automatically the text field next to it becomes enabled and by default it contains the most common port that it is used for the selected service. If the user needs more information about the service, they can click on the question mark and they will be redirected to an information page.

The ServiceType section has two options, Nodeport and LoadBalancer (Table 5.1). If the user selects Nodeport, they do not need to fill the IP (Internet Protocol) address field. If they select the LoadBalancer option, then they must fill the IP address field using the IPv4 format (IBM, *IPv4 and IPv6 address formats*). The question mark next to the ServiceType options redirects the user to an information page that explains all the Kubernetes ServiceTypes.

In the replicas section the user can specify how many replicas of the Helm Chart that they need. The allowed range of values is from 1 to 100. This information can be seen by hovering over the question mark next to the replicas text field.

Logs path allows the user to specify in which directory the honeypot logs will be extracted, on

the Control Plane (master node).

After the user fills the necessary fields and clicks the “Add” button, the process of validation and the creation of the Helm Chart structure begins. At first the program initializes the data structure that will be used to hold all the information that the user filled in the honeypot options. The data structure is shown in Figure 5.7.

```
var data = {
  name: "",
  service: {
    type: "",
    lbIp: null,
  },
  replicaCount: 0,
  honeypots: {
    names: [],
    dionaea: {
      volumes: [],
      services: [],
      containerports: [],
      protocols: [],
    },
    conpot: {
      volumes: [],
      services: [],
      containerports: [],
      protocols: [],
    },
    cowrie: {
      volumes: [],
      services: [],
      containerports: [],
      protocols: [],
    },
  },
};
```

Figure 5.7 Custom Honeypot Data Structure

Afterwards the program checks every service switch. For every switch that it is enabled, the program will check the validity of the user-defined port number. This was achieved with the use of a regular expression (Table 5.2), which allows integer values that range between 1 and 65535. Port numbers are 16-bit unsigned integers, which means that they range from 1 to $2^{16} - 1$ (*Port (Computer Networking)*). If the value is invalid, then an error message is shown “Invalid port number”. If it is valid, then the program pushes the service information to the corresponding honeypot services array in the following form: <service name>:<port number>. It also pushes the

container port to the containerports array. The container port number is predefined by the creator(s) of each honeypot. This information can be found in the docker-compose.yml file in all honeypot docker images that are available on Honeychart. Also the program pushes into the protocols array the corresponding communication protocol, either TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). During the service registration, we keep count of how many services are enabled. If they are zero then an error message is shown “Please select at least one service”. Otherwise the program continues to the next step.

If this is the first honeypot addition, then the name, service type and replicas get validated and added to the chart's data structure. The name of the chart gets validated by using a regular expression (Table 5.2). The regular expression only allows 3 to 20 letters, numbers and hyphens (-). This constraint was created because the name of the Helm Chart is used on certain commands when the program creates the Chart. Which means that if a malicious user finds out how the program works, then they can use the name field to execute commands of their choosing. If the name is invalid, then an error message is shown “Chart name must contain only 3 to 20 letters, numbers or '-'”. Otherwise the name gets added to the data structure.

The number of replicas gets validated with a regular expression (Table 5.2). The allowed values are integers ranging from 1 to 100. This constraint was created based on the available resources of the Kubernetes cluster. If the number is invalid then an error message is shown “The number of replicas can be from 1 to 100”. If it is valid, then the number gets added to the data structure.

Name	Regular Expression
Helm Chart Name	[a-zA-Z0-9-]{3,20}
Port Numbers	[1-9] [1-9][0-9]{1,3} [1-5][0-9]{4} 6[0-4][0-9]{3} 65[0-4][0-9]{2} 655[0-2][0-9] 6553[0-5]
IP Address	((([1-9]?\\d 1\\d\\d 2[0-5][0-5] 2[0-4]\\d)\\.){3}([1-9]?\\d 1\\d\\d 2[0-5][0-5] 2[0-4]\\d))
Replicas	([1-9] [1-9][0-9] 100)

Table 5.2 Regular Expressions Used in the Program

The program checks if the user chose Nodeport or LoadBalancer as a ServiceType. If they have selected Nodeport, then the program sets the data.service.type variable to “Nodeport” and the

data.service.lbIp variable to null. Otherwise if the user has selected LoadBalancer, then the program validates the IP address by using a regular expression (Table 5.2). This regular expression follows the IPv4 format (IBM, *IPv4 and IPv6 address formats*). If the IP is invalid then an error message is shown “False IP Address”. If the IP address is valid, then the variable data.service.type gets set to “LoadBalancer” and the variable data.service.lbIp gets the value of the user-defined IP address. In case the user did not pick any service type then an error message is shown “You must choose service type (Nodeport/LoadBalancer)”. At this point the program has all the general information of the Helm Chart and it can be shown on the page.

The information panel shown in Figure 5.5 is divided into two sections. The General Information section and the Honeypot Information section (Figure 5.8). The General Information section presents the name of the Helm Chart, its ServiceType and the number of replicas. The Honeypot Information section shows all the services that the user selected and the logs path.

