Degree Project in the Field of Technology Cyber Security and the Main Field of Study Computer Science
Second cycle, 30 credits

HackerGraph: Creating a knowledge graph for security assessment of AWS systems

ALEXIOS STOURNARAS
HackerGraph: Creating a knowledge graph for security assessment of AWS systems

ALEXIOS STOURNARAS

Date: September 7, 2023
Supervisor: Viktor Engström
Examiner: Mathias Ekstedt
School of Electrical Engineering and Computer Science
Abstract

With the rapid adoption of cloud technologies, organizations have benefited from improved scalability, cost efficiency, and flexibility. However, this shift towards cloud computing has raised concerns about the safety and security of sensitive data and applications. Security engineers face significant challenges in protecting cloud environments due to their dynamic nature and complex infrastructures. Traditional security approaches, such as attack graphs that showcase attack vectors in given network topologies, often fall short of capturing the intricate relationships and dependencies of cloud environments. Knowledge graphs, essentially a knowledge base with a directed graph structure, are an alternative to attack graphs. They comprehensively represent contextual information such as network topology information and vulnerabilities, as well as the relationships between all of the entities. By leveraging knowledge graphs’ inherent flexibility and scalability, security engineers can gain deeper insights into the complex interconnections within cloud systems, enabling more effective threat analysis and mitigation strategies. This thesis involves the development of a new tool, HackerGraph, specifically designed to utilize knowledge graphs for cloud security. The tool integrates data from various other tools, gathering information about the cloud system’s architecture and its vulnerabilities and weaknesses. By analyzing and modeling the information using a knowledge graph, the tool provides a holistic view of the cloud ecosystem, identifying potential vulnerabilities, attack vectors, and areas of concern. The results are compared to modern state-of-the-art tools, both in the area of attack graphs and knowledge graphs, and we prove that more information and more attack paths in vulnerable by-design scenarios can be provided. We also discuss how this technology can evolve, to better handle the intricacies of cloud systems and help security engineers in fully protecting their complicated cloud systems.

Keywords

Cloud security, Knowledge graph, Attack graph, Vulnerability assessment, Attack paths, Vulnerable-by-design systems, Cloudgoat
Abstract
Sammanfattning


Nyckelord

Molnsäkerhet, Kunskapsgraf, Attackgraf, Sårbarhetsanalys, Sårbara miljöer, Cloudgoat
Sammanfattning
Acknowledgments

I would like to express my deepest gratitude to Mathias Ekstedt, my professor, and Viktor Engström, my supervisor for their support, guidance, and invaluable expertise throughout the entire duration of my thesis. Their constructive criticism and insightful feedback have been instrumental in helping to shape this thesis.

I would also like to thank my friends who have provided me with support and encouragement during this challenging yet rewarding procedure. Their support made this journey more enjoyable and entertaining.

Finally, I would like to show my deepest appreciation to my family, who has supported me all the years of my life and helped me progress as a person. Their constant understanding, as well as the sacrifices they made, have allowed me to come to Sweden and KTH and get involved with such an interesting concept as this thesis.

Stockholm, September 2023
Alexios Stournaras
# Contents

1 **Introduction**  
1.1 Problem  .................................. 2  
1.2 Purpose .................................. 3  
1.3 Goals .................................. 4  
1.4 Delimitations .............................. 4  
1.5 Ethics and Sustainability ................. 5  
1.6 Structure of the thesis .................... 6  

2 **Related Work**  
2.1 Attack Graphs and Attack Graph construction methods  .................. 7  
  2.1.1 DSL for AWS  ........................ 9  
2.2 Cyber Security Ontologies and Knowledge Graphs ..................... 10  
  2.2.1 Awspx ................................ 12  

3 **Method**  
3.1 System analysis ............................ 14  
  3.1.1 System analysis of "rollback" scenario  ........ 16  
  3.1.2 System analysis of "cloud breach" scenario .......... 17  
  3.1.3 System analysis of "Attachment" scenario .......... 17  
  3.1.4 System analysis of "EC2 SSRF" scenario ............ 18  
  3.1.5 System analysis of "RCE Web App" scenario .......... 18  
  3.1.6 System analysis of "Codebuild Projects" scenario .... 19  
3.2 Data analysis of systems .................. 20  
3.3 Data Collection and Vulnerability Assessment ..................... 22  
  3.3.1 AWS proprietary tools .................. 22  
  3.3.2 Open source tools ...................... 23  
3.4 Graph Construction ........................ 27  
3.5 Graph analysis and finalization ................ 28  
  3.5.1 Policies, Users and Roles ................ 29
3.5.2 Other kind of AWS related Nodes .......................... 30
3.5.3 Privilege Escalation Nodes ............................ 33
3.5.4 Vulnerability Nodes .................................. 35
3.6 Verification of methodology ............................... 38

4 Design ......................................................... 41
4.1 Complete Algorithm .................................... 41
4.2 Scripts and Data ......................................... 42
4.3 Python code and Graph Construction .................. 42
4.4 Cypher Language and Graph analysis ................. 43

5 Results ......................................................... 44
5.1 Results of IAM-privesc-by-attachment ................. 44
5.2 Results of IAM-privesc-by-rollback .................... 45
5.3 Results of cloud-breach-s3 ............................. 46
5.4 Results of ec2-ssrf ...................................... 47
5.5 Results of rce-web-app .................................. 48
5.6 Results of codebuild-secrets ............................ 49

