Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Alakbar Mammadov

Building a prototype of web API honeypot for Electric Vehicle Charging Network operators

Master's thesis in Information Security
Supervisor: Prof. Dr. Bernhard Hämmerli
Co-supervisor: Øyvind Anders Arntzen Toftegaard
December 2022

Alakbar Mammadov

Building a prototype of web API honeypot for Electric Vehicle Charging Network operators

Master's thesis in Information Security
Supervisor: Prof. Dr. Bernhard Hämmerli
Co-supervisor: Øyvind Anders Arntzen Toftegaard
December 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Building a prototype of web API honeypot for Electric Vehicle Charging Network operators

Alakbar Mammadov

Abstract

Critical infrastructure is one of the main targets for malicious actors. Due to the crucial dependencies of public services on electric power, all components of the power generation and distribution process are of great interest to adversaries. Electric power distribution infrastructure keeps developing and offering new services to its users. The development process involves the addition of new automation mechanisms, software, and hardware components. These developments come at the cost of the increased attack surface and new security challenges. Electric Vehicle charging networks are not an exception. Their nature imposes a highly automated operations model with minimum human supervision. Application Programming Interfaces are often used to control and monitor electric charging systems. It is a software intermediary used for communication between decoupled computer programs. An Application Programming Interface may become subject to cyberattacks and proper threat intelligence is therefore important. This work develops a prototype of a web Application Programming Interface honeypot, mimicking an operational Electric Vehicle Charging Network management system. The study proposes a proof of concept prototype, deployed in the Amazon Web Services cloud, using automation tools. The contribution of the research is an affordable and scalable web Application Programming Interface Honeypot system, that can be used by small and middle-size Electric Vehicle Charging Network operators to collect valuable threat intelligence. The honeypot is a first step towards understanding the attack patterns against charging networks in order to build effective defense mechanisms.

Acknowledgments

I would like to thank my supervisors Prof. Dr. Bernhard Hämmerli (Department of Information Security and Communication Technology at NTNU) and Øyvind Anders Arntzen Toftegaard (Senior Adviser for The Norwegian Energy Regulatory Authority) for their support and assistance. Their deep knowledge and professionalism were the most important contributing factors to my work. Continuous support during evening hours and weekends made their contribution exceptional and inspiring.

I would like to thank Dr. Sebastian Obermeier and his team (penetration tester "Team 1") of Lucerne University of Applied Sciences and Arts for their assistance in a simulation of a cyberattack. Also, I would like to thank security professional Orkhan Yolchuyev (penetration tester "Team 2") for his assistance in testing the solution.

Finally, I would like to express my infinite gratitude to my wife Rosanna, and son Aslan for their love, patience, and understanding during all of my studies.

Alakbar Mammadov, Norway, December 2022

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

The importance of digital operations continues to increase with their advance to traditionally offline or hybrid aspects of life. Services and utilities used in daily life extended their reach to customers by adding an online interface to their businesses. Nowadays it is difficult to imagine any successful business having no presence online. Fully automated service became an operational necessity for many sectors of modern business. Transportation as one of the crucial services utilized daily is also influenced by this trend. A new generation of vehicles fully running on electric energy introduced new routines due to the way they top up their "fuel tanks".

Once an attractive concept and exotics rather than a practical tool, electric vehicles (EVs) expanded dramatically, entering world markets and forcing manufacturers to rethink their vision of the future of mobility. In 2021 EV sales reached almost 10 percent of global car sales. This rapid growth implies quick actions to build a sufficient infrastructure for EV charging. Although they existed before, EV charging solutions have never made it to nationwide or even regional networks until the 21st century. In order to work their way to the mass market, challenges like business justification and cost-effectiveness of EVs had to be answered and paired with green initiatives advocating for lowering CO2 emissions. This has never become the case before the last decade. Today humanity is witnessing a different picture. Electric vehicles made their way to the mass market of passenger cars and now are on the way up on the market of public transport and trucks. These developments imply a new burst in the number of EVs on the roads and growing demand for EV charging infrastructure. According to the last report by the International Energy Agency [1], in 2021 the number of EVs worldwide increased to 16.5 million. That is three times more than in 2018. Figure 1.1 depicts a global electric car stock.

Tesla became one of the pioneering manufacturers in the world, offering EVs and public charging solutions in one package from a single vendor. The rest of the car manufacturers took rather a different approach, focusing on car production and delegating the EV charging infrastructure development to other market players. Also, each region in the World decided to introduce its own standards for the

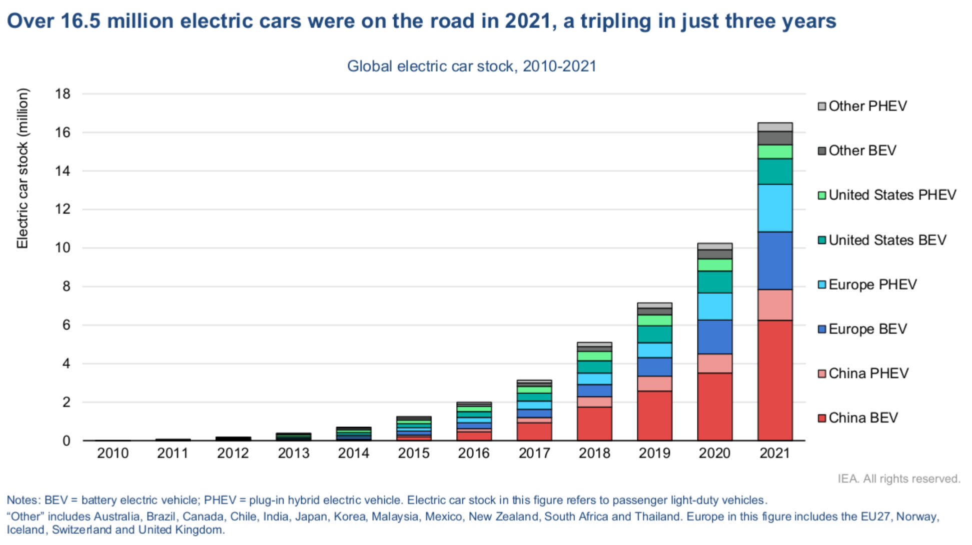


Figure 1.1: Global electric car stock[1]

charging interfaces. That led to some differences in physical charging interfaces for cars produced in different regions.

During the last decade, global EV manufacturers acknowledged the counter-productiveness of having different charging standards. To ensure functional and operational safety and efficiency, they started drifting towards unified standards, covering car batteries, onboard charging mechanisms, electric power specifications, and physical interfaces. Although the absolute consensus on charging standards and interfaces is yet to be achieved worldwide, some patterns and tendencies to unification can be witnessed based on the geography of manufacturing. Automakers strive to achieve common charging system architecture by implementing international standards and security requirements. European Union Agency For Network and Information Security published "Mapping of OES security requirements to specific sectors" in 2017 [2]. The document is intended for operators of essential services and provides a mapping of security requirements to sector-specific information security standards.

1.1 Problem description

Scientific efforts in the area of EV charging opened new perspectives, offering business opportunities. EVs previously seen as a challenge to power grids can become a stabilizing resource with the implementation of well-thought smart technologies [3]. Adding more intelligence into the power grid and peripheral components comes at the cost of increased demand for computational capabilities of electricity networks and electric cars. The level of integration necessary for these types of operations introduces new security risks and extends a potential attack surface of the critical infrastructure. At the same time, the growing demand for mobility and the existence of regional and global charging providers imply further integration of various charging, billing, banking, and identity providers, offering new business opportunities and technological advances.

The complexity of modern digital solutions and heavy reliance on them due to the necessity to maintain a highly automated operational model implies a secure and reliable way of interconnection and data exchange between various players, representing different industries. As in many other modern businesses, the unified means of digital integration and communication widely used in EV charging networks include the Application Programming Interface (API) component [4], linking remote endpoints and multi-vendor products in one single solution.

Due to the nature of the services provided, EV charging network (EVCN) operations imply software integration of the EVCN management system with remote endpoints, billing systems, identity providers, and other third-party organizations. APIs help to solve yet another challenge of integrating regional EVCNs into the infrastructures of global EV charging providers. Global EVCNs do not own every and each charger they operate. They provide umbrella services for vehicles traveling internationally. For example, a Norwegian EV driver does not need a user account for every EVCN he is going to use on his way from Norway to Italy. Global EVCN operators use API integration and provide single sign-on and single-window billing services.

API is comprised of protocols and definitions helping to integrate software applications. It is an interface accepting requests from its users and acting according to requests received. Users do not need to know the internals of the software behind the API they query. For example, the EVCN back-end management system can communicate with charger endpoints using (OCPP) protocol [5]. But applications using the back-end management system as an intermediary do not need to be OCPP-complaint and can communicate to an API using serialized sets of instructions. The role of an API, in this case, is to receive requests and process them according to the API specifications, in other words, "translate" them to OCPP-compliant commands.

APIs follow a generic design that is specific to its type and operate based on the pre-agreed sets of commands. Web APIs use a well-known and mature network protocol Hypertext Transfer Protocol (HTTP) [6] as a transport. A consumer has to be familiar with HTTP request and response mechanisms and the resource structure of the API and this knowledge is enough for successful communication. APIs differ by the level of their exposure to the open Internet. They can be open to the public or partners, intended for internal use, or composite.

The fact that APIs became a central point of data exchange between various mechanisms used in an EVCN also turned them into attractive targets for malicious actors. Thus, understanding of techniques used by attackers is important for building effective defense mechanisms.

The lack of threat intelligence data is a serious problem, negatively influencing digital defense strategy. Collecting threat intelligence is not an obvious task that can be solved by implementing generic solutions. Any given digital infrastructure has its own specifics and serves the core business. EV charging networks can be seriously disrupted or damaged by successful attacks against their APIs. This work is addressing the problem of collecting threat intelligence using a decoy API, pretending to be a part of operational EVCN.

1.2 Motivation

Increased usage and high reliance on API solutions may attract more attention from malicious actors. Depending on the level of exposure of an API to external resources on the Internet, the risk of being penetrated by an adversary remains quite high. The nature of API technology dictates that it is an interface with enabled remote access for various partners, internal and external to the organization. Along with security mechanisms implemented to protect access, it is important to have additional controls, strengthening the overall security level. To maintain uninterruptible and secure operations and win customers it is necessary to build an effective defense strategy based on a solid understanding of potential security threats. Although it is not possible to predict the time and means of attacks against digital infrastructure, continuously collected and analyzed threat intelligence data can help to mitigate the effects of security breaches or even prevent them.

Cyberthreats can take many different forms and shapes. Sometimes it is possible to disrupt digital operations using very obvious and well-known technologies, not intended at all for malicious activities. Brokenwire is one example of such an attack, capable of interrupting control communication between EV and charger, and causing charging session to stop [7]. According to the team of security professionals who discovered the vulnerability, the attack requires off-the-shelf radio hardware and minimal technical knowledge. In other cases, the sophistication level of attacks is well beyond the level of technical expertise and capabilities of an average cybercriminal. The attack against a power grid in eastern Ukraine is an example of well prepared and thoroughly executed operation, causing a massive power outage for hundreds of thousands of civilians. [8]

The motivation for building the web API honeypot is the attempt to develop a better understanding of adversaries and their means of operations. The data collected using honeypot systems and analyzed afterward is important for minimizing negative impact in case of security incidents or breaches. Honeypot systems mimic the behavior of real IT systems, attracting potential intruders to attack them. Information about techniques and mechanisms used by offenders believing that they attack a real digital system is priceless for security professionals investigating them.

This work presents a decoy web API, mimicking one of the web APIs used in EVCN management systems. Specifically, the presented API is an imposter for the production Application Programming Interface managing endpoint EV charger devices. Web API honeypot collects threat intelligence data about attacks conducted against a fictitious organization providing EV charging services. There is not much publicly available information about research or investigations conducted by real EVCN operators at the moment. Probably, businesses conducting similar experimentation and research prefer to keep the results for themselves rather than share the information with the general public and competitors.

1.3 Scope of the work

The scope of the work consists of three main parts. The first part provides an insight into the problem by investigating previous work, and special publications, and conducting an interview with the technical staff of one of the major EVCN operators in Scandinavia. Previous work helps to gather necessary information about existing research and developments, identifying missing parts. Special publications contribute to formulating an understanding of the current standards and regulations. The interview aims to provide insight into the way EVCNs are currently organized and their usage of APIs. It helps to understand how the data flow within EVCN digital infrastructure is organized.

The second part of the work is dedicated to designing and actual deployment of the prototype of the web API honeypot solution in the AWS cloud. Some of the AWS solutions, such as serverless Lambda functions ensure cost-effective architecture, utilizing resources on-demand, instead of allocating them permanently. There is no need for specialized hardware or software to deploy the prototype. It is enough to use a Free Tier 1-year AWS subscription in order to deploy the solution and have it up and running for testing purposes.

Depending on the traffic and load, the AWS resource utilization will change and has to be monitored and tweaked by the user himself. It is up to the users of the API honeypot how they organize their databases and the level of data exposure to the API. API resources, means of authentication, HTTP methods implemented and other infrastructure-specific details are to be adapted for each specific scenario the API honeypot is going to be used for. Although an example API honeypot setup is presented in Chapter 4 "Prototyping", the main goal of this work is to provide a general model or a prototype of the web API honeypot, rather than a full-scale solution.

The final part of the scope is to show one of many options to process the data collected through logs and organize them in a comprehensive way, enabling users to analyze the collected data. Logs will be collected using the AWS CloudWatch service and parsed using a custom application written for this study in Python programming language. The application will output the parsed data in JavaScript Object Notation (JSON) [9] format and write it to a local file and a database. JSON is widely accepted as a common data format and data represented in JSON can be used in almost any modern data analysis and visualization tool. This work will use the InfluxDB database to store the sample data and the Grafana visualization front-end tool to visualize it.

1.4 Research questions

To guide the research and organize it into an effective structure, the following research questions are formulated.

1.4.1 Question 1

Is it advisable to build a web API honeypot for EV charging networks?

1.4.2 Question 2

How to build a web API honeypot solution in the AWS cloud with an on-demand resource utilization design?