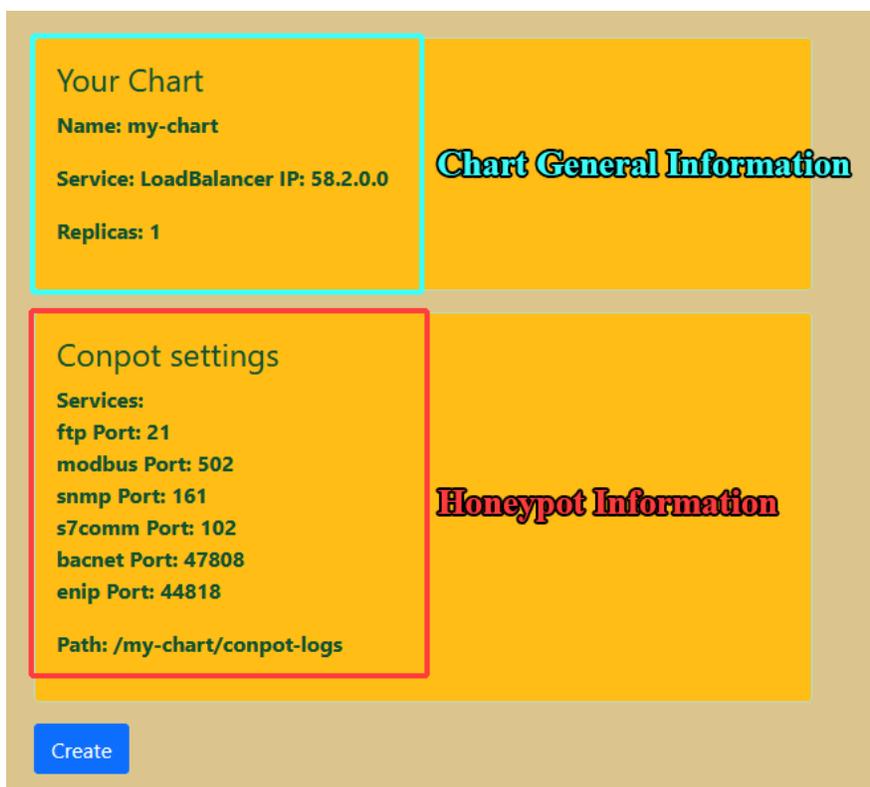


Figure 5.8 Chart Information

After showing the general information, the program pushes in the array data.honeypots.names (Figure 5.7) the name of the selected honeypot (e.g. “conpot”). This array is useful for the

creation of the Helm Chart in the backend. Then we push the logs path to the `data.honeypots.<honeypot name>.volumes` array.

Now that we have all the necessary information for the first honeypot addition, the program shows the Honeypot Information on the Information Panel. In case the user wants to add more honeypots they can select one from the drop down menu and repeat the previous steps without having to name the chart, select the service type, and replicas. Otherwise they can click the “Create” button to send the filled data structure to the backend and begin the process of Helm Chart creation.

5.2.2 The Creation of the Helm Chart

The process of the Helm Chart creation begins after the user has clicked on the “Create” button. Then the data structure (Figure 5.7) gets converted to a JSON string on the frontend and with a POST request (*POST - HTTP | MDN, 2021*) the program sends the JSON string to the backend.

When the server receives the JSON string, it validates the name of the Helm Chart for security purposes. Afterwards the program initializes the `generic_values` (Figure 5.9), `generic_deployment` (Figure 5.10), and `generic_service` (Figure 5.11) data structures. The process continues with specifying the logs path for both the container and the worker node. Depending on the selection of honeypots made by the user, the program fills out values, services and deployment data to the corresponding data structures. Then we create three YAML files, `values.yaml`, `deployment.yaml` and `service.yaml` using the three data structures (`generic_values`, `generic_deployment`, and `generic_service`).

Now that we have all the needed information, the program executes the command **helm create** `<Helm Chart Name>` (*Helm Create, 2022*). This command creates a chart directory with all of the necessary files and directories, an example is shown in Figure 5.12. Then the program moves the `values.yaml` file to the chart’s root directory and the `deployment.yaml` and the `service.yaml` to the `templates` directory. To make sure that the server has access to the Helm Chart directory, the program executes the command **chmod -R 770** `<Helm Chart Name>` (MacKenzie & Meyering, 2007). When the chart directory is ready the server creates a zip file from the directory using the command: **zip -r** `<Helm Chart Name> .zip <Helm Chart Name>` (*Zip(1): Package/compress Files - Linux Man Page, 2007*) and then the server sends the zip file to the client as a response.

```

generic_values = {
  honeypots: {},
  serviceAccount: {
    create: true,
    annotations: {},
    name: "",
  },
  podAnnotations: {},
  podSecurityContext: {},
  securityContext: {},
  service: {
    type: "",
    extTrafficPolicy: "Local",
  },
  volumes: {},
  replicaCount: 1,
  ingress: {
    enabled: false,
    className: "",
    annotations: {},
    hosts: {
      "- host": "chart-example.local",
      " paths": {
        " - path": "/",
        " pathType": "ImplementationSpecific",
      },
    },
    tls: [],
  },
  resources: {},
  autoscaling: {
    enabled: false,
    minReplicas: 1,
    maxReplicas: 100,
    targetCPUUtilizationPercentage: 80,
  },
  nodeSelector: {},
  tolerations: [],
  affinity: {},

  imagePullSecrets: [],
  nameOverride: "",
  fullnameOverride: "",
};