6 Analysis ......................................................... 53
6.0.1 Comparison with DSL ................................. 53
6.0.2 Comparison with awspx ............................. 54
6.1 Result Analysis of the attachment scenario .......... 55
6.2 Result Analysis of the rollback scenario ............... 56
6.3 Result Analysis of the cloud breach scenario ........ 57
6.4 Result Analysis of the ec2 ssrf scenario ............... 58
6.5 Result Analysis of the rce web app scenario .......... 60
6.6 Result Analysis of the codebuild-secrets scenario ... 61

7 Discussion ....................................................... 64
7.1 Limitations ................................................ 65
7.1.1 Dynamic Solutions .................................... 66

8 Conclusions and Future work ............................... 67
8.1 Conclusions .............................................. 67
8.2 Future work .............................................. 67

References ....................................................... 69
List of Figures

2.1 Flowchart for attack graph construction .......................... 7
2.2 Knowledge graph example in Neo4j .......................... 10

3.1 Summary of our methodology to develop HackerGraph, 
presented in a flow chart ........................................... 15
3.2 First metamodel figure .......................................... 30
3.3 Second metamodel figure ......................................... 31
3.4 Second metamodel figure ......................................... 32
3.5 Query for a creation of a node .................................. 33
3.6 Query for a creation of a relationship between nodes ....... 33
3.7 Python code for detecting any of the privilege escalation 
techniques ......................................................... 34
3.8 Cypher code for creating the privilege escalation nodes ... 35
3.9 Cypher code for creating the connection between privesc 
nodes and policies ............................................... 35
3.10 Cypher code for finding paths between users and privilege 
escalation techniques .............................................. 36
3.11 Bash code for locating credentials and private keys ........ 37
3.12 Queries for a creation of a relationship between User and 
Lambda or S3objecs ............................................... 38
3.13 Query for a creation of a relationship between keypairs and 
Lambda or S3objects ............................................... 39
3.14 Cypher code for creating vulnerability node in case of AWS 
creds. ............................................................... 39
3.15 Cypher code for creating vulnerability node in case of private 
keys. ............................................................... 40

4.1 Architecture of HackerGraph ................................. 42

5.1 HackerGraph result: Cloudgoat ”attachment” scenario .... 45
5.2 Escalation method for Kerrigan ........................................... 46
5.3 HackerGraph result: Cloudgoat ”Rollback” scenario ............ 47
5.4 Policy version v4 from ”Rollback” scenario ......................... 48
5.5 Finding a direct path between user raynor and the escalation node .......................................................... 49
5.6 HackerGraph result: Cloudgoat ”cloud breach” scenario ....... 50
5.7 HackerGraph result: Cloudgoat ”ec2-ssrf” scenario ............. 50
5.8 HackerGraph result: Cloudgoat ”rce-web-app” scenario ..... 51
5.9 Specific bucket information in rce-web-app ....................... 52
5.10 HackerGraph result: Cloudgoat ”codebuild-secrets” scenario . 52

6.1 Steps for the solution of ”attachment” scenario ................... 56
6.2 Steps for the solution of rollback scenario ......................... 57
6.3 Steps for the solution of cloud breach scenario ................. 58
6.4 Steps for the solution of ec2 ssrf scenario ......................... 59
6.5 Steps of the first path for the solution of the rce-web-app scenario .......................................................... 61
6.6 Steps of the second path for the solution of the rce-web-app scenario .......................................................... 62
6.7 Steps of the first path for the solution of the codebuild-secrets scenario ......................................................... 62
6.8 Steps of the second path for the solution of the codebuild-secrets scenario ......................................................... 63
List of Tables

3.1 AWS Contents of each scenario ........................................... 16
3.2 AWS Tools for Data Collection and Threat Analysis ........... 24
3.3 Open source Tools for Data Collection and Threat Analysis ... 25
3.4 Problems with other open source tools ................................. 28

6.1 Comparison of tools for each scenario, in regards to finding attack paths .................................................. 55
Chapter 1

Introduction

In recent years, cloud computing has become increasingly popular as businesses continue to recognize the benefits of this technology. According to Radware’s 2022 report, 99% of organizations now deploy applications in public cloud environments. This immediately causes concerns about the possible dangers and attacks that these environments are vulnerable to. Despite advancements in cloud security practices, cloud environments continue to face persistent vulnerabilities and remain a target for malicious attacks. According to the Thales Cloud Security study, 45% of organizations have experienced a data breach or failed an audit involving data in the cloud. One example of a breach is the one of FlexBooker, which happened in one of the biggest cloud providers, the Amazon Web Services. Due to misconfigurations in AWS buckets, hackers were able to steal data from 3.7 million users, including considerable Personally Identifiable Information (PII) data, IDs, hashed passwords, and even partial credit card numbers!

In order to avoid such further events, AWS has already developed a lot of tools like AWS Cloudwatch and AWS Cloudtrail, to help security and cloud engineers to monitor their systems. Furthermore, according to the shared responsibility model, AWS is responsible for securing the facilities, physical security of hardware, network infrastructure, and the virtualization infrastructure of the platform. On the other hand, security engineers themselves are responsible for securing their applications and the

---

2 https://cpl.thalesgroup.com/cloud-security-research
3 https://www.vnmentor.com/blog/report-flexbooker-leak
4 https://aws.amazon.com/cloudwatch/
5 https://aws.amazon.com/cloudtrail/
data they store on the cloud.