1.4.3 Question 3

To what degree the deployment of the web API honeypot can be automated?

1.5 Claimed contribution

Building an API from the systems development point of view is not a dramatically difficult task for big companies with enough technical resources. But for middle-size and especially small EVCNs, it is not as trivial as it looks. Small and middle-size EVCNs might not have enough hardware resources and sufficient staffing to build, operate and support an on-premise web API honeypot solution from scratch. API can be built in a variety of ways, including the provisioning of server instances and other components. That will imply unavoidable investment in software, hardware, and a certain waste of resources when the solution is idling or not in use. Also, it will require a certain level of expertise to set up and maintain underlying services on the systems and network level. The main contribution of this work is a web API honeypot solution built using cloud automation tools and mechanisms, reducing setup and maintenance costs. There is no need for on-premise hardware infrastructure, software licenses, or trained personnel to maintain the underlying services, such as systems and networks, in order to spin up and run the web API honeypot.

Another contribution of this work is a proposed model for security log processing and extraction of the data of interest. Having a decoy API alone, producing tens or hundreds of thousands of security logs daily is not of much help for security engineers. The manual processing and extraction of this amount of data within a reasonable timeframe are not possible. This work proposes a model of how to do it and how to organize and visualize extracted data.

This work will assist small and middle-size EVCN operators to build and operate their own honeypot operation in the AWS cloud within minutes. The cost of the solution will be negligible compared to the benefits it brings. The data collected will help EVCN operators to understand their adversary, and identify the source IP addresses, methods, techniques, level of technical sophistication, and intensity of the attacks. The overall value of the research is important because deploying and running a web API honeypot is a first step towards securing the EVCN network.

This chapter described the problem and explained the motivation for the project. Also, it set research questions and described the scope of the work. The next chapter will provide detailed information on the literature review, special publications, and the interview conducted on the topic.

Chapter 2

State of the art

This chapter provides information on the background and related work in the area. The first section reviews special publications, concerning API security and standards. APIs are common communication mechanisms adopted worldwide. Certain standards and regulations are of great importance to maintain a minimum required baseline in terms of API security. The second section reviews previous work on web APIs and honeypot systems. API as a technology went through various transformations in the last decade and there is a significant number of publications on the topic. The literature of interest is identified by the specific topics. The literature review sheds light on honeypot systems as an important tool for collecting threat intelligence data. The third section provides highlights from the interview conducted with the team of technical engineers working for one of the major EVCN operators in Scandinavia. The last subsection concludes the chapter and shows how the state of the art is related to the project.

2.1 Special Publications

The Confidentiality, Integrity, and Availability triad (CIA) is a fundamental principle of cybersecurity widely used since the 1980s. The National Institute of Standards and Technology (NIST) in its publication "Framework for Improving Critical Infrastructure Cybersecurity" [10] proposes a set of activities based on the CIA triad model to achieve a specific security outcome. Core framework functions are the following ones - Identify, Protect, Detect, Respond and Recover. Each function has categories and subcategories, providing detailed insight into specific technical or management activities. The Detect function of the CIA triad includes categories such as Anomalies and Events, Security Continuous Monitoring, and Detection Processes.

ISO/IEC 27005:2022 standard provides guidance on managing information security risks[11]. It assists organizations to perform a security risk assessment and treatment activities. The information security risk management process consists of two main cycles - strategic and operational. On the strategic level the assets, sources of risk, and threats are considered in the organization-wide context. The operational cycle relies on the inputs from the strategic level and focuses on risk assessment review and updates. To evaluate the risks, the criteria of risk acceptance have to be determined, considering several influencing factors. Operational activities, processes, and supplier relationships are among those factors. Considering the nature of APIs widely used for data communication over the Internet, their usage influences the factors mentioned and consequentially the criteria for overall risk assessment. Organizations need to analyze and understand the level of exposure to certain risks in order to establish criteria for risk acceptance. Assessing the likelihood of risk scenarios and events is a crucial activity of the risk assessment process. Threat intelligence could be one of the key factors contributing to determining the likelihood criteria and overall risk assessment.

Another publication by the NIST is "Security Strategies for Microservices-based Application Systems: SP 800-204" [12]. The publication covers APIs and API gateways due to the increasing usage of APIs in a microservices architecture. API gateways are described as the entry points for microservice applications, helping to perform protocol translations, route incoming requests to the relevant downstream services and serve as the means of access to the back-end services. Since the crucial role API gateways play in modern infrastructures, NIST recommends equipping them with adequate infrastructure services such as authentication, access control, service monitoring, attack detection and response, security logging, and other necessary services. Security logging and monitoring data are emphasized as a source of crucial security data, required for organizing adequate defense against potential attacks. The publication suggests that security monitoring should be implemented on the gateway and application levels to detect unexpected behavior. Input validation failures and attempts to feed unexpected parameters to API are obvious signs of injection attacks and should be monitored for.

The "API technical and data standards" guidance by Central Digital and Data Office UK provides general recommendations on designing, building, and operating APIs [13]. The publication provides recommendations on response data formats, naming best practices, performance considerations, and other practical points. Also, it emphasizes the importance of designing and hosting API securely. Data and application-level security, auditing, and monitoring are noted as important components in API security.

"API security project" by The Open Web Application Security Project (OWASP) focuses on strategies to defend against API security risks [14]. Among various design issues found in modern APIs, the most notorious ones are related to broken object-level authorization, broken user authentication, excessive data exposure, lack of resource control, broken function-level authorization, various types of security misconfiguration, and insufficient logging and monitoring. Logging of access data is an obvious security precaution. Access logs are a relatively easily collected type of data, contributing to cybersecurity risk analysis. It is expected that any production API has a sufficient level of security logging enabled.

The UK National Cyber Security Centre provides general guidance on the cybersecurity design principles for critical national infrastructure [15]. Although the guidance is not specific to API security, it covers 5 main principles, applicable in order to design secure cybersystems. A deep understanding of a system design is one of the main principles. It is followed by other recommendations such as making compromise difficult, minimizing potential disruption, increasing chances of quick detection of compromise, and reducing the impact of a compromise.

2.2 Related work

The next few sections will review the existing work on the topic.

2.2.1 Honeypot systems

Biedermann et al (2012) [16] conducted research on honeypot systems in the cloud computing. The design proposes a solution for users of cloud computing services, offering Infrastructure-as-a-Service (IaaS). When an attack against the production system is detected by the honeypot controller system, the dynamic honeypot architecture clones it into a new honeypot virtual machine (VM), excluding sensitive data. The attack is redirected from the production system to the honeypot VM for further analysis without risking the production data. To avoid interruption of the attack while creating a clone of the production system, the attack is delayed and seamlessly transitioned to the honeypot VM after it is provisioned. To save time and avoid suspicion, the honeypot VM is not booted from scratch but live cloned instead, copying RAM. The file system is created from the snapshot of the target VM storage, with the subsequent step of removal of the sensitive data.

The whole process of cloning and diverging the attacker to a honeypot node takes a few seconds. The attacks are detected using events, triggering the honeypot controller to action - the number of new connections from the same source, and content of the incoming and outgoing payload. The number of rules can be extended and shaped for each infrastructure the dynamic honeypot solution is used for. The authors explain that their goal is to prove the feasibility of the approach they offer, not the actual data collection. After the all necessary information about the attack is collected and extracted, the attacker is banned from the network and the honeypot VM is terminated. The research emphasizes the importance of honeypot systems and how they can help to learn from attacks and discover vulnerabilities and configuration flows.

Ryandy et al (2020) [17] research categories of threat information collected from honeypots. In their attempt to analyze security threats, the authors investigate network traffic and payload artifacts. The network traffic analysis is important to understand the adversary's intentions and behaviors. The details such as domain name, IP address, and their relationship can help with revealing historical information about them and calculation of reputation scoring. The correlation analysis of such details is one of the key tasks for cybersecurity teams. Payload artifacts are responsible for executing a harmful activity to inflict damage on the target. To investigate the payload, it is often necessary to use a sandbox or isolated environment where malicious code can be safely executed and observed. The information gathered from network traffic and payload analysis is attributed to threat intelligence.

Threat intelligence paired with contextualized organizational threat analysis represents valuable information, raising security awareness and contributing to a better defense strategy. The research proposes a threat research framework XT-Pot, consisting of four major stages - data collection, data processing, analysis, and evaluation. The experiments conducted show that attackers spent most of their time trying to log in and doing a network-level reconnaissance. Another interesting observation is that the number of attacks is peaking during office hours on workdays. Table 2.1 shows the table of threat categories compiled during the experiment.

Table 2.1: Threat Categorization and Frequency Statistics[17]

Threat Category	Total	Threat Category	Total
Bruteforce	397719	Download tools	667
Profiling hardware	13072	Covering track	371
Profiling system	10054	Removing previously used tools	314
Profiling Linux tools	4471	Security bypass	56
Execution of tools	3291	Setup/Modify env PATH	40
Profiling file system	2742	Leaving mark/Narcissism	28
Enumerating task/process	2615	Copy of file	25
Enumeration login user	2614	Linux tool execution	15
SSH account config	2614	Setup persistence on boot	8
Privilege modification	2614	Silent run of tools	4

The research concludes with the importance of threat intelligence for the identification and categorization of security threats.

Ng et al (2018) [18] provide an overview of specialized honeypot applications. Web server-based and client-based honeypots, worm detection, bot detection, honeytoken concept, anti-phishing honeypot, insider detection honeypot, and advanced persistent threat honeypot systems are explained in detail. Cheh et al (2021) [19] analyzed OpenAPI specifications for security design issues. The authors acknowledge the complexity of modern web APIs and the challenges for security analysis. They propose a semi-automatic approach for security analysis and modeling of the OpenAPI specification. Another publication by Diaz-Rojas et al (2021) [20] states that the majority of the reported threats against web APIs are related to network traffic. The study proposes a wide variety of techniques and methods that can be implemented on the design level to defend against known web API threats.

2.2.2 Web API

Sohan et al (2015) [21] conducted a case study on the evolution of web API technologies. The authors acknowledge the importance of web API as a crucial interconnectivity mechanism, providing a cost-effective way of communication between applications. This is achieved at the cost of building unavoidable dependencies between interconnecting parties. Issues like backward compatibility, interoperability challenges to evolving web APIs, and scrupulous documentation can easily become a problem, if not addressed.

Wittern et al (2017) [22] conducted research on web API consumption and challenges related to making calls to web APIs. The work acknowledges an increase in the usage of web APIs and their significance in invoking third-party code. The work focuses on two main research threads - web API specification curation and static analysis of web API calls. In traditional APIs, the calling side downloads the local library and has great control over the code called. Web APIs do not use specific libraries. Instead, they use generic web technologies such as HTTP for transport and XML or JSON formats for data serialization. It is not known to a client whether her call has a correct signature including request payload and parameters until the actual call has been executed. Also, there are synchronization issues of a web API call in a Software Development Kit (SDK) used with the actual call. Finally, taking into account the remote nature of web services, all kinds of quality of service issues are of concern.

Web APIs bring a significant level of flexibility for service integration and relieve clients from the mandatory usage of certain software libraries. At the same time, web APIs require well-documented instructions to create specific signatures. Web APIs do not provide out-of-the-box instructions to the clients, describing their interfaces. OpenAPI specification can be a part of the solution for automatically creating, maintaining, and testing web API specifications. Otherwise, developers on the client side need to have documentation, describing the structure of a web API. The research emphasizes a problem in web APIs, lacking traditional compile-time error checking. The solution proposed is a static checker, extracting the content of requests and analyzing them for consistency with published web API specifications.

Tello-Rodriguez et al (2020) [23] proposed a design guide for building web APIs. The authors emphasize the importance of the usability of a web API in order to ensure a positive developer experience, working on client applications consuming the API. ISO/IEC 25010 is taken as a base to define usability, comprising effectiveness, efficiency, and satisfaction. The important characteristics of usability are ease of use, intuitiveness, and less need for documentation browsing.

Wilde (2018) [24] published a work on web APIs, discussing the difference between the usage of web resources by humans and the utilization of web APIs. While humans interact with the web using client software such as internet browsers, the APIs are utilized using programmatic tools and specific data structures such as Extensible Markup Language (XML) [25], JSON, or Resource Description Framework (RDF) [26]. Web APIs utilizing HTTP as a transport protocol can use a variety of representation languages, data structures, and HTTP tools to interact with clients. Following predefined specifications to expose their services, web APIs relieve their clients from the need to know their internals. Knowledge of the standards and resource design of a specific web API is sufficient for its successful utilization.

2.2.3 Web honeypot systems

In his article on low-interaction honeypot systems, Watson (2015) [27] notes that these systems evolved from very basic emulators to capable solutions, providing valuable threat intelligence information. Low-interaction honeypots proved to be useful in a number of scenarios, where cost and scalability are of importance. Other factors, such as reduced operational risk, liability, and level of exposure are just a few of many ones making low-interaction honeypot systems great and useful tools.

Musch et al (2018) [28] conducted research on the automatic generation of low-interaction web application honeypots (LIHPs). Although LIHP is a known technique to mimic the behavior of a real web application, it is easily recognizable by potential attackers using fingerprinting. The fingerprinting technique is used to identify the type and version of a web server. The research proposes an automatic honeypot generator called Chameleon. Chameleon communicates to real web systems via network crawling of the public HTTP interface. Based on the response, it builds response templates, imitating a web application.

The main advantage of the proposed system is to quickly build many LIHPs with various web applications or different versions of the same application. It provides an automated and highly scalable environment with as many LIHP instances as required. Due to the lack of actual service behind response templates, the computational resources involved in building a large number of imposter responses by Chameleon are negligible. The concept focuses on the pre-authentication surface of the mimicked systems, trying to understand the types of systems chosen for attacks and actual means of gaining access.