```

Figure 5.9 generic_values

```

generic_deployment = {
  apiVersion: "apps/v1",
  kind: "Deployment",
  metadata: {
    name: '{{ include "" + CHART_NAME + ".fullname" . }}',
    labels: '{{- include "" + CHART_NAME + ".labels" . | nindent 4 }}',
  },
  spec: {
    '{{- if not .Values.autoscaling.enabled }}: "aa",
    replicas: "{{ .Values.replicaCount }}",
    '{{- end }}': "aa",
    selector: {
      matchLabels:
        '{{- include "" + CHART_NAME + ".selectorLabels" . | nindent 6 }}',
    },
    template: {
      metadata: {
        '{{- with .Values.podAnnotations }}: "aa",
        annotations: "{{- toYaml . | nindent 8 }}",
        '{{- end }}': "aa",
        labels:
          '{{- include "" + CHART_NAME + ".selectorLabels" . | nindent 8 }}',
      },
      spec: {
        '{{- with .Values.imagePullSecrets }}: "aa",
        imagePullSecrets: "{{- toYaml . | nindent 8 }}",
        '{{- end }}': "aa",
        serviceAccountName:
          '{{ include "" + CHART_NAME + ".serviceAccountName" . }}',
        securityContext:
          '{{- toYaml .Values.podSecurityContext | nindent 8 }}',
        containers: {},
        '{{- with .Values.nodeSelector }}: "aa",
        nodeSelector: "{{- toYaml . | nindent 8 }}",
        '{{- end }}': "aa",
        '{{- with .Values.affinity }}: "aa",
        affinity: "{{- toYaml . | nindent 8 }}",
        '{{- end }}': "aa",
        '{{- with .Values.tolerations }}: "aa",
        tolerations: "{{- toYaml . | nindent 8 }}",
        '{{- end }}': "aa",
      },
    },
  },
};

```

Figure 5.10 generic_deployment

```

generic_service = {
  apiVersion: "v1",
  kind: "Service",
  metadata: {
    name: '{{ include "" + CHART_NAME + '.fullname' . }}',
    labels: '{{- include "" + CHART_NAME + '.labels' . | nindent 4 }}',
  },
  spec: {
    type: "{{ .Values.service.type }}",
    externalTrafficPolicy: "{{ .Values.service.extTrafficPolicy }}",
    ports: {},
    selector:
      '{{- include "" + CHART_NAME + '.selectorLabels' . | nindent 4 }}',
  },
};

```

Figure 5.11 generic_service

```

foo/
├── .helmignore # Contains patterns to ignore when packaging Helm charts.
├── Chart.yaml # Information about your chart
├── values.yaml # The default values for your templates
├── charts/    # Charts that this chart depends on
├── templates/ # The template files
└── tests/    # The test files

```

Figure 5.12 Helm Chart Directories Example (*Helm Create*, 2022)

5.3 Prebuild Interfaces Page

At the prebuild interfaces page the user can select specific setups from a dropdown menu and then choose the service type and set the number of replicas and the honeypot log path (Figure 5.13). Prebuilt Interfaces were created in case a user does not know which honeypot services they need from it.

Siemens PLC ▾

Nodeport

LoadBalancer 58.2.0.0

Replicas 1

Logs path (optional) /siemensPLC-logs

Confirm

Figure 5.13 Prebuild Interface Options Example

Every available interface is created in the form of a JSON file. The file contains the name of the chart, the names of the honeypots that will be used, a default logs path, the services, the container ports, the protocol type for every service that is going to be used and a string that contains a description of the honeypot's services. Figure 5.14 shows the Siemens PLC JSON file that is used in this example. After the user has selected the prebuilt interface, the service type, the number of replicas and the log path they must click confirm to move to the next step of the process.

```
{
  "name": "siemens-hp-chart",
  "honeypots": {
    "names": ["conpot"],
    "conpot": {
      "volumes": ["/honeypot-logs/conpot"],
      "services": [{"ftp": 21}, {"modbus": 502}, {"s7comm": 102}, {"enip": 44818}],
      "containerports": [2121, 5020, 10201, 44818],
      "protocols": ["TCP", "TCP", "TCP", "TCP"]
    }
  },
  "infoServiceText": "Services:\nFTP\nModbus TCP\nS7Comm\nEtherNet/IP"
}
```

Figure 5.14 Siemens PLC JSON file

Prebuild Interfaces page follows the same principles as the Custom Honeypots page. First the program initializes the data structure shown in Figure 5.7. Afterwards it checks the ServiceType. If the LoadBalancer is selected then it validates the IP address. The program then fills the ServiceType and the IP address (if applicable) to the data structure. Next it validates the replica number and registers the number to the data structure. Finally, it fills the logs path in the data structure.