To help them achieve those goals, third-party entities like the National Institute of Standards and Technology (NIST), have developed cyber security frameworks [2], that help engineers identify and handle cyber security risks. Moreover, companies like MITRE have been investigating ways to develop an ontology for the cybersecurity domain environments[3] [4]. The same company has also developed the MITRE cloud matrix\(^1\), which presents attack tactics and techniques, relative to all the major cloud platforms, such as privilege escalation techniques, defense evasion tactics, credential access methods, etc. Blogs like the Rhino security blogs\(^2\) and hackingthe.cloud \(^3\) are also utilized by security engineers to learn about new techniques and ways to defend against them. More resources about vulnerabilities, attack tactics, and methods are stored in databases like the Common Vulnerabilities Enumeration (CVEs) database\(^4\) and the US National Vulnerability Database (NVD)\(^5\).

Security engineers utilize all of the aforementioned resources to create different methods of finding vulnerabilities in their systems and thus, better securing them. One of those methods is attack graphs, that showcase attack vectors in the topologies of interest, in order to check what kind of steps an attacker can take in a vulnerable network [5].

However, current tools that construct attack graphs are inadequate in describing the topology of a cloud system and its vulnerabilities. This is evident from the fact that no matter what tools have been developed, attacks still happen, with the most recent being against the servers of Pegasus Airlines, hosted in AWS, in May of 2022.\(^6\). The DSL for AWS [6] is one of the few tools that are doing a comprehensive job of analyzing an AWS cloud environment, but it still does not find all of the AWS-specific exploitation methods.

### 1.1 Problem

The problem then is not the lack of resources or tools, but the lack of a holistic method for security engineers to assess their systems. In particular, while there are tools that use the aforementioned databases that contain a lot of cloud-specific methods of attack, they rarely capture the underlying

---

1. [https://attack.mitre.org/matrices/enterprise/cloud/](https://attack.mitre.org/matrices/enterprise/cloud/)
2. [https://rhinosecuritylabs.com/blog/](https://rhinosecuritylabs.com/blog/)
3. [https://hackingthe.cloud/](https://hackingthe.cloud/)
4. [https://www.cvedetails.com/](https://www.cvedetails.com/)
5. [https://nvd.nist.gov/](https://nvd.nist.gov/)
infrastructure and the relationships between the entities. As such, a lot of attacks that could happen in a system remain unnoticed. In particular, as one can see from one of the cloud security blogs like Hackerone\(^1\), lots of attacks use different relationships and connections between the cloud entities and, due to the vast amount of different APIs cooperating, showing the relationships between them becomes extremely complicated. For example, when assessing a cloud system, one has to take into consideration the different Identity and Access Management (IAM) permissions that exist, that might allow an attacker to pivot through the system. Knowledge graphs \(^7\), a graph-based knowledge representation framework that captures relationships, concepts, and attributes within a particular domain, could be utilized to help provide more accurate and detailed information. However, even tools like the awspx \(^2\), that have been designed to showcase in detail the infrastructure of an AWS system, lack a lot of attack techniques and do not take into account the most famous methods of cloud hacking, according to the Pandemic 11 \(^3\), like checking for inline credentials or looking at misconfigurations on codes that are publicly available. This means that security engineers lack a method of having a holistic view of their environment, that showcases both the complicated relationships that exist between their systems, as well as possible vulnerabilities that might be connected with those relationships.

1.2 Purpose

The purpose of this work then is to improve the security assessment of an AWS system, by creating a graph construction tool called HackerGraph, that will focus on two things. Firstly, it will focus more on the knowledge extraction of both the infrastructure details of the system, as well as attack techniques and methods, specific to the AWS platform. In contrast to the state-of-the-art tools that exist, Hackergraph combines various different tools that can collect AWS metadata, inline credentials, and hacking techniques so that we can create more accurate relationships and connections between AWS entities and attack tactics. Secondly, it will demonstrate all of this detailed information, along with the relationships and connections created by it, in one single knowledge graph. This will be created by using common programming languages like Python and Cypher.

\(^1\)https://www.hackerone.com/knowledge-center/penetration-testing-aws-practical-guide
\(^2\)https://github.com/WithSecureLabs/awspx
\(^3\)https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-pandemic-eleven/
As such, through this tool, a faster and more efficient method of collecting data is provided, as well as a detailed and organized way of depicting them. Security engineers can thus much more easily and completely assess their environments, and change their policies, entities, and even storage techniques, accordingly. HackerGraph also offers modularity and scalability, since it makes it easy to add more tools in it and search for even more data, depending on a security engineer’s current needs.

### 1.3 Goals

In order to achieve this purpose, this thesis has the following goals:

1. Find the appropriate tools to gather the data needed to fully depict the infrastructure details and the vulnerabilities of an AWS system.
2. Find the best methods of utilizing those data so that they can be transformed into a graph.
3. Combine all the different methods described above to create a coherent knowledge graph, for any AWS system given.
4. Assess the effectiveness of a knowledge graph to provide appropriate information about the system, as well as security weaknesses.
5. Compare its effectiveness with other state-of-the-art tools, like awspx and DSL for AWS.

### 1.4 Delimitations

It is important to note, that our tool will only examine cloud-specific methods of attack. This means that it will focus on the following three kinds of vulnerabilities:

1. Finding inline AWS credentials of other users or roles that might exist in plain sight in various AWS entities.
2. Finding private keys of keypairs that provide access to other AWS entities, such as instances.
3. Analyzing the policies of an AWS system for misconfigurations. Specifically, HackerGraph will check the permissions that are part of a policy and check if there are any combinations that lead to privilege escalation techniques.