Chameleon divides operations into three phases. First, it probes a web resource of interest with a web crawler and extracts the data. In the next phase, it parses the extracted data, distinguishing static content from dynamic and processing them differently. At the end of this phase response templates are generated, replacing dynamic content with placeholder Chameleon's syntax. The last phase is publishing when the system responds to the requests of attackers. Choosing a matching template for response in the publishing phase is the most challenging task because requests can be malformed or no suitable template found among generated templates. To address the challenge, parts of HTTP requests were analyzed by their significance with the following ranking list being produced in the order of importance:

- HTTP method
- Path of the URL

- HTTP body(if PUT/POST) or query of the URL(otherwise)
- HTTP headers

The research is concluded with an average time of 18 minutes required for the creation of an instance of honeypot and approximately 256 templates generated per Content Management System (CMS). The main problem of a honeypot being recognized by using fingerprinting technique proved to be solved, due to the similarity of templates generated with the real CMS systems.

Rist et al (2010) [29] conducted experimentation on a low-interaction web honeypot Glastopf. The project proposes a web honeypot, emulating the behavior of a web server in a way that is expected by intruders. One example is a Remote File Inclusion vulnerability emulation. A response provided from a web server to the attacker is sufficient to be recognized as valid. Another example is RFI Bot, where a malicious file is executed on the web server, making the bot connect to an Internet Relay Chat IRC network and listen to commands from the bot master. Connecting many web servers in a botnet using this technique will cause an orchestrated attack or Distributed Denial of Service (DDoS) attack against the victims chosen.

The project also considers Local File Inclusion attacks by executing previously injected code or obtaining security-critical information by running system files. To attract potential attackers, the project relies on search engines and web crawlers used by them. The solution is comprised of several sensors, linked via a Python application called Central Database Daemon. The central database collects data such as IP addresses, helping to identify and block attackers, and actual requests sent via the web, revealing attack vectors and techniques. Glastopf honeypot is equipped with a web interface, that visualizes statistics about attackers.

Chiapponi et al (2020) [30] present a platform to mimic an airline website, luring Advanced Persistent Bots (APBs) and feeding them fake information. The motivation behind the project is to combat the web crawling of airline web applications by third parties. The research aims to build a mock web application emulating the behavior of a real airline web application, thus wasting resources and investment of parties misusing it. The project acknowledges two main problems and tries to address them. The first problem is the continuous improvement of bots and the ability of bot operators to modify them very rapidly. The second problem is that data collected by bots is verified. In case of discovering fake information is fed to their bots, operators immediately remove them as disclosed bots.

The experimentation part of the project involves a lot of tweaking to avoid the detection of the honeypot by bots. The data fed to bots were altered randomly and the research aimed to understand if these changes were recognized as something abnormal by bot operators. The behavioral analysis showed that the payloads of bot requests have a lot in common, such as the information about return flight tickets, and the time window between request and departure. Also, it was discovered that APBs crawling the web content send one or at most two requests per day in order to avoid detection by anti-bot systems. The research concludes that almost a third of the IP addresses were reused over time, probably indicating that they were used by the same bot operator.

Idris et al (2021) [31] proposed a learning environment for API security called Vulnerable Academic Information System (VAIS) based on the OWASP API Security Risks. The authors designed a vulnerable web application or a sandbox to assist researchers and penetration testers in their learning process. Soliman et al (2018) [32] conducted research on web application API blind Denial of Service attacks. The authors acknowledge that DoS attacks are difficult to detect and costly to defend against. The research investigates a hybrid type of DoS technique - a blind DoS attack. It benefits both from network and application-level attacks.

The usage of web APIs is increasing rapidly due to their flexibility and relatively easy access by consumers. APIs are built on generic technologies such as HTTP for transport and do not require costly integration into third-party infrastructure or consumer-side applications. Standards and recommendations advise on the importance of sufficient security logging and monitoring systems. Web APIs rely on JSON or XML as common data formatting standards. Web APIs use HTTP for the transport layer and information about HTTP methods and payload used for requests issued by attackers is of interest to security teams.

One of the main challenges for web application honeypots is their relatively easy recognition by attackers. Web application honeypots can be used for getting the adversary resources bogged down or for revealing attack source information, to block unwanted connections. Cloud services provide new technical capabilities for building and demolishing honeypot systems rapidly and on-demand, saving time and resources. Threat intelligence is vital for successful cyberdefense strategies. Production systems following best design practices will simplify matters for legitimate consumers of an API. On the contrary, the lack of documentation for honeypot API should provoke malicious actors to try various techniques, potentially revealing their methods.

2.3 Interview with EVCN operator

To strengthen the background information and learn more about commercial implementations, a team of engineers responsible for operations at one of the major EVCN operators was interviewed. The team confirmed that the EVCN utilizes multiple web APIs for managing subscriptions, chargers, and other intermediary devices. Web APIs are also used for the monitoring of the infrastructure. The particular EVCN has an operations center hosting a back-end management solution, remotely interacting with users, partner networks, and the charging network via a set of web APIs. The implemented web APIs enable remote operators to make all possible changes on the devices, altering settings, resetting them, and switching on and off chargers. According to the interlocutors, the communication from the external world to the back-end management system is restricted to the following set of APIs:

- EVCN customer portal API. This is a REST API enabling customers to manage their subscriptions, and charging sessions, and monitor charging activities and billing information.
- EVCN chargers API. This is an API used to manage charger devices operated by the network.
- EVCN management portal API. This is REST API enabling EVCN operators to manage charging sessions, and charging devices, monitor the health status of the network and its segments, and perform administrative tasks and duties.
- External partner network chargers API. This is an API used to communicate with partner EVCNs and enable customers to use third-party partner charging networks.
- External partner users API. This is an API used by external partner companies to enable their users to use the charging network.

Figure 2.1 shows the generalized communication scheme of the back-end management system with other components, as it was described by the interviewed EVCN technical team.

The interviewed team did not disclose any security-sensitive information regarding the infrastructure setup, security, and defense mechanisms implemented. During the conversation, the engineers confirmed that the company among other security counter-measures also relies on data collected using honeypot systems. The threat intelligence data is gathered through a set of decoy applications mimicking the behavior of production systems. Also, the technical team confirmed that having a web API honeypot system mimicking one or several components of a production back-end solution built in the cloud would be an interesting addition to the existing set of API honeypot systems in service.

2.4 Relating state of the art to EVCN web API honeypot

Standards and special publications contain a number of recommendations, emphasizing the importance of threat intelligence for developing effective cyberdefense strategies. CIA triad's Detect function implies integration of comprehensive security monitoring and anomalies detection mechanisms into digital infrastructures [10]. ISO/IEC 27005:2022 standard provides guidance on assessing of likelihood of risk scenarios and establishing criteria for risk acceptance [11]. "Security Strategies for Microservices-based Application Systems: SP 800-204" by NIST covers APIs and API gateways as the entry points and recommends the implementation of security monitoring and anomaly detection solutions [12].

The "API technical and data standards" guidance by Central Digital and Data Office UK emphasizes the importance of the security audit and monitoring activities [13]. OWASP's "API Security Project" also focuses on the importance of secur-

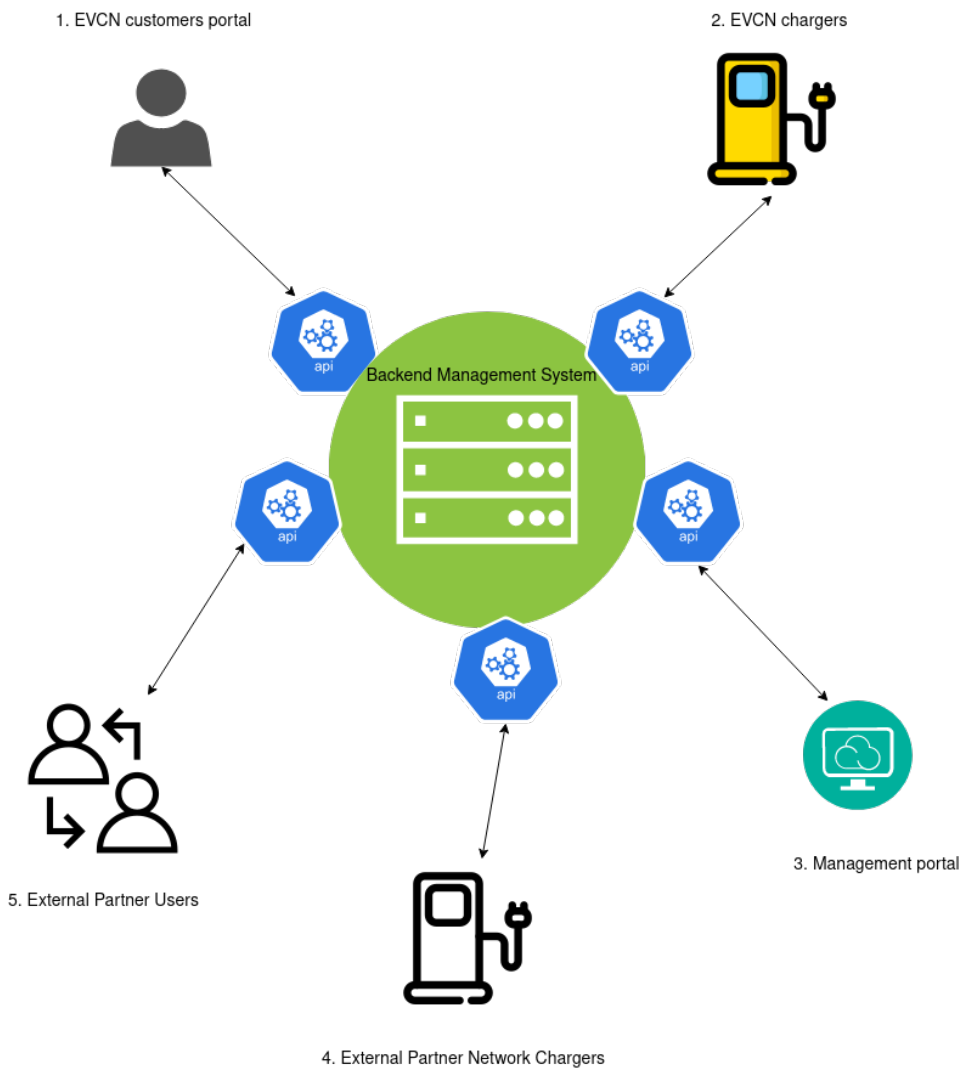

1. EVCN customers portal
2. EVCN chargers
Backend Management System
api
api
api
api
api
3. Management portal
4. External Partner Network Chargers
5. External Partner Users

Figure 2.1: EVCN communication diagram.

ity logging [14]. The UK National Cyber Security Centre recommends designing critical national infrastructure in accordance with certain principles, minimizing the impact of potential security breaches [15]. Threat intelligence gathered by honeypot systems could be of great value, helping to understand the behavior of potential intruders and ensure solid security design. Web API honeypot systems employed by EVCN operators might become important contributors to collecting threat intelligence and complying with the recommendations and guidances of security standards .

The previous work section shows that the digital cloud is a flexible and cost-effective way of building and operating honeypot systems [16]. Honeypot systems are great contributors to the task of gathering threat intelligence [17]. Web API honeypot systems provide valuable information helping to combat digital threats [18], [19], [20]. Web API systems evolved into crucial interconnectivity mechanisms [21], relieving their consumers from using custom libraries. Once designed properly web APIs make communication between business systems efficient and reliable [22], [23], [24].

Low-interaction honeypots are useful where cost and scalability are deciding factors [27]. At the same time, in certain scenarios low-interaction honeypots are capable of delivering valuable threat intelligence [28], [29], [31]. There is another use for low-interaction honeypot systems, such as performing as imposter applications and wasting resources of attackers [30].

This chapter provided information on general recommendations and standards. Also, it offered some insight into the previous work by reviewing the literature. The chapter is concluded with an interview with a technical team of one of the main EVCN operators in the Scandinavian region. The next chapter describes the methodology used to design the research and build the prototype.

Chapter 3

Methodology

3.1 Problem Statement

The following problem statement is formulated for the research:

EV charging networks use web API solutions for the management and integration of physical infrastructure. Little knowledge is publicly available about the degree such systems are targeted by cyberattackers.

3.2 Research Objectives

The following objectives will be achieved in order to solve the problem stated:

Main objective: Build a prototype of a low-interaction web API honeypot with a security logging subsystem. The application has to mimic a web API of EVCN management system.

In order to achieve the main objective, the following sub-objectives have to be addressed:

- Build a decoy EVCN web API to manage chargers.
- Make an attempt to fully automate the deployment of core components.
- As a proof of concept, make the solution available online, and log requests issued by attackers.
- Organize the collected data related to attacks in JSON format and plot in Grafana visualization solution.

3.3 Research design

The research is designed to identify and address the shortcomings of previous work in the same area. Also, the work explores research questions, that previously have not been answered in detail. It is comprised of two main activities - the background review and prototyping. The background review includes open sources such as special publications, previous work, and an interview with an

EVCN operator advising on the management system used for the network. Based on this, the work can be classified as one following the qualitative approach, conducting exploratory research.

3.4 State of the art review

The State of the art review is comprised of three parts. Special publications describing standards and recommendations from internationally recognized authorities are placed in the first part. A literature review is a method employed to investigate a previous work accomplished on the same problem. Academic papers are to contribute to the task of avoiding unnecessary repetition of previous research and help to identify research areas missed.

In order to identify techniques, tools, and approaches used in the previous work on the same topic, a systematic literature review was conducted in the second part of the State of the art review. General information on honeypot systems, usage of web APIs in honeypot systems, and previous efforts to build similar platforms are the main objectives of the literature survey. To search for scientific papers of interest, the search string "WEB AND API OR API Honeypot" was created and used. The databases used are ACM digital library, SpringerLink, and IEEE Xplore. Table 3.1 shows the number of hits returned by the databases used :

Table 3.1: Scientific databases search results

ACM Digital Library	64751 hits
IEEEXplore	2291 hits
SpringerLink	560 hits

Using exclusion parameters the total number of filtered papers was reduced to 223. Then the method of grey literature review, examining the initial data including title, year of publication, and abstract, helped to shortlist the number of papers chosen for review to 39. In the last stage, a final list of 15 papers of interest was compiled, excluding duplicates and less relevant ones. After selecting all the necessary resources, papers were read and all relevant information concerning method, process, phases, and techniques was extracted and reflected on in the section "2.2 Related work".

The final third part of the background review contains information about an interview with the EVCN operator. The interviewed technical team has provided important information about commercial implementations of web APIs in modern EVCNs. The interview confirmed some assumptions that were made previously, based on the literature review and available public information.