When the data structure is ready then the chart information gets displayed on the page (Figure 5.15). With the information panel the user can review their choices and then they can click Create to finish the chart creation. The chart creation process is the same as the custom build option.

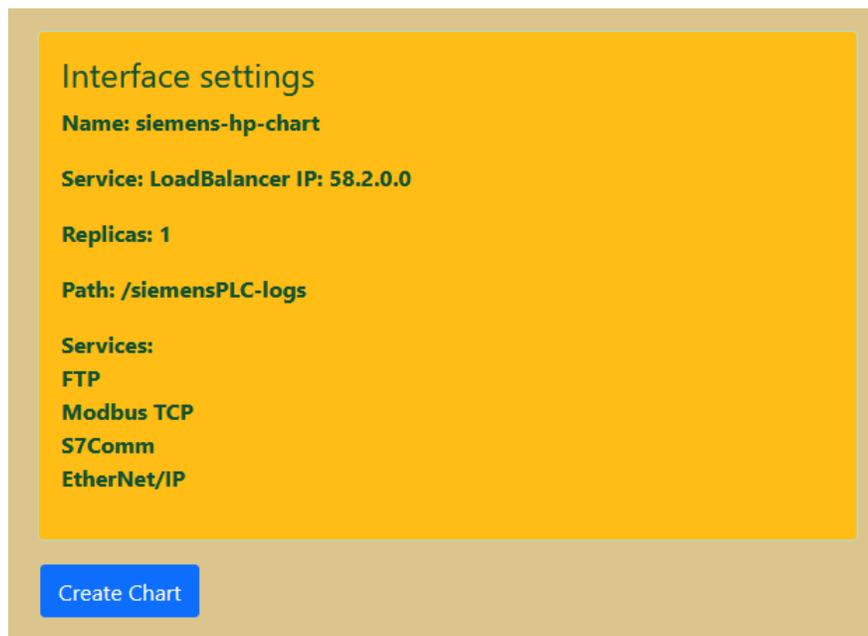


Figure 5.15 Prebuild Interfaces Information Panel

5.4 Deployment Example

In this example we are creating a chart for Conpot. Conpot, as mentioned previously ([Section 3.3, p. 30](#)), is an ICS/SCADA honeypot. Table 5.3 shows which ports are going to be exposed after deployment. These ports were selected for demonstration purposes. In case we wanted to simulate an industrial device like Siemens S7-300/400 PLC we would have enabled the following protocols (*Which Types of Connection/protocols Do the S7-300/400 CPUs and the CPs Support by Default?*, 2009).

- FTP
- S7Comm
- Modbus TCP
- EtherNetIP

The ServiceType in this example is LoadBalancer and we are deploying a single replica.

Port	Protocol
21/TCP	FTP
69/UDP	TFTP
102/TCP	S7Comm (iso-tsap)
161/UDP	SNMP
502/TCP	Modbus TCP (mbap)
623/UDP	IPMI (asf-rmcp)
44818/TCP	EtherNetIP (enip)
47808/UDP	BACnet

Table 5.3 Conpot Exposed Ports

After the creation of the chart the user can download it and they can deploy it on their Kubernetes cluster. The file is downloaded in ZIP format. After extraction, the chart can be deployed using the installation command of Helm: **helm install** (*Helm Install*, 2022). In this example the name of the chart is **conpot-chart**.

```
user@master-node:~$ helm install conpot-chart conpot-chart
```

After running the above command the Kubelet on the worker node will instruct Docker to download and run the Conpot's container image from the image registry. To check if the pod is running we can run the **kubectl get pods** command (*Kubectl Reference Docs*, 2022). In Figure 5.16 is shown the result.

```
user@master-node:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
conpot-chart-7b9f5db579-blwb4	1/1	Running	0	148m

Figure 5.16 Result of kubectl get pods

When the pod is ready we can run **kubectl get services** to display additional information about the running pods, such as ServiceType, Cluster-IP, External-IP, exposed ports, and the amount of time the pod is running. In Figure 5.17 is shown the result of the command. The external IP is blurred for security reasons.

```
user@master-node:~$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
conpot-chart	LoadBalancer	10.104.154.88	██████████	21:32198/TCP,69:31679/UDP,502:31630/TCP,161:30847/UDP,102:30661/TCP,47808:32703/UDP,623:30070/UDP,44818:30126/TCP	151m

Figure 5.17 Result of kubectl get services

Now that we know that the pod is running properly on the Kubernetes cluster, we can scan the external IP using the Nmap scanning tool. Nmap stands for Network Mapper and it is an open-source Linux command-line tool for scanning IP addresses and ports in a network (*Nmap*, 2022). To scan the exposed ports we used the following commands. Figure 5.18 depicts the summary of all the executed Nmap commands.