While we could incorporate tools that discover traditional network attack paths in our systems, like exploiting a web page vulnerability to access an instance, we chose not to. This work is meant to present how the relationships between the different cloud entities can be used by a hacker to pivot in a system, and not how a hacker can access the system in the first place. We check every entity of an AWS system, as well as possible misconfigurations a security engineer might have done while he is setting it up. We do not check for example about misconfigurations in a web page that runs on an EC2 instance, with which one can perform an RCE for example. We take for granted that an attacker can access an instance with non-AWS-specific attacks and we examine how can he pivot and continue to attack the system afterwards. As such, we will focus only on attacks that suppose that some credentials have already been exposed and the attacker got access to the system. Moreover, we will not take into account attacks that use the metadata server of an AWS platform. We can neither visualize the metadata server nor gather data about how to access it through an AWS system. However, if, for example, an attacker can get some credentials of a role from the metadata server, with which he can access EC2 instances, buckets, objects, etc, HackerGraph will still show those connections. It will show that the role exists, and how it is connected or is part of the rest of the system, but it will just not show that the credentials of the role exist in the metadata server.

1.5 Ethics and Sustainability

In regards to developing a knowledge graph for analyzing vulnerabilities and safeguarding cloud systems against hacking, ethical considerations are of utmost importance. Upholding ethical standards at every stage, ranging from data collection to analysis and decision-making is crucial. Protecting the privacy and confidentiality of sensitive information, acquiring informed consent when applicable, and adhering to relevant laws and regulations are fundamental aspects. Furthermore, ensuring transparency and accountability in the design and implementation of the knowledge graph is essential, providing stakeholders with a clear understanding of the underlying algorithms, methodologies, and potential biases. By embracing an ethical
framework, the project fosters trust, fairness, and responsible data usage, ultimately contributing to the creation of a more secure and resilient cloud environment. Specifically for our case, our knowledge graph respects the ethical considerations related to cloud security. Our graph does not provide exact attack paths of the system but rather is a tool for security engineers to assess their own infrastructure. It helps protect sensitive information, such as credentials and private keys, and provides transparency about the permissions of each user of an AWS system.

Furthermore, it is equally vital to consider sustainability aspects while developing a knowledge graph for vulnerability analysis. This entails taking into account the project’s environmental impact across its entire lifecycle. By optimizing resource allocation, minimizing energy consumption, and employing efficient algorithms and infrastructure, the project can actively contribute to reducing its carbon footprint and promoting sustainable practices. Additionally, the knowledge graph can empower cloud systems to identify and proactively address vulnerabilities, thereby mitigating potential breaches and minimizing the environmental ramifications associated with cyberattacks. By integrating sustainability principles into the knowledge graph’s design and operation, the project aligns with the broader objective of establishing resilient and environmentally conscious cloud systems. In our case, we provide a method for cloud engineers to test their systems for unnecessary use of cloud resources, such as permissions, instances, etc. As such, it supports sustainability, since cloud engineers can use it to test for cloud resources being wasted. Moreover, by displaying unprotected keys and credentials, it allows for protection against an intrusion event, that could possibly lead to many environmental problems, depending on the company that is attacked.

1.6 Structure of the thesis

The structure of the rest of the thesis is as follows. Chapter 2 presents relevant background information about attack graphs, knowledge graphs, and tools used to create them. Chapter 3 presents the methodology and method used to solve the problem. Chapter 4 presents our whole implementation of the system. Chapter 5 presents our results demonstrated on set cases of systems designed to be vulnerable. Chapter 6 presents the analysis of the results and their comparison with the state-of-the-art tools. Finally, Chapter 7 offers a discussion based on the previous results and analysis and then, Chapter 8 presents our conclusions of the thesis and possible future work.
Chapter 2

Related Work

2.1 Attack Graphs and Attack Graph construction methods

In the area of cyber security, attack graphs are a standard way of modeling a system. Attack graphs are designed to represent the abstracted network topology with a directed acyclic graph. One of the main application scenarios is for network vulnerability analysis. The vertices of attack graphs can be related to elements such as host, authority, vulnerability, service, and even some network security status, depending on the attack behavior analysis requirements. Unlike the diversity of vertex, the edges in attack graphs generally indicate the perpetration of attacks [8]. A flowchart of the procedure required to construct an attack graph is demonstrated in Figure 2.1.

Figure 2.1: Flowchart for attack graph construction

Philips and Swiler [5] [9] are considered to be the pioneers of the attack graph method by using probabilistic edge weights between the nodes. They use as data input attack templates, a configuration file, and an attacker profile. Attack templates represent generic (known or hypothesized) attacks including conditions, such as operating system version, which must hold for the attack to be possible. The configuration file gives detailed information about the specific system to be analyzed including the topology of the network and configuration of particular network elements such as workstations, printers, or
routers. The attacker profile contains information about the assumed attacker’s capabilities, such as the possession of an automated toolkit or a sniffer as well as skill level.

Another landmark method in attack graph construction is the Topological Vulnerability Analysis tool developed in [10], which analyzes vulnerability dependencies and shows all possible attack paths in a network. Extensive information about known vulnerabilities and attack techniques are collected and correlated with network vulnerability information gathered from the systems. Along with its follow-up, Cauldron [11], and NetSecuritas [12], these three approaches attempted to suggest security controls. NetSPA [13] has also been developed, which used network models and external scanning software for data collection. It was later extended with client-side vulnerabilities in [14] and visualizations in GARNET [15] and NAVIGATOR [16]. MulVal [17], a network security tool analyzer tool is another graph constructor tool developed. It combines vulnerability information from databases provided by the bug-reporting community with configuration information of each machine in the network, as well as of the network itself. ZePro [18] and k-Zero Day Safety [19] are more focused with zero-day attacks, which exploit vulnerabilities that have not been disclosed publicly [20]. The former analyzes which attack paths likely contain zero-day exploits, while the latter estimates how many zero-days would be required to penetrate a network. All of these tools however have not been tested in cloud environments. They might be able to assess a cloud environment but there are significant differences between classic network topologies of routers, switches, and hosts and AWS topologies of Virtual Private Clouds (VPCs), EC2 instances, and external storage with S3 buckets, for that to happen.