3.5 Prototyping

The second method employed to address the main and underlying objectives is prototyping. Prototyping is a technique widely used in scientific research in order to build a proof of concept model. A prototype usually has limited functionality compared to a final product, but it has to reflect the main design architecture and meet the functionality requirements of a solution proposed[33]. Although it is not expected to be a final product, the prototype built for this work will provide the declared functionality of the web API honeypot. The next section explains in detail the prototype model and proposed architecture.

3.5.1 Prototype model

The prototype will emulate one API out of five, described in section 2.3. Specifically, it will emulate the behavior of API, connecting the EVCN back-end management system to the management portal, used by operators. Operators manipulate charger settings through the management portal using the REST API. The back-end management system receives commands issued by operators and sends them to the chargers using the OCPP protocol.

The rest of the APIs is not implemented in the prototype, as it is just a matter of increasing the number of APIs deployed and does not influence the core functionality of the API honeypot. One of the goals of the project is to build an API honeypot prototype, that is easy to rescale, deploy and operate. This goal is achieved by employing cloud technologies with no upfront cost and on-demand resource utilization. The low-level maintenance burden for modeling and building the prototype is another factor, influencing the architecture. Cloud providers offer various solutions to help users with quick and low-cost deployments.

AWS cloud offers a concept of serverless applications, where no static resources are allocated. The utilized resource pool is hidden behind the deployment boundaries. The resources are used on demand so that users are billed based on the actual utilization of the cloud resources. The API honeypot model for this work is based on AWS services and aims to use AWS cloud-native components without any third-party add-ons.

There are various categories of web APIs, with Representational State Transfer (REST) [34] or Simple Object Access Protocol (SOAP) [35] protocols-based APIs being the most widespread ones. A web API complying with the design principles of the REST protocol is referred to as RESTful API. The protocol of choice for this work is REST due to its current prevalence on the market and AWS native support. This work uses the terms "web API" and "REST API" interchangeably unless specified otherwise.

Although the concepts of high and low-interaction honeypot systems have been through many developments and changes in the last two decades, some generic characteristics still remain the same [27]. Considering the fact that attackers will not interact with the operating system or underlying services, the proposed

API honeypot can be attributed to the family of low-interaction honeypot systems.

3.5.2 Expected functionality

The solution will receive web requests from the Internet and log request data such as timestamp, HTTP method, body of the request, source IP address, and Transmission Control Protocol (TCP) port information. This list can be extended depending on the future needs of the API honeypot operators and the capabilities of the logging components utilized. The AWS REST API gateway will be published on the Internet and listen for incoming HTTP requests. When a request arrives, the API gateway will accept it and trigger the back-end application.

Although there are a few HTTP methods that can be implemented in the REST API [36], this work will implement the most common three of them - GET, PUT and DELETE. HTTP GET method is used to request data and it is not able to modify data. It can be used by legitimate users for fetching information from a web API as well as by adversaries for the reconnaissance stage of attack against a web API. The HTTP PUT method is used to create new or modify existing resources. Attackers can use this method to manipulate the settings of EVCN devices, causing service interruptions and instability. The HTTP DELETE method is used to delete resources [37]. It can be used for wiping out objects behind web API and rendering it useless.

The constant development of technologies implies platform-independent data communication. Common data formats help to resolve differences between various applications, participating in the data exchange. Data meant for further processing or investigation needs to be formatted in a way that third parties are able to computationally read and process it. The output data for this work is decided to be in JSON format. After analyzing and processing logs, the JSON output will enable data for further analysis and usage in most modern data storage and visualization system.

3.5.3 Expected outcome

The expected outcome of the research is a prototype of the AWS cloud-based low-interaction REST API honeypot. The solution proposed will help small and middle-size EVCN operators to quickly build and re-scale their own honeypot application, collecting threat intelligence data. The output datasets will provide important details about sources of attacks conducted, used methods, and a request body for each issued request. Threat intelligence data can significantly improve the overall security preparedness of EVCN operators against attacks. It will also contribute to more systematic threat analysis and comprehensive statistics concerning the security and availability of APIs.

This chapter discussed the problem statement and research objectives. It presented the methodology chosen for the research. The approaches used in conducting interviews with EVCN operators, literature review and special publications review were explained. Also, the prototype model and expected functionality were explained. The next chapter proposes the general architecture and introduces a prototype of the low-interaction REST API honeypot built in the AWS cloud.

Chapter 4

Prototyping

This chapter describes the setup of the REST API honeypot solution and all underlying services and components, required for its operation. The chapter walks users through each component and explains its role and configuration details. This project uses the AWS cloud as a building platform and utilizes Free Tier eligible components for prototyping. The reason for choosing AWS as a building platform for prototyping is its flexibility and offer of cost-free subscription if resources are used within given limits. Users must read and understand AWS billing documentation to avoid unexpected costs related to the deployment of the REST API honeypot.

4.1 EVCN REST API structure

Real-life EVCNs have various API resources covering charger and intermediary devices, users, billing, monitoring, and other services. Compared to a prototype, there would be a lot more objects and their attributes kept in the back-end database, serving data to a production API. This project builds an operational REST API with bare minimum functionality that can be easily extended to any scale. The REST API honeypot structure proposed in this prototype has very few resources and attributes. It serves the main purpose of the project, reflecting a functionality of the REST API honeypot sufficient for the demonstration of the prototype.

4.1.1 General architecture

The general architecture of the REST API honeypot is comprised of several components:

- API gateway accessible from the Internet
- Back-end database solution
- Logging component, registering all incoming requests and response data
- Log data parsing solution, searching through log files for the data of interest, extracting it, representing in a JSON format, and writing to a database and local file

Figure 4.1 depicts high-level API honeypot architecture and indicates its abstract workflow.

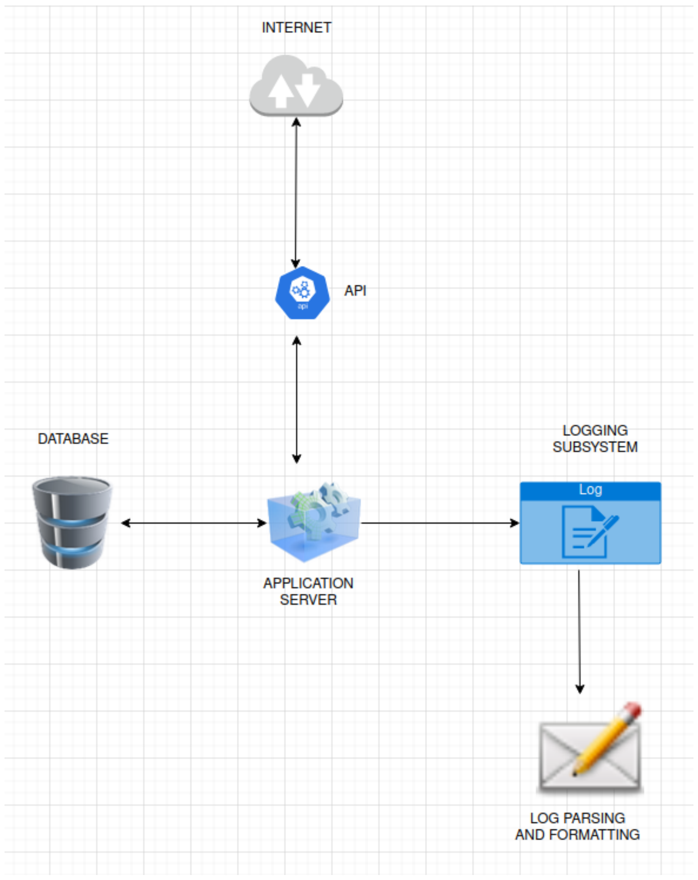


Figure 4.1: High-level architecture of the REST API honeypot.

The proposed architecture can be implemented using various solutions and technologies. However, the implementation using components having no integration is associated with problems. The following section proposes a model, solving the integration issues and offering a cost-effective and straightforward solution for building a low-interaction REST API honeypot in the AWS cloud.

4.2 AWS architecture, core components

This section presents the implementation of the proposed architecture in the AWS cloud. The prerequisite for the project is an AWS cloud account [38]. A free AWS cloud account is also can be used for testing purposes. After obtaining an AWS cloud account the AWS command-line access was set and configured [39].

The flowchart of the AWS REST API honeypot is shown in Figure 4.2 and comprised of the following components:

- API Gateway - the front door of the application, providing access to the back-end data, functionality, or business logic [40].
- Lambda function service - AWS service enabling users to run a code without provisioning server instances [41]
- IAM Role and Policy/Permissions - web service maintaining authentication and authorization over the AWS resources [42]
- Dynamo DB - AWS NoSQL database service [43]
- CloudWatch Logs - AWS monitoring and logging service [44]

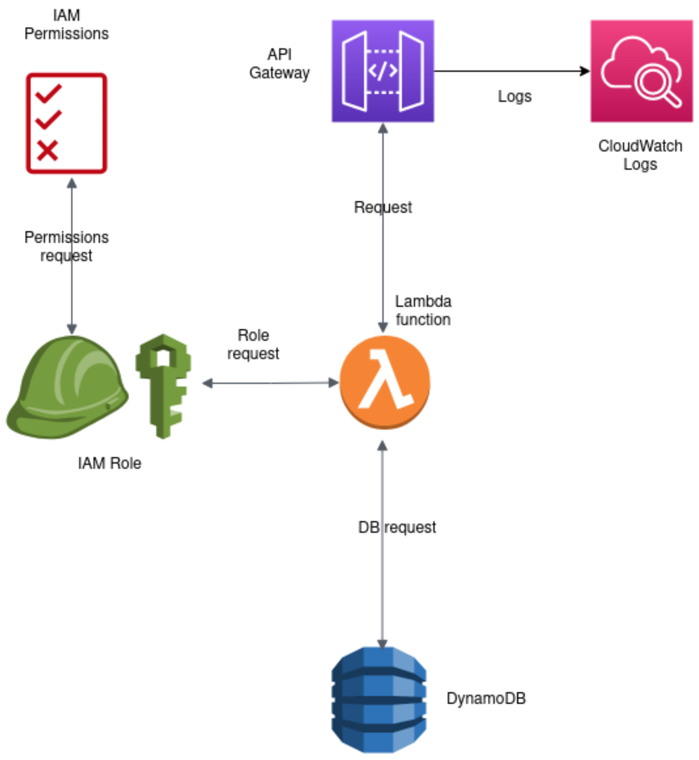


Figure 4.2: Flowchart.

4.2.1 AWS DynamoDB

Before provisioning any resources and building a REST API honeypot the database to store data was created. DynamoDB is a database of choice for the current work. The database schema defined for the project is kept as simple as possible. The first steps were to create a DynamoDB table, assign a name to it and specify the partition key "id" with "String" being a value type. Alternatively, the DynamoDB table can be created from the AWS command line using the Code listing 4.1:

Code listing 4.1: AWS CLI create DynamoDB table

```
# Create a DynamoDB table, and assign its name "ChargingNetworkDB"
aws dynamodb create-table \
--table-name ChargingNetworkDB \
--attribute-definitions AttributeName=id,AttributeType=S \
--key-schema AttributeName=id,KeyType=HASH \
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
```

After the table was created the database was populated with sample data. DynamoDB allows a user to create objects of various types - string, number, boolean, binary, sets of strings, numbers or binaries, list, or map. For this project charger objects with fictitious attributes and assigned string values were created. The example settings for the charger object are shown below:

```
"id":"1"
"SN":"MD129F643H"
"Model":"c12a"
"Output":"22"
"City":"Korsa"
"Street":"Korsveien 3"
"PostCode":"19021"
```

During the setup of resources, all Amazon Resource Names (ARNs) were noted down. ARNs are unique identifiers used to specify resources across the whole AWS. The example DynamoDB ARN looks similar to the following one:

```
"arn:aws:dynamodb:us-east-1:ACCOUNTID:table/ChargingNetworkDB"
```

where ACCOUNTID is the ID of your AWS user account.

4.2.2 AWS IAM

The next step was to create a role and assign policies, enabling AWS resources to interconnect. The "Trusted entity type" of the role was set to "AWS service" with "Lambda" selected for "Use case". The role was configured to have the "AWSLambda-BasicExecutionRole" Permissions Policy assigned. Also, an inline permission policy allowing specific actions such as "GetItem" or "PutItem" were assigned to the same role.

In some scenarios "All Resources" permission can be chosen, assuming that the account is used only for experimentation and does not contain any production resources. As an alternative to the AWS Web GUI management console, an AWS IAM role and relevant policies can be created from the AWS command line using the Code listing 4.2:

Code listing 4.2: AWS CLI create IAM role and policy

```
# Create IAM role and assume trust policy
aws iam create-role --role-name ChargingNetworkRole \
--assume-role-policy-document file://trust.json

# Attach inline IAM policy to the role created
aws iam put-role-policy --role-name ChargingNetworkRole --policy-name \
ChargingNetworkInlinePolicy --policy-document \ file://inline_policy.json

# Attach Lambda execution role policy
aws iam attach-role-policy --policy-arn \
     arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole \
     --role-name ChargingNetworkRole
```

The information about policy files trust.json and inline_policy.json used in the commands above can be found in Appendix A.5.

4.2.3 AWS Lambda

The next step was to deploy a set of Lambda functions, that process requests hitting the API. It was decided to deploy three Lambda functions per REST API resource with one of them having working functionality and two functions returning fake error messages and trying to lure an intruder to alter his attempt and try again. The prototype used GET and DELETE methods as dummy ones and had the PUT method actually implemented. Python 3.9 as a runtime and Boto3 AWS Software Development Kit were used for Lambda functions. Boto3 is a well-known SDK and widely used for managing AWS services [45].