For TCP ports

```
user@pc:~$ nmap -sS -p <port number> <network IP address>
```

For UDP ports

```
user@pc:~$ nmap -sU -p <port number> <network IP address>
```

```

      PORT      STATE      SERVICE
      21/tcp    open      ftp
      69/udp    open|filtered tftp
      102/tcp   open      iso-tsap
      161/udp   open      snmp
      502/tcp   open      mbap
      623/udp   open      asf-rmcp
      44818/tcp open      EtherNetIP-2
      47808/udp open|filtered bacnet

```

Figure 5.18 Result of Nmap Commands

With the above results we confirmed that the exposed ports are indeed open to the outside world.

5.5 Log Management and Visualization

After deploying a honeypot container, with every interaction it logs information about it like

- Timestamp
- Source IP and port (Attacker)
- Destination IP and port (Target System)
- Information Message

These logs are stored in both the container and the worker node. The worker node logs remain stored regardless if the container stops working.

For the management and visualization of logs we used the ELK Stack. ELK is an acronym describing three open-source projects - Elasticsearch, Logstash, and Kibana. Afterwards Beats open platform was added to the Stack (*The ELK Stack: From the Creators of Elasticsearch*, 2022). We used the ELK Stack, because its components provide a complete solution for log

management and visualization. In our system the logs are harvested by running Filebeat on the worker node. Filebeat reads each log file line by line and sends the content to the master node. The output from Filebeat goes to Logstash, where the logs get converted, and filtered. Logstash also deciphers geographical coordinates from IP addresses, which is useful information for the presentation of statistics. To store the converted logs we used Elasticsearch and to visualize the data we used Kibana.

From the Kibana visualization we can extract useful information. For example, which port was attacked the most, and knowing that we can investigate the reason for the attack and the methods used by the attacker. After determining the type of attack, the administrators can take steps to close the hole that allowed the exploit to occur. Also the source location of the attack is useful, because this information shows whether the attack originated from the organization's network or somewhere else.

Figures 5.19 and 5.20 show two examples of Kibana visualization. Figure 5.19 shows how many times an exposed port has been contacted by an outside source. This information can help us identify which protocols are attacked the most. In our case FTP seems to be mostly attacked followed by S7comm. It is logical for the FTP to be attacked the most because a malicious actor can use FTP to extract files from our system, or send malicious software to it. As for the S7comm protocol, attackers can use it to extract information from the network's PLCs, or control them remotely to sabotage the system. By having this information, we can take measures to increase the security of these particular protocols on our real system.

Figure 5.20 shows the total number of interactions and which countries they originated from. This information is derived from the source IP address and can be used either for research purposes or for identifying insider attacks.