In more recent years, the authors in [21] use big data and model different types of security requirements into the context of an attack graph, containing business process targets and critical assets identification, configuration items, and possible impacts of cyber-attacks. Specifically, they use a comprehensive database of threat intelligence, vulnerabilities, configuration issues, and other security hygiene issues such as clear text credentials that map them to network entities. Unfortunately, there is no mention in this paper of how this system would handle an AWS system. There is no mention of specific cloud-attack databases and no experiment based on AWS infrastructure information being provided. In [22], an algorithm is used to create a graph that uses existing model-checking tools, an architecture description tool, and the authors’ own code to generate an attack graph that enumerates the set of all possible sequences in which atomic-level vulnerabilities can be exploited.
to compromise system security. In order to manage that in microservice architectures, they basically model these architectures into similar normal network ones. Thus, they adapt existing attack graph generation methods from the computer networks field to the microservices ecosystem. The modeling is an intriguing idea, but again, due to the differences between a traditional network and a network of microservices, the results might be incorrect in the case of an AWS system. This is reinforced by the fact that the underlying cloud infrastructure where the microservices run is not considered. In AGBuilder [23], Artificial Intelligence is used for automatically generating, updating, and refining attack graphs by using textual descriptions of vulnerabilities found in the CVE system or the NVD system to automatically generate attack graphs. In [24], the authors create an attack graph for Internet of Things sensors. They use CVSS vulnerability information along with network information to manage their goal. In both of these papers though, the databases that are used are quite incomplete in regards to cloud architecture vulnerabilities. Moreover, some vulnerabilities that alone seem insignificant and are not being considered, could become quite dangerous when considering the whole AWS network.

A general problem with all of these newer papers is also that they do not really discuss how the data are acquired. Most of them actually do not even mention the data acquisition procedure and the ones that do, use tools like NMAP ¹ or Nessus ², which are not cloud-specific and do not really indicate the intricacies of an AWS system.

### 2.1.1 DSL for AWS

The most important attack graph construction tool, however, that actually focuses on the AWS platform and creates an attack graph of specifically AWS systems is the DSL for AWS [6]. This tool uses the Meta Attack Language (MAL) [25] to develop an AWS-specific threat modeling and attack simulation strategy. MAL is a culmination of previous efforts of creating a threat modeling language, presented in [26],[27],[28] and [29]. It thus can model AWS systems and assess their security by creating and traversing attack graphs. Users can manually or automatically provide AWS system configuration and DSL reasons with both traditional software vulnerabilities and cloud-specific attacker tactics and techniques, to perform the assessment. DSL specifically mentions however, that the information collection and the modeling of the

¹https://nmap.org/
²https://www.tenable.com/products/nessus
AWS systems that were tested for validation happened through Vanguard \(^1\). A comparison of the results of this attack graph tool will be made with our own HackerGraph in Chapter 6.

### 2.2 Cyber Security Ontologies and Knowledge Graphs

As discussed in Chapter 1, more knowledge of the infrastructure of an AWS system is required. Also, AWS-specific attack tactics must be correlated with it, in order to have a more effective vulnerability assessment of a system. Based on the previous discussion, most of the aforementioned attack graph tools didn’t really delve into these two factors. Moreover, in the face of large-scale networks and massive data, the existing methods cannot meet the actual needs in terms of knowledge representation and reasoning. The knowledge graph for cyber security provides a new solution to the above problem[30]. An example of such a graph is presented in Figure 2.2.

![Figure 2.2: Knowledge graph example in Neo4j](image)

The foundation of knowledge representation in the field of cyber security relies on utilizing an ontology within the cyber security knowledge graph. This enables the expression of knowledge in a structured and relational manner, allowing for the visualization of information in a graphical format [30].

Ontology serves the purpose of describing concepts and their interrelationships within a specific field or even across broader domains. These concepts and relationships are defined in a universally accepted and unambiguous manner, facilitating effective communication between humans and machines.

\(^1\)https://www.foreseeti.com/securicad-vanguard-for-aws/
This shared understanding then enables seamless interaction and collaboration between humans and automated systems [31]. By leveraging the ontology-based knowledge representation approach, it becomes possible to effectively integrate diverse data from multiple sources within a particular domain. Additionally, the utilization of a comprehensive ontology language enables the realization of knowledge reasoning and classification. This combination empowers the system to make informed deductions and categorizations based on the available knowledge [32].

The development of the cyber security ontology aims to seamlessly integrate diverse data resources related to cyber security. Its primary objective is to efficiently organize and leverage knowledge within the cyber security domain, ultimately providing crucial support for assessment and analysis of cyber security. Extensive research has been conducted on knowledge bases built upon a singular ontology, with notable advancements made in the vulnerability description framework, which represents the most mature aspect of this field [33]. This directly promotes the development of vulnerability databases, like the ones mentioned in Chapter 1.