The Lambda PUT function used for the prototype is shown in the Code listing 4.3:

Code listing 4.3: Python Lambda handler for PUT method

```
import boto3

def lambda_handler(value, context):
  connector = boto3.resource('dynamodb')
  db = connector.Table('ChargingNetworkDB')
  api_resp = db.put_item(
  Item={
  "id": value['id'],
  "City": value['City'],
  "Model": value['Model'],
  "Output": value['Output'],
  "PostCode": value['PostCode'],
  "SN": value['SN'],
  "Street": value['Street']
  }
  )
  return {
  'statusCode': api_resp['ResponseMetadata']['HTTPStatusCode'],
  'API response': 'Record ' + value['id'] + ' added'
}
```

The information about Lambda GET and DELETE functions used for the project can be found in Appendix A.6. To configure the AWS Lambda service, the AWS Lambda functions with Python language runtime and appropriate permissions were created. The default execution role was set to the role created in section 4.2.2. After creating a new function, the placeholder code was replaced with the corresponding Lambda functions code. After adding the code and creating test events[46], functions were deployed.

Alternatively, Lambda functions can be created from the AWS command line. Actual lambda functions have to be uploaded as separate files in compressed format. The example code can be found in Appendix A.1, "install.sh" script.

4.2.4 AWS API Gateway

API Gateway is a central mechanism tying all the previous components together and providing a single interface towards a customer, in this case - towards the adversary. The example prototype API structure used in this project is shown in Figure 4.3:

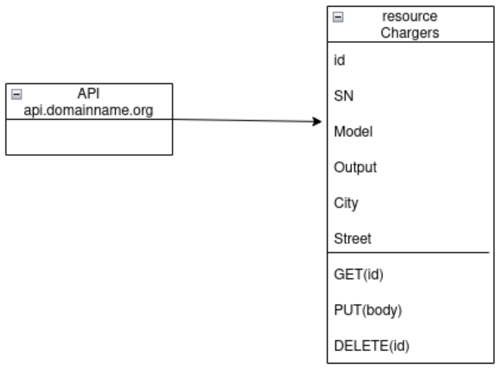


Figure 4.3: Prototype API structure.

The proposed prototype uses the AWS REST API with one single resource "Chargers". After creating the "Chargers" resource in the AWS API Gateway console, the HTTP methods such as "GET", "PUT" and "DELETE" were created to interact with the resource. The methods were configured with integration type "Lambda Function" with corresponding Lambda functions created in section 4.2.3. Also, the methods created had relevant permissions assigned to the corresponding Lambda functions. Methods were configured with method requests and corresponding URL Query String Parameters settings. For example, the "PUT" method used the following mapping template in order to answer API calls:

```
{
"id": "$input.params('id')",
"City": "$input.params('City')",
"Model": "$input.params('Model')",
"Street": "$input.params('Street')",
"SN": "$input.params('SN')",
"Output": "$input.params('Output')",
"PostCode": "$input.params('PostCode')"
}
```

GET and DELETE methods had the same settings, including integration type, mapping template, and URL Query String Parameters. The only entry required for GET and DELETE methods in the mapping template was the "id" parameter. After configuring resources and methods, the API was ready to be deployed. It was deployed from the API resources page. Once deployed, the API stage editor produced the summary and "Invoke URL" for the API. The example invoke URL is shown below:

```
"https://h6da3w6ume.execute-api.us-east-1.amazonaws.com/production".
```

The complete invoke URL for "/chargers" path is shown below:

```
"https://h6da3w6ume.execute-api.us-east-1.amazonaws.com/production/chargers"
```

4.2.5 AWS CloudWatch

To collect logs for further analysis, the AWS CloudWatch service was used. AWS API Gateway service has integration with the AWS CloudWatch logging subsystem, enabling users to collect and analyze logs. "Full Request and Response Logs" was chosen as the verbosity level. Logs are organized in groups with a naming standard corresponding to the service they are utilized for. This project used a custom log parsing application and exported logs to the AWS S3 storage service for that matter. After exporting logs they were downloaded to Linux Ubuntu Desktop 22.04 virtual machine for further processing.

4.2.6 Automate the deployment of the API honeypot

The core of the REST API honeypot solution is comprised of the four main elements - IAM role and policies, DynamoDB table, Lambda functions, and API Gateway with resources and methods. The rest of the components described further are not directly involved in the functioning of the API honeypot and are related to the logging subsystem and further processing of collected data. Considering that, the automation step of the project has focused on the four core elements mentioned.

The automation was implemented in AWS CLI, version 2.3.6. The automation solution "install.sh" presented in appendix section A.1 starts with the creation of the DynamoDB table. In the next step, it employs the AWS CLI IAM module to create a role and attach inline and Lambda execution policies. After setting up roles and policies, Lambda functions are created. Lambda functions for the project are Python scripts, implementing GET, PUT and DELETE methods using Python 3.9 AWS Boto3 library. To upload files containing functions to the Lambda service, the files are compressed into a ZIP archive and then uploaded to the AWS Lambda service. In the next step, the script sets Lambda functions up, configuring runtime and linking the IAM role.

Once the DB table, IAM role, and Lambda functions are created and configured, the "install.sh" script creates REST API. The API is configured with a resource and HTTP methods assigned to it. HTTP methods created need to be configured with necessary links to the corresponding Lambda functions. Also, method request and response, as well as integration request and response have to be configured with relevant settings, adding request templates and response models. This is done for all three HTTP methods created for each specific API resource. Then Lambda functions' permissions need to be configured to link them to the specific source in the REST API. Once permissions are configured, the API has to be deployed in order to accept requests and provide responses. The deployment step requires a stage name, that will be used at the last step of updating stage with the enabled CloudWatch logging subsystem. The install.sh script starts Docker containers with the Grafana data visualization application and InfluxDB database engine. It also creates the AWS S3 service bucket used for log export from the CloudWatch service and sets the access policy on the bucket.

There is a few important points for users of the automation tool "install.sh" to be noted. It is assumed that the user of the tool is an advanced Linux user and has a decent understanding of Linux shell scripting. The "install.sh" should not be used against the AWS account containing any production or other important resources. The best scenario for deployment of the REST API honeypot is under a dedicated AWS account, isolated from all other resources. Usage without understanding what the tool is doing and using accounts other than intended for the deployment of the REST API honeypot can destroy other existing resources if they are linked to the same account. The tool is written just for demonstration purposes to prove that the deployment of the proposed solution can be fully automated.

4.3 Optional components

This study provides a functional prototype, that is ready for deployment. Components in the following sections concerning log parsing and storing extracted data in a database, as well as data visualization, can be altered or replaced by a user according to her needs and the specifics of the project the solution is used for.

4.3.1 Authentication and HTTP methods implementation

After configuring HTTP methods, a user can choose to enable authentication in the REST API Gateway for each method. Enabling API authentication is a straightforward process and requires a very little configuration in the AWS API Gateway service. For each created method GET, PUT and DELETE, a user needs to enable authentication by requesting an API key from the Method Request settings. The next step is to create an actual API key and assign it to the Usage Plan created for that purpose. The usage Plan feature also enables users to control the access limits to the API. After configuring the API key, setting the Usage Plan, and enabling authentication for each HTTP method, the API has to be redeployed to enable authentication.

4.3.2 Mapping a custom domain name for the REST API

After deploying the AWS gateway and making sure that all the functionality works as expected, the next step was a setup of a custom domain name for the API honeypot. AWS generates random names for API gateways such as:

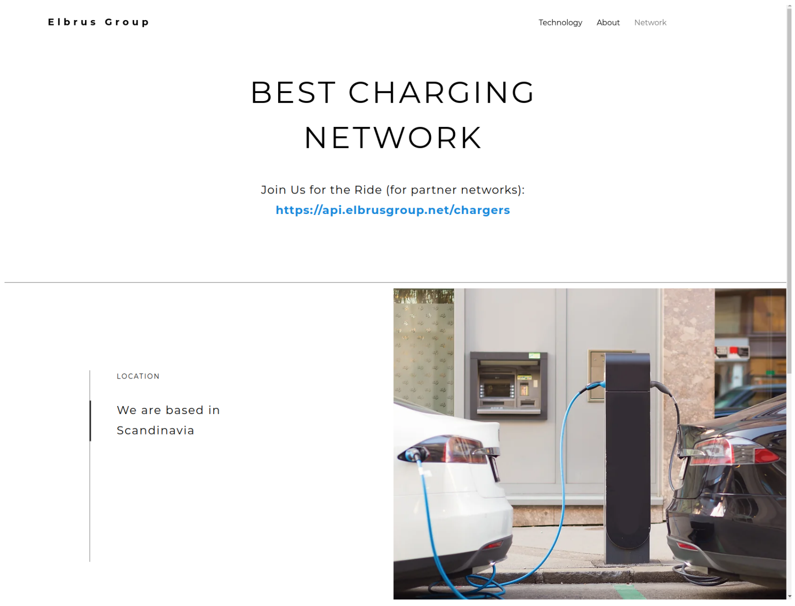
```
"https://h6da3w6ume.execute-api.us-east-1.amazonaws.com/production/chargers"
```

Usually, businesses do not use AWS-generated names and prefer to map APIs to domains of their own. This project used the "elbrusgroup.net" domain name for the REST API mapping and further activities. The domain name was registered using AWS "Route 53" service and belongs to the author of the project[47]. After a domain name was registered, it was mapped to the REST API following AWS instructions "Setting up custom domain names for REST APIs" [48]. It is required to use SSL certificates to establish a mapping. SSL certificate for the domain was requested using AWS Certificate Manager [49]. The project used the hostname of "api.elbrusgroup.net" to route API calls to an actual API gateway with stage "Production" and no path specified. Effectively, the auto-generated API URL was forwarded to the URL "api.elbrusgroup.net". Path parameter "chargers" was added directly to the Uniform Resource Locator (URL), used to call the API, e.g.

```
"https://api.elbrusgroup.net/chargers".
```

4.3.3 "Leaking" API settings to the Internet

To attract potential attackers and make them aware of the API, it is necessary to advertise it. It is out of the scope of this work how to prepare and execute a deception operation to make attackers believe that the API honeypot represents a real operational solution. Intruders probe the Internet all the time and soon or later the API honeypot will be discovered by them. However, for demonstration purposes to get some attackers to quickly realize that a new API is on the air, this project published the API using a custom domain name and a website for fictitious EVCN operator Elbrus Group. The website can be found following the link "www.elbrusgroup.net". The last page of the fake EVCN with the REST API honeypot URL is shown in Figure 4.4. This is a very naive illustration of how the actual honeypot can be advertised or "leaked" to potential attackers.


Elbrus Group

BEST CHARGING NETWORK

Join Us for the Ride (for partner networks):
https://api.elbrusgroup.net/chargers

LOCATION

We are based in Scandinavia

Figure 4.4: Elbrousgroup EVCN web-site.

4.3.4 AWS S3

To export collected logs to AWS S3 service a bucket in the S3 storage with a linked policy allowing export was created. The policy was applied on the permissions page of the bucket. To make AWS accept the policy it was necessary to disable Block Public access temporarily, add Bucket policy, save it and then enable back Block public access. The S3 policy "s3-policy.json" can be found in Appendix A.5. After the policy was created and applied, the logs were exported from AWS CloudWatch to the AWS S3 bucket.

4.3.5 Processing collected logs

After the Linux Ubuntu Desktop 22.04 virtual machine was ready and it had access to the S3 bucket with exported CloudWatch logs, the log files were synced to the machine's local storage from the S3 bucket. The log files ended up in a folder structure with one root folder and sub-folders corresponding to each log file. All log files in the sub-folders at the time of this writing were compressed in GZIP format and had the default naming format "000000.gz". The next section provides an explanation of the log parsing process, implemented in the Python3 programming language.

4.3.6 Parsing logs using Python application and storing the data in JSON format

After the log files were organized in the folder structure described above and uncompressed, the data of interest were extracted. This work used the Python3 applications "logparser_batch.py" and "logparser_realtime.py" written to search through the log file structure, find the log files, analyze their content and process them depending on the HTTP method registered. The "logparser_batch.py" can be found in Appendix A.2. Its functionality is to process logs in batch mode. The "logparser_realtime.py" can be found in Appendix A.3. Its functionality is to process logs in a continuous or near real-time mode. Besides the processing mode, the functionality of both applications is the same. This work will refer to "logparser_batch.py" and "logparser_realtime.py" as "logparser.py", as they both have the same functionality and can be used in post-processing or real-time scenarios accordingly. There are three HTTP methods activated in this prototype - GET, DELETE and PUT. It means only log files containing one of the three methods are to be investigated. If none of the three methods is found in a log file, it will be ignored by the application.

The application maintains the following workflow:

- main() - the main function is initialized first and requires a path to the root folder of the log structure as an argument. The main function uses the helper function file_reader() searching recursively through the folder and sub-folders of the path provided and finding all compressed files and uncompressing them. Then files are searched for the HTTP method logged. If GET, PUT, or DELETE methods are found within the body of a log file, it is passed to one of the following parser functions accordingly.
- delete_parser() - the first function out of three parser functions reading a file passed to it by the main function. It reads through the content of a log file and extracts the following values concerning a specific request - epoch or timestamp of the request, source IP address, TCP port, HTTP method, the body of the request, AWS unique request ID, resource path, and the status returned by the API gateway. The function creates a dictionary corresponding to each unique timestamp. The main dictionary contains a timestamp and 2 sub-dictionaries, containing the HTTP method used, source IP address, source port, body or payload of the request, AWS API gateway request ID, resource path, and status code returned by the API gateway. This output structure with relevant elements is maintained in all three parser functions for using it in the InfluxDB structure. Two sub-dictionaries are used as fields and tags in the InfluxDB structure [50].
- get_parser() - the second function out of three parser functions. It reads through the content of a log file and extracts the following values - epoch or

timestamp of the request, source IP address, TCP port, HTTP method, body or payload of the request, and AWS unique request ID. The function creates a dictionary corresponding to each unique timestamp. The main dictionary contains a timestamp and 2 sub-dictionaries, containing the HTTP method used, source IP address, source port, body or payload of the request, AWS API gateway request ID.