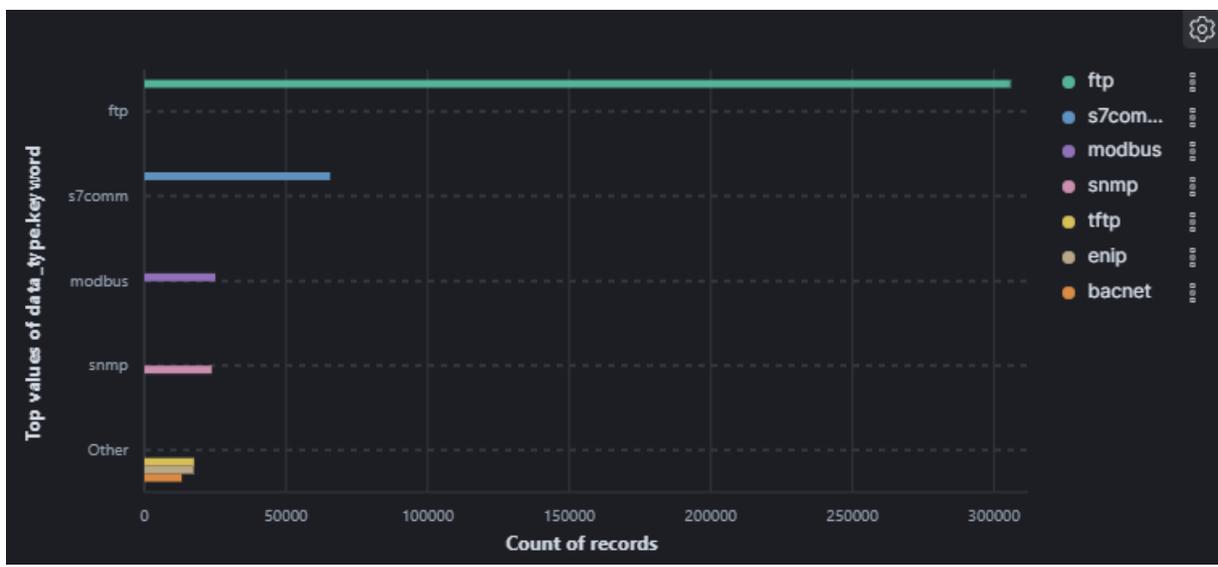


Figure 5.19 Number of Recorded Port Interactions

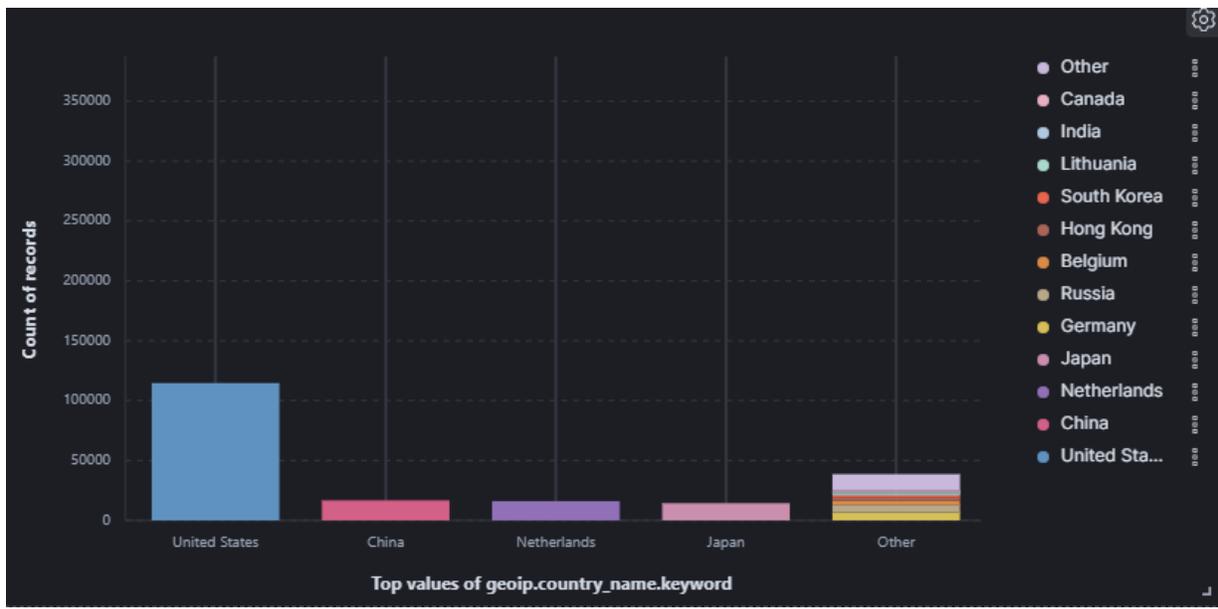


Figure 5.20 Number of Recorded Interactions Based on Geographical Coordinates

5.6 System Architecture

Now that the Honeychart solution has been described in the previous sections, the system architecture can be summarized. To deploy and test the available honeypots (Conpot, Dionaea and Cowrie) and to collect their logs, we used a Kubernetes cluster with a single worker node ([Section 4.4, p. 42](#)).

On the worker node is installed a load-balancer named MetalLB. MetalLB is an open-source load-balancer implementation for bare metal Kubernetes clusters (*MetalLB - GitHub, 2020*). This load balancer was needed because Kubernetes does not provide an implementation of network load balancers. Which means that Kubernetes itself cannot expose a service or an application to the outside world. MetalLB directs the network traffic to the honeypot container.

When an outside source comes in contact with the honeypot container, the honeypot logs this interaction in a directory inside the container and in a directory in the worker node file system. Then Filebeat ([Section 5.5, p. 62](#)) reads these logs and sends them to the master node.

On the master node is installed the rest of the ELK Stack ([Section 5.5, p. 61](#)). The output of Filebeat goes to Logstash and the output of Logstash gets stored in Elasticsearch. And finally Kibana visualizes the logs as shown in Figures 5.19 and 5.20 ([Section 5.5, p. 63](#)).

The system architecture, that is described above, is shown in Figure 5.21.