In the context of the knowledge graph, ontology serves as the central component for knowledge management. The research findings related to ontology provide a solid theoretical foundation for governing entities, relationships, as well as the associations between objects, such as types and attributes, within knowledge graphs. These insights contribute to establishing structured and standardized frameworks for organizing and representing knowledge in a coherent manner [34]. In short, an ontology is the base for creating a knowledge graph. There are various mature structured knowledge bases in the cyber security domain. Jia et al. [35] suggest a framework for building a cyber security knowledge graph, that includes a five-tuple model of security knowledge base (concepts, instances, relationships, attributes, and rules) and a cybersecurity ontology (assets, vulnerabilities, and attacks). However, the article only implements the construction of cyber security ontology and entity extraction and does not show the construction process of a complete knowledge graph.

YHSAS [36] is a situation awareness system for backbone cyber security and large-scale network environments such as large network operators, large enterprises and institutions. This system applies knowledge graphs to large-scale cyber security knowledge representation and management, enabling obtaining, understanding, displaying, and predicting future development trends of the security elements that cause the network situation to change. Zhu et al. [37] proposed a cyber-attacks attribution framework, based on the
constructed cyber security knowledge graph. This allows them to track the attack source in the air-ground integrated information network and solve the difficult problem of locating the attack source.

Wang et al.\cite{38} proposed an integrated intelligent security event correlation analysis system. The system integrated the network infrastructure knowledge base, vulnerability knowledge base, cyber threat knowledge base, and intrusion alert knowledge base into the cyber security knowledge graph to support correlation analysis of security events. MITRE has developed a situation awareness system called CyGraph \cite{39}, primarily designed for network warfare task analysis, visual analysis, and knowledge management in the field of cyber security. CyGraph effectively consolidates fragmented data and events, constructing a comprehensive knowledge graph using a unified graphical representation of cyber security. This system enables various capabilities, including the analysis of attack paths, prediction of critical vulnerabilities, correlation analysis of intrusion alarms, and interactive visual queries.

As for AWS specifically, the work of Ammi \cite{40} does a great job of collecting data in XML files and using it to create a knowledge graph to do threat analysis. In order to achieve this, the authors of this paper use the Neo4j graph database \cite{41}, which allows them not only to store the data in graph format but also process them by doing queries that will enable them to look for attack paths between the entities. However, there are still some gaps in their data accumulation. For example, they are using pre-gathered data and don’t really showcase how they actually accumulate them. Furthermore, instead of using logs, it would be better to actually use different tools that have been developed for data accumulation and threat analysis.

\subsection*{2.2.1 Awspx}

The tool that comes the closest to the purpose of this thesis, is awspx\footnote{https://github.com/WithSecureLabs/awspx}, since it gathers information about the AWS system infrastructure and some well-known privilege escalation methods and presents them in a knowledge graph. In particular, it uses the AWS CLI \footnote{https://aws.amazon.com/cli/} with a profile with security audit permissions or administrator access (provided by the user) and gathers information about EC2 instances, S3 buckets, Lambda functions and the IAM API. It then checks the 21 privilege escalation methods, as described in
hackingthe.cloud \(^1\), and checks if any user of the AWS system can perform any of them. After that, it presents all of the information gathered in a knowledge graph. The results of this tool will also be compared to the results of HackerGraph in Chapter 6.

\(^1\)https://hackingthe.cloud/
Chapter 3

Method

As mentioned before, this thesis aims to create a tool, HackerGraph, in order to apply the use of ontologies and knowledge graphs in AWS. It will improve on the vulnerability assessment of AWS systems, compared to the literature described before, by focusing on creating better ontologies, both for the AWS system infrastructure, as well as the AWS-specific vulnerabilities and attack methods. In contrast to the related work, we will extensively analyse how we gather our data and we will take into account vulnerabilities like inline credentials and private keys, that tools like AWSPX do not. Finally, similarly to the work of Amni [40], we will use queries and the knowledge graph itself to process our data and do vulnerability analysis on our system. Our vulnerability analysis though, is not going to showcase the actions an attacker can take in the traditional way, i.e show how to perform an attack. What our vulnerability analysis will present is the vulnerabilities that exist between the system, and the connections they have with other entities, in the form of relationships. This will be proven to be enough, according to the results analysed in Chapter 6.

In order to achieve our goal, we followed a specific methodology to develop HackerGraph, a summary of which is depicted in Figure 3.1. To better justify our methods, the methodology is going to be mapped to the steps required to create an attack graph, as presented in Figure 2.1.

3.1 System analysis

First and foremost, we had to create an AWS account in order to be able to install and test vulnerable by-design systems. Afterward, we installed Cloudgoat. Cloudgoat 1, developed by the Rhino Security team, is a variety of

---

1https://rhinosecuritylabs.com/aws/introducing-cloudgoat-2/
vulnerable-by-design AWS systems, created to practice penetration testing on AWS. There are 13 systems that can be installed, but only those that are being deployed in the DSL for AWS paper [6] were tested. DSL for AWS is one of the most state-of-the-art tools for creating an attack graph for AWS systems and as such, we are going to compare our results to the scenarios they tested. To truly grasp how an attacker can use misconfigurations and relationships between entities to exploit vulnerabilities, and thus what kind of data and metadata we need to collect, we manually solved these scenarios ourselves. We used the AWS command line interface (CLI) to accomplish that. Table 3.1 displays all of the AWS entities in each scenario. From that table, and from the small description of each scenario that follows, one can understand that there are very few entities in each scenario, compared to real life AWS environments, and all of these entities have a part to play in the solution of each scenario. As such, we choose to include every entity of each scenario and their relationships with each other in our Hackergraph. In real life scenarios, where systems that could entail millions of nodes would exist, we would still entail every entity for data gathering but we would only showcase users and their paths to vulnerabilities, for display reasons. This will be analyzed in detail in Section 3.5. The following subsections describe the data and the relations required to be gathered, in order to follow the attack paths described in Chapter 6 and solve the scenarios.
### 3.1 Method