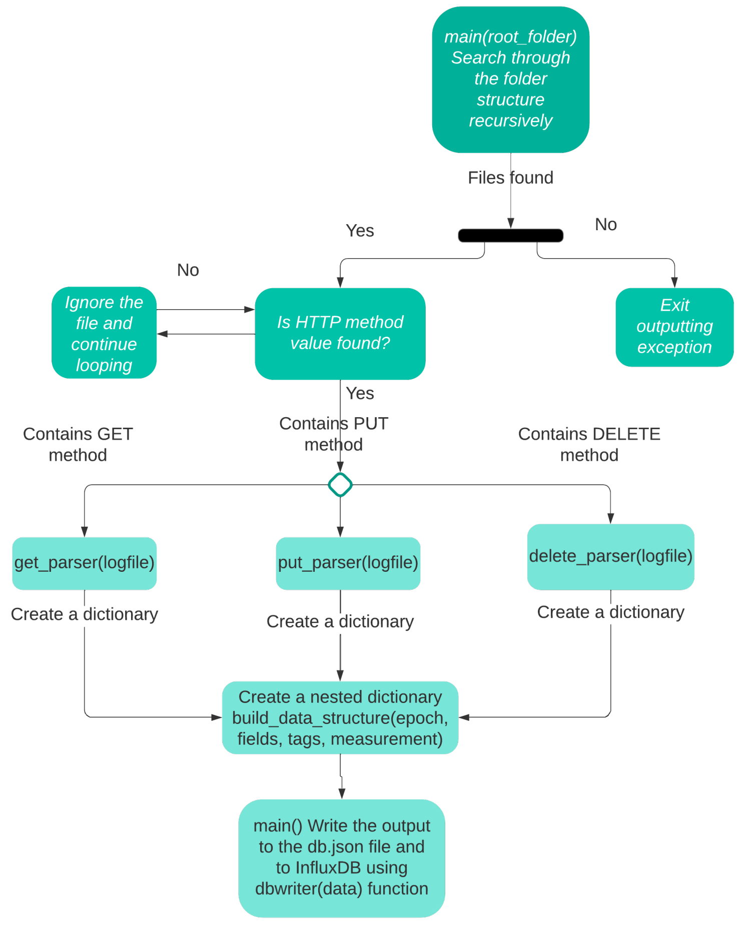

main(root_folder) Search through the folder structure recursively
Files found
Yes
No
No
Ignore the file and continue looping
Is HTTP method value found?
Exit outputting exception
Yes
Contains GET method
Contains PUT method
Contains DELETE method
get_parser(logfile)
put_parser(logfile)
delete_parser(logfile)
Create a dictionary
Create a dictionary
Create a dictionary
Create a nested dictionary build_data_structure(epoch, fields, tags, measurement)
main() Write the output to the db.json file and to InfluxDB using dbwriter(data) function

Figure 4.5: "logparser.py" application workflow diagram.

- put_parser() - the last function out of three parser functions. It reads through the content of a log file and extracts the following values - epoch or timestamp of the request, source IP address, TCP port, HTTP method, the body of the request, and AWS unique request ID. The function creates a dictionary corresponding to each unique timestamp. The main dictionary contains a timestamp and 2 sub-dictionaries, containing the HTTP method used, source IP address, source port, body or payload of the request, AWS API gateway request ID.

- build_data_structure() - after parsing a file, the control is returned back to the main function. The main function calls build_data_structure() function to create a JSON data structure and append all required entries to it. The resulting output for one single log file processed is a list of JSON data containing a timestamp, fields dictionary, tags dictionary, and measurement value. Every next file analysis by the application results in a similar output appended to the list created.
- As a final step the main function writes the output in JSON format to a local file "db.json", located in the same folder as "logparser.py" and to the database specified via dbwriter() Python function. dbwriter() Python function takes the final list created as input, creates a connection to the InfluxDB instance specified, and writes data to the database.

Figure 4.5 shows the activity diagram reflecting the procedural flow of actions of the Python3 application "logparser.py" and how log files with each method were processed using the application. When the final output data structure was created and written to a file or a database, the results were ready for visualization. The output data can be investigated to register source IP addresses and their geolocations, the intensity of attacks based on the timestamps, the prevalence of specific HTTP request methods over others, and the content of the payload of the requests issued by attackers. These data points can be combined and correlated with specific events of interest. This work is not focusing on such events or correlations as the conclusions drawn from the available data are out of the scope of this project.

4.3.7 Alternative way of parsing logs using AWS Custom Access Logging

Alternatively, users can parse access logs using the Custom Access Logging feature of AWS [51]. AWS defines variables and functions that the REST API can use with AWS CloudWatch access logging. It is possible to output the data of interest in a JSON format using log formatting templates offered by AWS. At the time of this writing, the AWS Custom Access Logging was not sufficiently granular and flexible for this project and it was decided to use a custom parser written in Python programming language written for this project and described in section 4.3.6.

4.3.8 Visualization of the collected data using Grafana and InfluxDB

The data output in JSON format can be used in various ways for analysis and visualization. Grafana is Open Source front-end solution for data visualization. It represents data in tables, graphs, charts, and many other forms. For this project, Grafana version 9.2.0 running in a Docker container was used. InfluxDB version 1.8.4 running in a Docker container was used as a database. The output of "logparser.py" application has been written to the InfluxDB and then visualized using Grafana plots. Some sample plots demonstrating the data points are shown in the next chapter.

This chapter explained the architecture and setup of the REST API honeypot solution in the AWS cloud. Sections 4.2.1 up to 4.2.5 are necessary to follow in order to build a core of the REST API honeypot solution with a logging subsystem. Further sections are optional and it is up to the user how to process, parse and organize data once the setup is completed and CloudWatch logs are collected. The next chapter will present the results of the experimentation with the deployment of the REST API honeypot prototype.

Chapter 5

Results

5.1 Test deployment

In order to test the functionality and prove that the REST API honeypot works as expected, it was deployed in the AWS cloud using a Free Tier AWS account. Immediately after deployment, the API honeypot was ready to catch attempts made by potential attackers and in a few days, it showed traces of interaction in the CloudWatch logging subsystem. The backend DynamoDB database contained 25 charger objects with attributes shown in section 4.2.1. The test deployment used "All resources" permission for IAM inline policy attached to the IAM security role created for the Lambda execution. IAM role and policy configuration reflected the section 4.2.2.

As was described in section 4.2.3, three Python functions were defined to perform GET, PUT and DELETE HTTP methods. Test deployment included one REST API resource "chargers" and three methods corresponding to GET, PUT and DELETE Lambda functions. API Gateway configuration has been done according to section 4.2.4. All four core elements of the REST API honeypot were deployed and configured using AWS web graphical user interface. To test the automation attempt made in section 4.2.5, the deployment was deleted and rebuilt using "install.sh" script shown in Appendix A.1. The automated deployment has been completed successfully with all the components being installed and configured as expected.

After completing the deployment of core components, the test deployment continued with the mapping of a custom domain name to the REST API. At this point the REST API honeypot was published on the Internet, listening for incoming requests. The prototype was online from 20.10.2022 to 23.11.2022, when it was stopped and removed. In order to demonstrate the capability of the API honeypot, two teams of penetration testers "Team 1" and "Team 2" were asked to make an attempt on the published API. The logs were collected from the CloudWatch logging subsystem and exported to the AWS S3 storage. To parse and further process the data of interest, the data was downloaded from S3 buckets to a virtual machine with Linux Ubuntu Desktop 22.04 operating system.

Section 4.3.6 describes the Python application "logparser.py" written for log parsing and storing the data of interest in the JSON format. Collected logs were processed using the "logparser.py" application and extracted data was sent in a local "db.json" file and the InfluxDB instance in a Docker container. The test deployment used a Grafana front-end solution for the data visualization, as it is described in section 4.3.8. The next section contains some sample data visualized using instances of InfluxDB and Grafana. The data caught was a mix of data originating from attempts of "friendly" penetration testers and real attempts to access the API honeypot by unknown adversaries. Although the deliverable for this project is the REST API honeypot prototype and full data analysis is out of the scope of the project, some data was visualized in Grafana plots and discussed below.

5.2 Data visualization

Figure 5.1 demonstrates the plot showing IP addresses attacking the API honeypot and the number of requests issued by "Team 1" against the API honeypot on 23.11.2022 between 18.55 and 19.32 CET. IP addresses are blurred out due to privacy reasons.

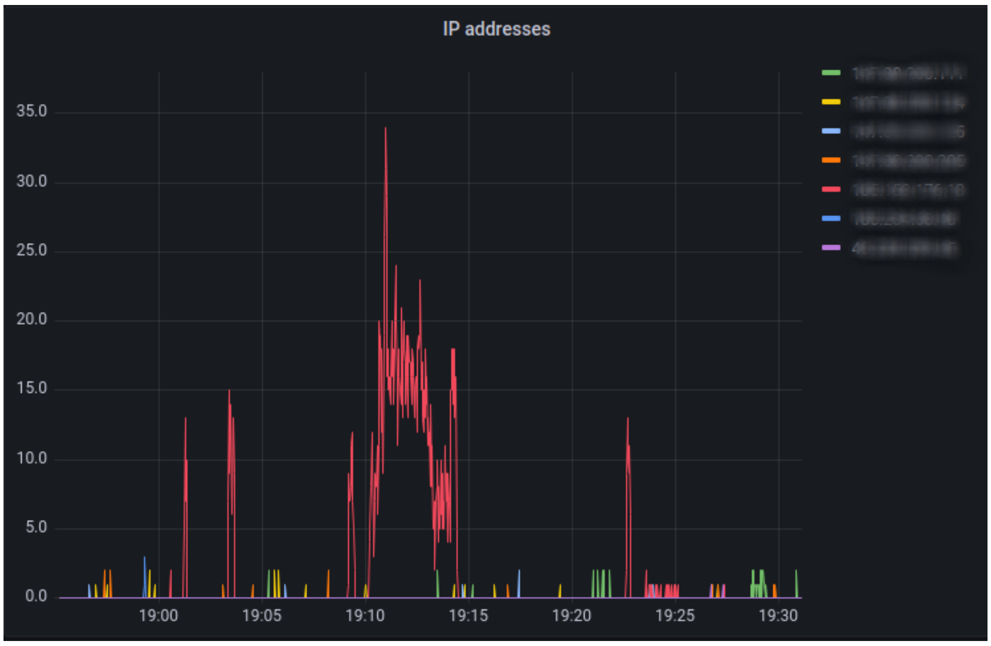


Figure 5.1: Team 1 attacking IP addresses and the number of attempts.

Next Figure 5.2 demonstrates the second plot with statistics of HTTP methods used by "Team 1" in the same period. The plot shows that the PUT method issued by "Team 1" significantly prevails over the GET and DELETE methods in the given period of time.

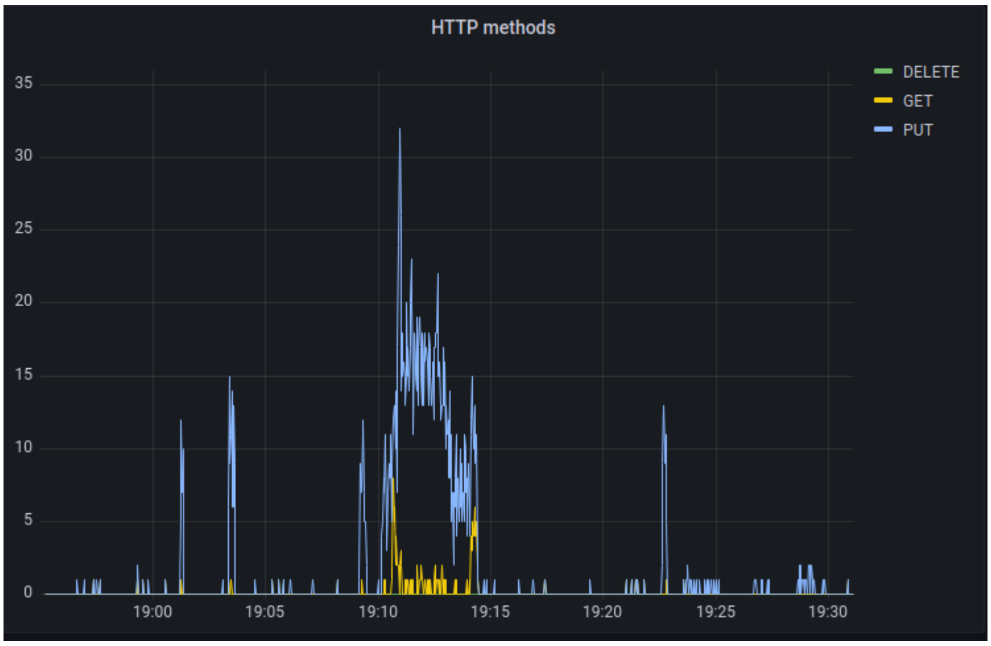


Figure 5.2: Team 1 statistics on HTTP methods.

The next plot shown in Figure 5.3 visualizes the statistics and content of the payload issued by attackers for the same period of time. The right side of the plot shows the payload of the requests issued. The next three plots demonstrate some statistics for the traffic generated by "Team 2". The plot in Figure 5.4 demonstrates the IP address attacking the API honeypot and the number of requests issued by "Team 2" on 04.11.2022 between 17.41 and 17.44 CET. The IP address is blurred out due to privacy reasons.

The last Figure 5.5 shows the plot with statistics of HTTP methods issued by "Team 2". The plot shows that the GET method issued by "Team 2" prevails over the PUT method in the given period of time. The plots above provide an example of how the data extracted from logs can be used for statistics and threat intelligence. The list of fields extracted and visualized can be easily extended enriching the threat intelligence data. The overall results of the test deployment were satisfactory and met the expectations. The test deployment of the API honeypot prototype was running for the period of one month and demonstrated the expected stability level.

This chapter presents the results achieved by the study. The next chapter discusses the results and provides an overview of the research questions and whether they were answered.