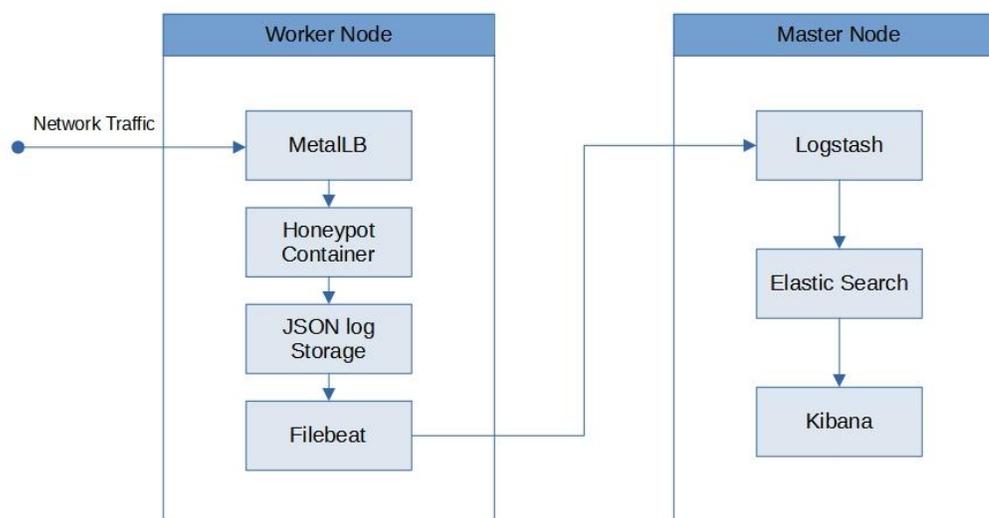


Figure 5.21 System Architecture

6 Conclusion

In this work is presented how honeypots can be used to protect ICS/SCADA networks and how they can be easily deployed using modern methods. By simulating the most frequently used communication protocols in industrial networks, ICS honeypots can protect these networks on different levels. They can be fully interactive with the attacker (high-interaction honeypots) because they contain a fully functional OS making the attacker believe that they are using the actual system. Or they can emulate certain services (low-interaction honeypots), without the complete functionalities of an OS.

Nowadays with container technology and Kubernetes, the deployment of applications is easier than before. All that is needed is a Kubernetes cluster, a containerized application, and a description file (container manifest). After deploying the application, Kubernetes takes care of the service discovery and load balancing. It also makes sure that the application is always running, by restarting the pod in case of a failure. And completely isolates running applications from each other. The last feature is very important, especially when a honeypot is deployed in the cluster.

With the HoneyChart solution ([Section 5, p. 45](#)) we achieved fast and automatic deployment of honeypots in a network. Giving the user the ability to choose which communication protocols to enable on a honeypot, they can build industrial device profiles so that the honeypot will behave like an actual factory network device and deceive possible attackers. And finally, using the logs that the honeypots provide, we can investigate the motives of the attacker and their methods and also whether the attack originated from inside the network or outside. With this information we can make our system more secure.

References

- About Node.js*. (2022, March 17). Node.js. <https://nodejs.org/en/about/>
- Aliyev, V. (2010). *Using honeypots to study skill level of attackers based on the exploited vulnerabilities in the network*. Chalmers University of Technology.
- Buza, D. I., Juhász, F., Miru, G., Félegyházi, M., & Holczer, T. (2014). CryPLH: Protecting smart energy systems from targeted attacks with a PLC honeypot. In *International Workshop on Smart Grid Security* (pp. 181-192). Springer, Cham.
- Conpot Documentation*. (2018). Retrieved July 05, 2021, from <https://conpot.readthedocs.io/en/latest/>
- cowrie/cowrie - Docker Image*. (2015). Docker Hub. <https://hub.docker.com/r/cowrie/cowrie>
- Dalamagkas, C., Sarigiannidis, P., Ioannidis, D., Iturbe, E., Nikolis, O., Ramos, F., Rios, E., Sarigiannidis, A., & Tzovaras, D. (2019). A survey on honeypots, honeynets and their applications on smart grid. In *2019 IEEE Conference on Network Softwarization (NetSoft)* (pp. 93-100). IEEE.
- dinotools/dionaea - Docker Image*. (n.d.). Docker Hub. <https://hub.docker.com/r/dinotools/dionaea>
- East, S., Butts, J., Papa, M., & Sheno, S. (2009). A Taxonomy of Attacks on the DNP3 Protocol. In *Critical Infrastructure Protection III* (pp. 67-81). Springer.
- The ELK Stack: From the Creators of Elasticsearch*. (2022, March 18). Elastic. <https://www.elastic.co/what-is/elk-stack>
- Helm Create*. (2022). Helm. https://helm.sh/docs/helm/helm_create/

- Helm Install*. (2022, March 18). Helm. https://helm.sh/docs/helm/helm_install/
- honeynet/conpot - Docker Image*. (n.d.). Docker Hub. <https://hub.docker.com/r/honeynet/conpot>
- IBM. (n.d.). *IPv4 and IPv6 address formats*. IBM.
<https://www.ibm.com/docs/en/ts3500-tape-library?topic=functionality-ipv4-ipv6-address-formats>
- Joshi, R., & Sardana, A. (2011). *Honeypots: A new paradigm to information security*. CRC Press.
- Kołtyś, K., & Gajewski, R. (2015). SHaPe: A honeypot for electric power substation. *Journal of Telecommunications and Information Technology*, 37-43.
- Kubectl Reference Docs*. (2022, March 16). Kubernetes.
<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get>
- Lahza, H., Radke, K., & Foo, E. (2018). Applying domain-specific knowledge to construct features for detecting distributed denial-of-service attacks on the GOOSE and MMS protocols. *International Journal of Critical Infrastructure Protection*, 20, 48-67.
- Lewis, L., & Peterson, D. (n.d.). SCADA Honeynets How to Build and Analyzing Attacks.
- López-Morales, E., Rubio-Medrano, C., Doupé, A., Shoshitaishvili, Y., Wang, R., Bao, T., & Ahn, G.-J. (2020). HoneyPLC: A next-generation honeypot for industrial control systems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (pp. 279-291).
- Luksa, M. (2017). *Kubernetes in Action*. Simon and Schuster.
- MacKenzie, D., & Meyering, J. (2007, July 14). *chmod(1): change file mode bits - Linux man page*. Linux Documentation. <https://linux.die.net/man/1/chmod>