#### 3.1.1 System analysis of "rollback" scenario

When we have a profile available like in this case (and most of the other scenarios), we try to check what permissions the user has and hence, what API calls the user can perform. As such, we look for details for our user with the command "IAM get-user –profile raynor" so we can get his username, userID and ARN. Then we look for its attached policies, to see what we have access to. To do that, we use a command that contains the username of raynor. Thus, we get the policy names and ARNs. With the policy ARN and the "get-policy" command we get details like PolicyId, name, path, version id etc. After that, we can see that we can get all of the policy versions with the same ARN and ID. We get details for each version ID and we find one that gives administrator access. We also saw in our default policy that one permission allows us to set a policy version. This is a privilege escalation technique. As such, we understand that we can set as a default policy the one that gives us administrator access. This scenario demonstrates the importance of the relationships between policies and users, as well as policies and their versions, based on the version ID.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>AWS Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priv. esc. by Rollback</td>
<td>1 IAM User, 1 Policy and 5 policy versions.</td>
</tr>
<tr>
<td>Cloud breach S3</td>
<td>1 Policy, 1 Role, 1 EC2 Instance, 1 Instance Profile, 1 S3 bucket and 4 S3 objects.</td>
</tr>
<tr>
<td>Priv. esc by Attachment</td>
<td>1 IAM user, 3 Policies, 2 Roles, 1 EC2 Instance and 1 Instance Profile.</td>
</tr>
<tr>
<td>EC2 SSRF</td>
<td>3 IAM Users, 4 Policies, 2 Roles, 1 EC2 Instance, 1 Instance Profile, 1 Lambda function, 1 S3 Bucket and 1 S3 object.</td>
</tr>
<tr>
<td>RCE Web App</td>
<td>2 IAM Users, 4 Policies, 1 Role, 1 ELB, 1 EC2 Instance, 1 Instance Profile, 3 S3 buckets, 5 S3 Objects and 1 RDS Database.</td>
</tr>
<tr>
<td>Codebuild Project</td>
<td>2 IAM Users, 3 Policies, 3 Roles, 1 Codebuild project, 1 Lambda function, 1 EC2 Instance, 1 Instance profile and 1 RDS Database.</td>
</tr>
</tbody>
</table>
3.1.2 System analysis of ”cloud breach” scenario

Here, we can access the metadata server through the IP of the instance that we are given. The most important thing from the metadata server was the IAM folder that contained the role that the instance had attached to it. In AWS, one cannot attach a role directly through an instance but has to use an instance profile instead, that has this role attached. This role has a policy attached to it with a set of permissions, that define what APIs the role is allowed to access and work with. This signifies the importance of the relationship between role and instance profile. We understand then, that through assuming that role for ourselves, we can access what the instance accesses. We then can get the security credentials for that role, where we can configure a new profile with the access key, the secret access key, and the security token, to assume that role. With this role, we check again what kind of permissions we have. We check the S3 API of course because we know that this scenario has one bucket, and we can get it and its objects. This means that the role, through its connection with a policy and its permissions, has a connection with a bucket and its objects. The object of course has as metadata the ID of the bucket that is connected to, which basically demonstrates the connection between objects and buckets. One of the objects contains our goal.

3.1.3 System analysis of ”Attachment” scenario

Here we are provided with a user called Kerrigan. We again try to list policies for our user and we fail, thus we try to list other things, like roles. We see that we have roles that can work with instances, one called meek and one called mighty, so we can understand that the mighty one might give stronger privileges. Then, we can try to list the instances and the instance profiles as well. We describe the instances and we see the details of an instance where we find a security group that has to do with ssh, along with its IP. As such, we try to log in to that instance. We can’t log in since we don’t have the key, but the action itself is allowed. That means we can try to create a new instance with a new keypair, that we will also create, to have access to that instance. By listing the instance profiles, we find one that has the meek role attached. It is important to note here that this instance profile is not attached to the instance mentioned before, since there is no ”Profile” in the attributes of the instance when we described it. Since we can work with roles, instances and instance profiles, we can try to create a new instance with an instance profile that has the stronger role, in order to get its permissions. This is another privilege escalation technique. As such, we try to remove the old role from the instance.
profile (which of course it works) and then we add the mighty role to that. We do that with the AWS CLI by using the names of the instance profiles and the names of the roles. After that, by using data from the instance, in particular the image ID, the instance type, the key-name we created, the subnet id and the security group id, along with the new instance profile, we create a new instance with the mighty role. Then, we log in to the new instance, we install the AWS CLI and we realize we have administrator access.

3.1.4 System analysis of "EC2 SSRF" scenario

Again, we start with the IAM service. We get the username of Solus and we try to see where we have access, by trying to list permissions, users, etc. We can only list lambda functions, and we see that we get a function with credentials inside its environmental variables. This is one of the two cases regarding credentials in our delimitations. We configure a new profile with those credentials, and we do the same. We can describe the instances, where we find one instance from which we can get, among other details, its public and private IP. It's called the "SSRF" scenario, so we try to go to its IP with a browser and access the metadata server. We can then try to find credentials that exist in this metadata server and get the credentials for a role, just like in the cloud breach scenario. By assuming that role, with its access key, secret access key and token, we can list buckets and their objects. In short, the instance that we accessed with an "SSRF" attack, had an instance profile attached, which had a role that allowed us to access buckets. If we use a different user that has administrator access, in order to get more information about the instance, we will truly see that the instance has a property named "profile", with a value of the ARN of the instance profile. In turn, the profile has a "Role" attribute with the ID of the role it is using. The role of course, through its policy and its permissions, allows us to access the buckets. In the bucket's objects, we find credentials (same case of our delimitations as before) for a third user. Again, we try different APIs as that user (through different permissions), and we see that we can invoke the lambda function, which is the goal of this scenario.