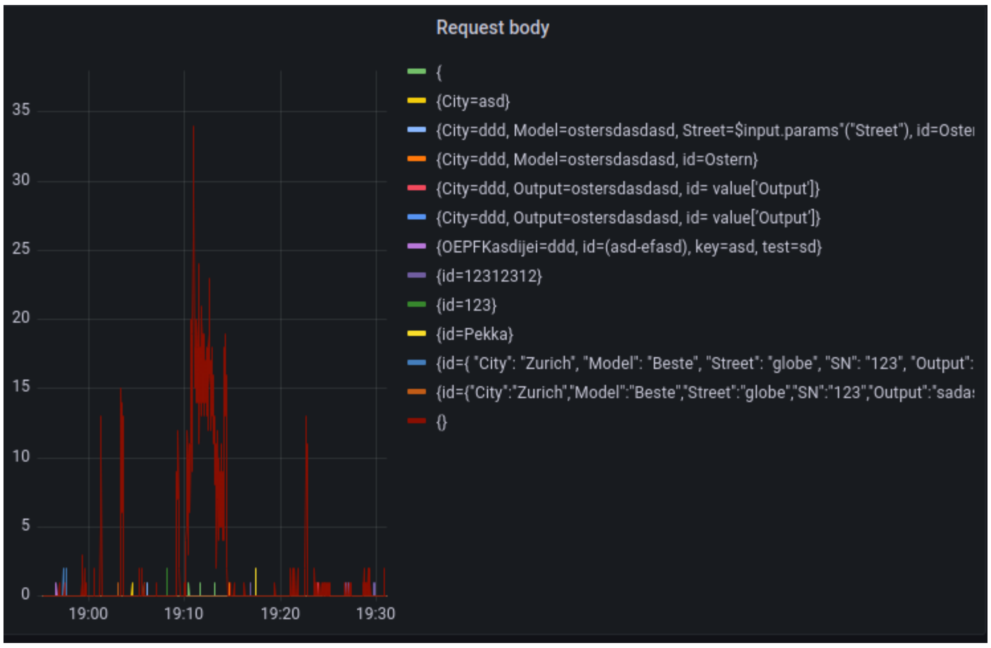


Figure 5.3: Team 1 payload statistics.

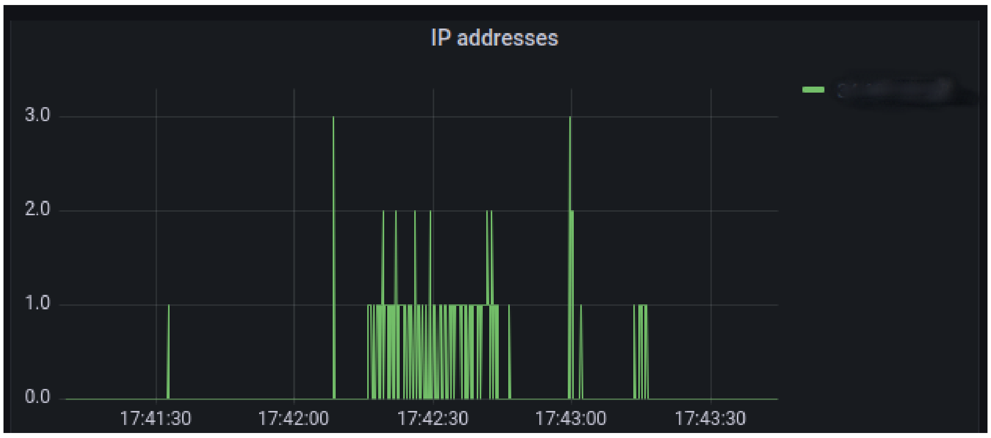


Figure 5.4: Team 2 attacking IP addresses and the number of attempts.

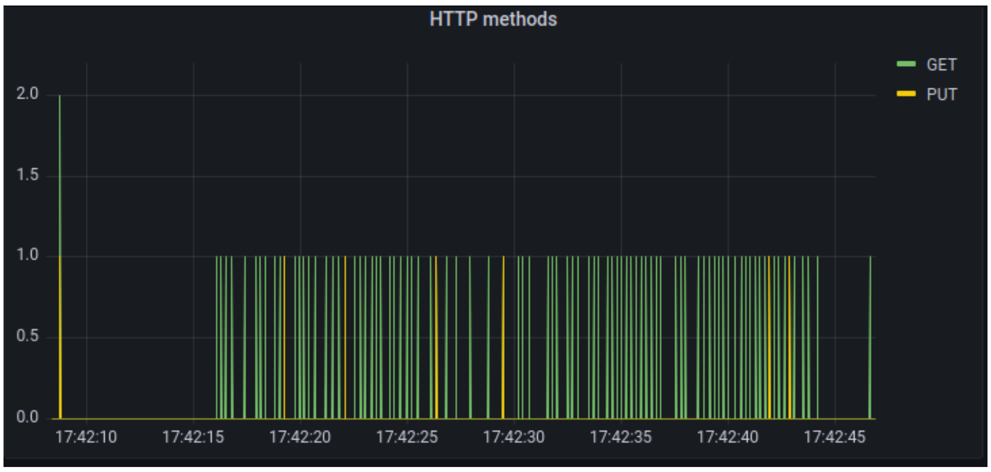


Figure 5.5: Team 2 statistics on HTTP methods.

Chapter 6

Discussion

The study achieved the declared objectives and answered research questions. The achievement of objectives is discussed in the next chapter. This chapter will clarify how the research questions were answered, what was established by the study, what are the findings, and the advantages, and weaknesses of the proposed solution.

6.1 Findings

After the deployment and configuration of the prototype, the API honeypot was successfully published using a custom domain name. Also, a web-builder service for creating websites was hired and a website for fictitious EVCN operator "Elbrus Group" was created. The website also published the information about API.

Although naive, these actions were undertaken to provide more credibility to the API honeypot in the view of potential intruders. Two weeks after setting up a website and publishing web API information on it, some access attempts were registered and logged. Obviously, an existing EVCN operator would have an online presence with a website and other components, making published API more credible.

The prototype proved that it is possible to deploy and start a web API honeypot operation using the proposed prototype with a reasonable effort and low investment for small and middle-size EVCN operators. It will let users focus on their core activities instead of investing significant effort and financial resources to build and support additional server and network infrastructure, developing API with underlying log collection, parsing, and visualizing systems. The web API honeypot can be used for collecting threat intelligence data as well as for distracting adversaries from real production systems and wasting their effort and resources.

The prototype has implemented only three HTTP methods - GET, DELETE and PUT. The list can be extended to cover the rest of the HTTP methods available in the AWS REST API Gateway service. Also, only one resource "chargers" was used for the prototype. The model should be enriched with any other resources covering objects relevant to an operational EVCN such as "users", "subscriptions", "vehicles" and others. [52] The deployment shell script "install.sh" provided in Appendix A.1 and the batch mode and near real-time versions of the "logparser.py" Python application provided in Appendix A.2 and A.3 can be easily extended to cover new methods and resources.

Initially, there were fully operational Lambda functions and authentication enabled for each HTTP method. But after some considerations, authentication was disabled for the prototype. The prototype was deployed as a proof of concept for a limited time, rather than with the intention to perform a full-blown honeypot operation. To make the prototype catch requests from attackers immediately after deployment and demonstrate the functionality, it was decided to make all calls by intruders reach the API by removing authentication. The thinking behind it is that authentication would significantly complicate intrusion for attackers and prolong the time until they gain access to the prototype.

Also, the Lambda functions corresponding to GET and DELETE HTTP methods initially were functional and could get and delete the objects requested from the API. Later, it was decided to replace both with dummy handler functions returning the following message with the relevant operation name instead of "X": "An error occurred (ValidationException) when calling the X operation: One or more parameter values are not valid!". GET calls are usually made to fetch some data from the API and can be used for reconnaissance to build an understanding of the victim API structure. Once the attacker believes that he knows enough about the structure and design of the API, her next step would be the execution of PUT or DELETE calls against the API trying to inflict damage or alter the data behind it. In this case, if the GET call does not return any valuable information about objects behind API, the attacker is expected to try finding a way around it by guessing the correct parameters. Also, he most probably will make more attempts using PUT and DELETE calls, guessing the REST API structure.

The techniques used to overcome this barrier are of interest to security professionals. For the full honeypot operation in order to make the web API honeypot look like an operational API, the authentication should be enabled and API should have more resources, objects, and HTTP methods implemented. At the same time, the credentials should be either "leaked" to intruders or weak enough to not withstand brute-force attacks. Once intruders pass the authentication mechanism, the API honeypot will collect their HTTP requests and analyze information about their reconnaissance techniques. This information will help to build a pattern over the actions and parameters they use, contributing to the development and implementation of security countermeasures, and defending production APIs.

After using the API honeypot for a while its data post-processing architecture became a limitation. It was developed further to a near real-time model with up to 180 seconds of delay by making the log export and parsing process continuous. The 180 seconds delay is caused by subsequent delays in AWS log generation and export systems. The data visualization component Grafana has all the real-time data processing tools embedded and engaging them has enabled a continuous data stream from the InfluxDB to visualization plots. The information about the near real-time data processing model of the application is presented in Appendix A.3.

Considering the small size of the dataset collected by the prototype, first, it was decided to store the data in a regular file and use it as inline input to the Infinity plugin of the Grafana data visualization solution [53]. After some experimentation with Grafana and Infinity plugin, the results were less sufficient than expected and it was decided to use the InfluxDB database as a data storage.

6.1.1 Advantages of the web API honeypot

One of the main advantages of the proposed prototype is its scalability. The flexibility of the proposed prototype significantly exceeds the potential of an average small or middle-sized infrastructure in terms of deployment of API resources and the number of requests it can process. The deployment can be easily enlarged to tens, hundreds, or thousands of API resources with almost limitless transactions and storage for database items. Hardware and network utilization limits of the proposed API honeypot in the cloud are equal to the boundaries drawn by the specific cloud provider. Usually, those boundaries are well-beyond beyond the technical capabilities of the majority of on-premise infrastructures. This is especially relevant to small and middle-sized businesses, such as local and regional EVCN operators. The ability to rescale the web API honeypot resources will help to match and understand the resource-wise capabilities of adversaries and distinguish between them based on the scale of attacks.

Another main advantage of the web API honeypot is its modular nature. Core components such as API Gateway, Lambda functions, and DynamoDB database are decoupled from each other and have no dependency ties configuration and resource-wise. Any component can be duplicated without interrupting the running API honeypot operation, altered and swiftly attached to the solution preserving the operational status.

Automation is a very important feature in modern computing and often becomes a deciding factor for using or declining digital solutions. The automation potential of the proposed system is extremely high and not limited to one single solution, e.g. command line. Cloud providers offer a rich toolset to automate operations and deployment. AWS cloud is not an exception and constantly develops and introduces new technical features, helping to take automation to a new level. One example of an alternative to the AWS command line is the AWS CloudFormation service [54]. Users can automate the deployment and management of AWS resources using CloudFormation templates.

6.1.2 Limitations of the web API honeypot

The web API honeypot solution was deployed using both the AWS web console and AWS CLI script "install.sh", shown in Appendix A.1. Although the AWS web console is user-friendly and intuitive, there are a few drawbacks related to using it. As with any GUI, its usage is time-consuming and not flexible. It is not possible to automate user actions via the web console. So, the AWS command line could be the preferable solution for deployment and operation. It means that a user needs to familiarize herself with the AWS command line environment if automation is of importance for the specific project.

Another limitation of the solution is its bond to AWS. It is not possible to easily port the solution to other cloud service providers. Although the general architecture can be preserved while deploying to other cloud providers, the components and configuration will be different and specific to the provider of choice. Although, the log parsing application "logparser.py" written for the project still can be reused with minor alterations in the text parsing mechanisms.

6.2 Answering questions

6.2.1 Research question 1

Is it advisable to build a web API honeypot for the EV charging networks?

Question 1 was answered by studying existing work and interviewing an EVCN operator. The conclusion drawn after the literature review is that honeypot systems in general are of great interest to researchers and businesses. Due to its widespread usage, a web API is an attractive target for adversaries. The interview with EVCN technical team led to the conclusion that a web API honeypot system can be a useful tool, but it takes some effort and resources to build and maintain it on-premise. It helps to collect threat intelligence data, contributing to a better understanding of an adversary and a more effective cyberdefense strategy. Based on the literature review performed, no similar research on building a low-interaction web API honeypot for EVCN operators in the AWS cloud was conducted before. It was found advisable to build a web API honeypot for the EV charging networks.

6.2.2 Research question 2

How to build a web API honeypot solution in the AWS cloud with an on-demand resource utilization design?

Considering the increasing usage of cloud services and automation tool-set offered by modern cloud providers, it was decided to build the solution in the AWS cloud. The study answered research question 2 by building a prototype of web (REST) API in the AWS cloud. The prototype is fully functional and equipped with a logging subsystem.

After analyzing the technical potential of the AWS cloud, it was decided to take one step further and deploy a serverless REST API honeypot, using cloud-native tools based on the on-demand resource utilization model. The proposed prototype uses the AWS API Gateway service paired with the AWS Lambda service. Requests sent to the API honeypot trigger Lambda functions, using an on-demand resource utilization model instead of allocating the resources permanently. The proposed design and the prototype provided in Chapter 4 answered research question 2.

6.2.3 Research question 3

To what degree the deployment of the web API honeypot can be automated?

Along with configuration through AWS web GUI, the research proposes automated deployment of core components using the "install.sh" deployment script presented in Appendix A.1. Research question 3 was answered by creating the script and making test deployments. Taking the "logparser.py" application from the batch mode to the near real-time model was another successful step towards automation. Assuming correct AWS user account configuration, the whole solution can be deployed by running one single script "install.sh". After deployment, the solution can be run in batch or near real-time mode by running either "logparser_batch.py" or "logparser_realtime.py" applications accordingly.

This chapter discussed the findings, advantages, and limitations of the API honeypot and explained whether the research questions were answered. The next chapter provides the conclusion on the work conducted and proposes ideas for future work.

Chapter 7

Conclusion

7.1 Study and achievements

The main objective of the study is to build a prototype of a low-interaction web API honeypot with a security logging subsystem. The application has to mimic a web API of EVCN management system. The main objective is based on the following sub-objectives:

- Build a decoy EVCN web API to manage chargers.
- Make an attempt to fully automate the deployment of core components.
- As a proof of concept, make the solution available online, and log requests issued by attackers.
- Organize the collected data related to attacks in JSON format and plot in Grafana visualization solution.

The study aimed to achieve the main objective by addressing sub-objectives. Building a decoy EVCN REST API to manage chargers was the first one to address. The study reached an operational stage of the REST API honeypot prototype with working functionality. The setup with a declared functionality is reflected in the Chapter 4.