- Mashima, D., Chen, B., Gunathilaka, P., & Tjiong, E. L. (2017). Towards a grid-wide, high-fidelity electrical substation honeynet. In *2017 IEEE International Conference on Smart Grid Communications (SmartGridComm)* (pp. 89-95). IEEE.
- MetallB - GitHub*. (2020, August 4). MetallB. <https://metallb.org/>
- Metongnon, L., & Sadre, R. (2018). Beyond telnet: Prevalence of iot protocols in telescope and honeypot measurements. In *Proceedings of the 2018 Workshop on Traffic Measurements for Cybersecurity* (pp. 21-26).
- Musilová, S. (2020). *Profiling and detection of IoT attacks in Telnet traffic* [Doctoral dissertation, Master's Thesis]. Czech Technical University in Prague, Department of Computer Science.
- Nmap*. (2022, March 16). Nmap: the Network Mapper - Free Security Scanner. <https://nmap.org/>
- Paolinelli, F. (2020, September 12). *metallb*. GitHub. <https://github.com/metallb/metallb>
- Peter, E., & Schiller, T. (2011). A practical guide to honeypots. *Washington Univerity*. <https://www.cse.wustl.edu/~jain/cse571-09/ftp/honey/index.html>
- Port (computer networking)*. (n.d.). Wikipedia. [https://en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking))
- POST - HTTP | MDN*. (2021, August 13). MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST>
- Redwood, O., Lawrence, J., & Burmester, M. (2015). A symbolic honeynet framework for scada system threat intelligence. In *International Conference on Critical Infrastructure Protection* (pp. 103-118). Springer, Cham.
- Samanis, E. (2018). *Protection of SCADA Industrial Control Systems using a flexible open source architectural solution*.

- Scott, C., & Carbone, R. (2014). Designing and Implementing a Honeypot for a SCADA Network. *SANS Institute Reading Room*, 39.
- Serbanescu, A. V., Obermeier, S., & Yu, D.-Y. (2015). ICS threat analysis using a large-scale honeynet. In *3rd International Symposium for ICS & SCADA Cyber Security Research 2015 (ICS-CSR 2015)* 3 (pp. 20-30).
- Service*. (2022, February 3). Kubernetes.
<https://kubernetes.io/docs/concepts/services-networking/service/>
- Shape: A honeypot for electric power substation. (2015). *Journal of Telecommunications and Information Technology*, 37-43.
- The SHaPe project website*. (n.d.).
<https://app.assembla.com/spaces/scada-honeypot/subversion/source>
- Stallings, W., Brown, L., Bauer, M. D., & Bhattacharjee, A. K. (2018). *Computer security: principles and practice* (4th ed.). Pearson Education Upper Saddle River, NJ, USA.
- Tamboli, S., Rawale, M., Thoraiet, R., & Agashe, S. (2015). Implementation of Modbus RTU and Modbus TCP communication using Siemens S7-1200 PLC for batch process. In *2015 international conference on smart technologies and management for computing, communication, controls, energy and materials (ICSTM)* (pp. 258-263). IEEE.
- What is Kubernetes?* (2021, July 23). Kubernetes.
<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
- Which types of connection/protocols do the S7-300/400 CPUs and the CPs support by default?* (2009, December 11).
<https://support.industry.siemens.com/cs/document/24352751/which-types-of-connection-protocols-do-the-s7-300-400-cpus-and-the-cps-support-by-default-?dti=0&lc=en-FI>

Xiao, F., Chen, E., & Xu, Q. (2017). S7commtrace: A high interactive honeypot for industrial control system based on s7 protocol. In *International Conference on Information and Communications Security* (pp. 412-423). Springer.

Yang, Y., McLaughlin, K., Littler, T., Sezer, S., Pranggono, B., & Wang, H. (2013). Intrusion detection system for IEC 60870-5-104 based SCADA networks. In *2013 IEEE power & energy society general meeting* (pp. 1-5). IEEE.

Yau, K., Chow, K.-P., & Yiu, S.-M. (2018). A forensic logging system for siemens programmable logic controllers. In *IFIP International Conference on Digital Forensics* (pp. 331-349). Springer.

Yoo, H., & Shon, T. (2015). Novel Approach for Detecting Network Anomalies for Substation Automation based on IEC 61850. *Multimedia Tools and Applications*, 74(1), 303-318.

zip(1): package/compress files - Linux man page. (2007, July 11). Linux Documentation.

<https://linux.die.net/man/1/zip>