The second sub-objective aimed to automate the deployment of the core components. Deployment of all core components was automated and tested. Shell application for deployment is provided in Appendix A.1. The third sub-objective was to prove the concept and make the solution available online, trying to log requests and data issued by attackers. API honeypot was published online and collected some logs about requests made, fulfilling the sub-objective. Finally, the last sub-objective was to process the logs collected, organizing the data in a JSON data structure and visualizing it in Grafana plots. The sub-objective was achieved and the results of data visualization were demonstrated in the Chapter 5.

The study fulfilled the main objective by addressing the four sub-objectives. The prototype proved to be a working solution with a proof of concept model. Logs reflecting the activities of intruders were collected and parsed. The data of interest such as timestamps, attack source IP addresses, HTTP methods, and payload were extracted and organized in the InfluxDB database system. The data then was visualized in several plots using the Grafana solution. In the final stage, the web API honeypot was forked from batch mode, adding a near real-time version. The information about the details of near real-time setup can be found in Appendix A.3.

This work can be used by EVCN operators seeking to understand the adversary. IP addresses extracted from the logs can be used to block attackers and compute the geolocation of the attacking nodes using Golang, Python, or other programming language modules. The timestamps reflecting the intensity of the attacks, HTTP methods used and payload of the requests contain important information helping to understand the technical capabilities of attackers and design countermeasures to be taken for defense. The API honeypot is useful for researchers interested in collecting and analyzing threat intelligence data from low-interaction REST API honeypot systems. Also, the API honeypot can be used for distracting attackers from the actual API resources of an organization and making them waste time and resources trying to attack a decoy API. The solution is flexible and can be adapted or repurposed with relatively minor tweaking. The REST API honeypot built in the AWS cloud is innovative, as no similar solutions were found in the previous work.

The objectives for future work are proposed in the following section.

7.2 Future work

The proposed web API honeypot prototype has only one "chargers" resource. It is advisable to gather information about production APIs used by EVCN providers and add relevant resources to the prototype. Also, the attributes for charger objects in a production network would contain a lot more items and should be considered for relevant extensions.

The API honeypot has only three HTTP methods implemented. Production implementations should aim for the full implementation of all HTTP methods available in the AWS API Gateway service. Also, the study focused on the extraction of certain fields from the API honeypot log files. AWS logging system provides detailed request and response logs for API Gateway service and extraction of data from additional fields can be beneficial for researchers. One example of the existing field that was not used in this work is the user agent field in the log file, reflecting a user-side application used to access the API.

Bibliography

[1] International Energy Agency, ‘Global EV outlook 2022,’ 2022. [Online]. Available: `https://iea.blob.core.windows.net/assets/ad8fb04c-4f75-42fc-973a-6e54c8a4449a/GlobalElectricVehicleOutlook2022.pdf`.

[2] European Union Agency For Network and Information Security. ‘Mapping of OES Security Requirements to Specific Sectors.’ (2017), [Online]. Available: `https://www.enisa.europa.eu/publications/mapping-of-oes-security-requirements-to-specific-sectors`.

[3] International Organization for Standardization. ‘Road vehicles — vehicle to grid communication interface.’ (2019), [Online]. Available: `https://www.iso.org/standard/69113.html`.

[4] Redhat. ‘What is an API?’ (2022), [Online]. Available: `https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces`.

[5] Open Charge Alliance. ‘Open Charge Point Protocol 1.6.’ (2022), [Online]. Available: `https://www.openchargealliance.org/`.

[6] R. Fielding, U. Irvin, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. ‘Hypertext Transfer Protocol – HTTP/1.1.’ (1999), [Online]. Available: `https://www.rfc-editor.org/rfc/rfc2616.html`.

[7] S. Köhler, R. Baker, M. Strohmeier and I. Martinovic, *Brokenwire : Wireless Disruption of CCS Electric Vehicle Charging*, 2022. arXiv: `2202.02104 [cs.CR]`.

[8] P. Bock, J.-P. Hauet, R. Françoise and R. Foley, *Ukrainian power grids cyberattack*, 2017. [Online]. Available: `https://www.isa.org/intech-home/2017/march-april/features/ukrainian-power-grids-cyberattack`.

[9] IETF. ‘The JavaScript Object Notation (JSON) Data Interchange Format.’ (2017), [Online]. Available: `https://www.rfc-editor.org/rfc/rfc8259`.

[10] NIST. ‘Framework for Improving Critical Infrastructure Cybersecurity.’ (2018), [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/cswp/nist.cswp.04162018.pdf`.

[11] International Organization for Standardization, *ISO/IEC 27005:2022*, 2022. [Online]. Available: `https://www.iso.org/obp/ui#iso:std:iso-iec:27005:ed-4:v1:en`.

[12] R. Chandramouli. 'Security Strategies for Microservices-based Application Systems.' (2019), [Online]. Available: `https://csrc.nist.gov/publications/detail/sp/800-204/final`.

[13] Central Digital Data Office. 'API technical and data standards.' (2018), [Online]. Available: `https://www.gov.uk/guidance/gds-api-technical-and-data-standards`.

[14] Open Web Application Security Project. 'OWASP API Security Project.' (2019), [Online]. Available: `https://owasp.org/www-project-api-security/`.

[15] National Cyber Security Centre. 'Cyber security design principles.' (2022), [Online]. Available: `https://www.ncsc.gov.uk/collection/cyber-security-design-principles/cyber-security-design-principles`.

[16] S. Biedermann, M. Mink and S. Katzenbeisser, 'Fast Dynamic Extracted Honeypots in Cloud Computing,' in *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '12, New York, NY, USA: Association for Computing Machinery, 2012, pp. 13–18, ISBN: 9781450316651. DOI: `10.1145/2381913.2381916`. [Online]. Available: `https://doi.org/10.1145/2381913.2381916`.

[17] Ryandy, C. Lim and K. E. Silaen, 'XT-Pot: EXposing Threat Category of Honeypot-Based Attacks,' in *Proceedings of the International Conference on Engineering and Information Technology for Sustainable Industry*, ser. ICONETSI, New York, NY, USA: Association for Computing Machinery, 2020, ISBN: 9781450387712. [Online]. Available: `https://doi.org/10.1145/3429789.3429868`.

[18] Ng, Chee Keong and Pan, Lei and Xiang, Yang, 'Specialized Honeypot Applications,' in *Honeypot Frameworks and Their Applications: A New Framework*. Springer Singapore, 2018, pp. 15–41, ISBN: 978-981-10-7739-5. [Online]. Available: `https://doi.org/10.1007/978-981-10-7739-5_3`.

[19] C. Cheh and B. Chen, 'Analyzing OpenAPI Specifications for Security Design Issues,' in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 15–22. DOI: `10.1109/SecDev51306.2021.00019`.

[20] J. A. Díaz-Rojas, J. O. Ocharán-Hernández, J. C. Pérez-Arriaga and X. Limón, 'Web API Security Vulnerabilities and Mitigation Mechanisms: A Systematic Mapping Study,' in *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 2021, pp. 207–218. DOI: `10.1109/CONISOFT52520.2021.00036`.

[21] S. Sohan, C. Anslow and F. Maurer, 'A Case Study of Web API Evolution,' in *2015 IEEE World Congress on Services*, 2015, pp. 245–252. DOI: `10.1109/SERVICES.2015.43`.

[22] E. Wittern, A. Ying, Y. Zheng, J. A. Laredo, J. Dolby, C. C. Young and A. A. Slominski, ‘Opportunities in Software Engineering Research for Web API Consumption,’ in *Proceedings of the 1st International Workshop on API Usage and Evolution*, ser. WAPI ’17, IEEE Press, 2017, pp. 7–10, ISBN: 9781538628058S.

[23] M. Tello-Rodríguez, J. O. Ocharán-Hernández, J. C. Pérez-Arriaga, X. Limón and Á. J. Sánchez-García, ‘A Design Guide for Usable Web APIs,’ *Programming and Computer Software*, vol. 46, no. 8, pp. 584–593, Dec. 2020, ISSN: 1608-3261. DOI: 10.1134/S0361768820080241. [Online]. Available: https://doi.org/10.1134/S0361768820080241.

[24] E. Wilde, ‘Surfing the API Web: Web Concepts,’ in *Companion Proceedings of The Web Conference 2018*, ser. WWW ’18, Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, pp. 797–803, ISBN: 9781450356404. DOI: 10.1145/3184558.3188743. [Online]. Available: https://doi.org/10.1145/3184558.3188743.

[25] World Wide Web Consortium. ‘Extensible Markup Language (XML).’ (2008), [Online]. Available: https://www.w3.org/TR/REC-xml/REC-xml-20081126.xml.

[26] World Wide Web Consortium. ‘RDF 1.1 Concepts and Abstract Syntax.’ (2014), [Online]. Available: https://www.w3.org/TR/rdf11-concepts/.

[27] D. Watson. ‘Low-interaction honeypots revisited.’ (2015), [Online]. Available: https://www.honeynet.org/2015/08/06/low-interaction-honeypots-revisited/.

[28] M. Musch, M. Härterich and M. Johns, ‘Towards an Automatic Generation of Low-Interaction Web Application Honeypots,’ in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018, Hamburg, Germany: Association for Computing Machinery, 2018, ISBN: 9781450364485. DOI: 10.1145/3230833.3230839. [Online]. Available: https://doi.org/10.1145/3230833.3230839.

[29] L. Rist, S. Vetsch, M. Koßin and M. Mauer. ‘A dynamic, low-interaction web application honeypot.’ (2010), [Online]. Available: https://www.honeynet.org/download/glastopf-a-dynamic-low-interaction-web-application-honeypot/#.

[30] E. Chiapponi, O. Catakoglu, O. Thonnard and M. Dacier, ‘HoPLA: a Honeypot Platform to Lure Attackers,’ in *CESAR 2020, Computer Electronics Security Applications Rendez-vous, Deceptive security Conference, part of European Cyber Week, 14-15 December, Rennes, France*, EURECOM, Ed., 2020.

[31] M. Idris, I. Syarif and I. Winarno, ‘Development of Vulnerable Web Application Based on OWASP API Security Risks,’ in *2021 International Electronics Symposium (IES)*, 2021, pp. 190–194. DOI: 10.1109/IES53407.2021.9593934.

[32] M. Soliman and M. A. Azer, 'Web Application API Blind Denial of Service Attacks,' in *2018 14th International Computer Engineering Conference (ICENCO)*, 2018, pp. 249–253. DOI: 10.1109/ICENCO.2018.8636115.

[33] B. Camburn, V. Viswanathan, J. Linsey, D. Anderson, D. Jensen, R. Crawford, K. Otto and K. Wood, 'Design prototyping methods: state of the art in strategies, techniques, and guidelines,' *Design Science*, vol. 3, e13, 2017. DOI: 10.1017/dsj.2017.10.

[34] R. T. Fielding. 'Architectural Styles and the Design of Network-based Software Architectures.' (2000), [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.html.

[35] World Wide Web Consortium. 'SOAP Version 1.2.' (2007), [Online]. Available: https://www.w3.org/TR/soap/.

[36] Amazon Web Services. 'Set up an HTTP method.' (2022), [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-settings-method-request.html#setup-method-add-http-method.

[37] R. Fielding, M. Nottingham and J. Reschke. 'RFC 9110 HTTP Semantics.' (2022), [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110.html.

[38] Amazon Web Services. 'How do I create and activate a new AWS account?' (2022), [Online]. Available: https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/.

[39] Amazon Web Services. 'Configuring the AWS CLI.' (2022), [Online]. Available: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html.

[40] Amazon Web Services. 'What is Amazon API Gateway?' (2022), [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html.

[41] Amazon Web Services. 'What is AWS Lambda?' (2022), [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[42] Amazon Web Services. 'What is IAM?' (2022), [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html.

[43] Amazon Web Services. 'What is Amazon DynamoDB?' (2022), [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

[44] Amazon Web Services. 'What is Amazon CloudWatch?' (2022), [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html.

[45] Amazon Web Services. 'Boto3 documentation.' (2022), [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[46] Amazon Web Services. 'Testing Lambda functions in the console.' (2022), [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/testing-functions.html.

[47] Amazon Web Services. 'Registering a new domain.' (2022), [Online]. Available: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/domain-register.html.

[48] Amazon Web Services. 'Setting up custom domain names for REST APIs.' (2022), [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-custom-domains.html.

[49] Amazon Web Services. 'Issuing and managing certificates.' (2022), [Online]. Available: https://docs.aws.amazon.com/acm/latest/userguide/gs.html.

[50] Influxdata. 'InfluxDB key concepts.' (2015), [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/concepts/key_concepts/.

[51] Amazon Web Services. 'API Gateway mapping template and access logging variable reference.' (2022), [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html.

[52] eDRV. 'Data Hierarchy.' (2022), [Online]. Available: https://docs.edrv.io/docs/data-hierarchy.

[53] Grafana. 'Grafana visualization.' (2022), [Online]. Available: https://grafana.com/grafana/.

[54] Amazon Web Services. 'AWS CloudFormation Documentation.' (2022), [Online]. Available: https://docs.aws.amazon.com/cloudformation/index.html.

Appendix A

API deployment automation

The Github repository containing software and configuration files:

```
https://github.com/mamedxanli/webAPIHoneypot
```

The following sections describe all components in the repo.

A.1 install.sh

Install.sh script installs and configures the main components and some sub-components of the web API honeypot:

A.2 logparser_batch.py

Python application for parsing logs and writing data to InfluxDB in a batch mode. Can be scheduled to run via crontab.

A.3 logparser_realtime.py

Python application for parsing logs and writing data to InfluxDB in a near real-time mode. Can be scheduled to run via crontab.

A.4 rollback.sh

Shell script to remove resources created by install.sh script

A.5 aws_policies folder

The folder contains all AWS-specific policies: trust.json, inline_policy_dynamodb.json, s3-policy.json.

A.6 cnDeleteFunction.py, cnGetFunction.py, cnPutFunction.py

Python functions used in AWS Lambda service.

A.7 db-var.env

Environmental variables for InfluxDB

A.8 docker-compose.yml

Docker-compose setup file