3.1.5 System analysis of "RCE Web App" scenario

This scenario has two users. We will start with user McDuck. Again, we check what our permissions allow us to do, and we find out that we can list buckets. We find three buckets, but we can only list the objects from only one, where we find a pair of SSH keys. We then list instances and describe them,
and we discover that we can use that private key we found to connect to the
instance (the other case in our delimitations regarding credentials). We know
that because when we describe that instance, we see that it has a keypair ID,
thus we know to try the private key on that instance. Moreover, by describing
the instance we get its IP, and we know where to connect. We also discover
that the instance has an instance profile attached and thus, a role. By installing
the AWS CLI on the instance we can now check what that role, through its
permissions, does. We see that we can get info on a bucket that was not
visible before. In this bucket, we see an object, that has some credentials for a
database. We check if we can list a database, which of course we can (through
the instance) and we access the database with these credentials, where we get
the secret text.

The other user is user Lara. Again, we check our permissions, and we
can list buckets. We can access a different bucket now, that has logs for a Load
Balancer. We also check the load balancer API, which gives us a webpage that
tells us basically to look through the logs for an admin page. We go through
the logs, and we do find an admin page. Through it, we perform a remote code
execution and we get access to the instance that is hosting the web page. Of
course, this instance is the same as before, and we can follow the same steps
as before after accessing the instance. Another way to continue though, is to
access the metadata server, however, we won’t bother with that part since we
described in our delimitations that we won’t handle the metadata server aspects
of the scenarios.

3.1.6 System analysis of ”Codebuild Projects” scenario

This scenario starts with the user Solo. We again check what we can do with
the user, and we find that we can list Codebuild Projects. From there, we can
get the details for a project, where we see that we have credentials for another
user, called Calrissian. After assuming his identity, we check again what we
can do, and we find out we can list the RDS databases and instances. We
cannot access it, but we have the permission to create a snapshot of it, and a
new RDS instance through that snapshot. To do that, we just need the database
ID and the subnet it belongs to. Moreover, we need to find a security group
that we have access to, to put the new database in and be able to access it.
We do that by getting the security group of the EC2 instance we can describe.
Thus, we create the new RDS database in that subnet, and then by changing
its password, we can access it and get the sensitive data.
Another path is the following. Solo can also list SSM parameters. In the parameters of the account, we have a pair of keys. We can then describe EC2 instances, where we see a "Keypair" property and its IP. Of course, we try to connect to that instance IP with that key which succeeds. From there, we can access the metadata server and get the instance profile’s IAM keys and assume its role. As such, we can use the instance’s role to access and list lambda functions and RDS databases. In the lambda environment variables, we find again credentials that we can use on the RDS database we discovered, and thus, get the secret that is our goal.

3.2 Data analysis of systems

From the descriptions of the scenarios above, we observed some important details. First of all, we noticed that every single AWS entity, be it a policy, a role, or an instance, has a unique ID. This would allow us to create unique nodes of each type in the knowledge graph, based on that ID. Next, we realized that policies, and particularly the permissions they include as metadata, were the most important step to figure out in each scenario. Policies can be attached to roles, users, or groups (although none of the scenarios contained groups and policies attached to them). In the metadata of the policy, one could see its unique ID, its Amazon Resource Name (ARN) \(^1\) which is another unique identifier, its name and the permissions. Similarly, for roles and users, the most important metadata were their ARNs, their IDs and their names. By using the ARNs of the policies and using the AWS IAM command "list-entities-of-policy", we could check which role or user had which policy, and thus, what permissions. That is of utmost importance, since in this way, we could understand what other entities each role or user could access through the policy’s permissions. Moreover, policies could have more than one version, with one being the default being used and the others being older versions that could be switched to become the default version again. All of those entities belong to the IAM API, and we had to gather all of this information, in order to solve every single scenario.

The next important observation was that almost all of the scenarios had to do with specific APIs. Apart from the IAM API that was already discussed, the others were the S3 buckets, the EC2 instances, the Lambda functions and the RDS databases. These 5 APIs are also the most popular and most used APIs of AWS. As mentioned in section 3.1, the details and metadata from those entities

provided almost all the information required to solve each scenario. As such, we had to gather as much data as we could about them:

1. In regards to EC2 instances, the most important metadata that we had to use in most of the scenarios was the name, the id, the image-id, the subnet-id, the security groups, the private and public IPs they might have, their state, and particularly, the instance profile that might have been attached to it and the keypairs that they might have.

2. In regards to the instance profiles of instances, the most important metadata were of course the name, the ARN and the ID, and the Id of the role that was attached to it. It is important to note here that an instance profile must always have a role attached to it. This happens because instance profiles are basically the method that is used to give instances permissions to access other APIs.

3. In regards to buckets, we only used their Ids. What was of value were the objects they contained, which might have contained credentials, access keys or other important information. To get that, we only needed the name of the object and the name of the bucket it was in.

4. In regards to databases, not a lot were required apart from the ARN, the name, the ID and the Subnet they belonged to. Anything that had to do with databases was mostly based on the kind of the database itself, and not on AWS.

5. In regards to Lambda functions, the most important metadata were the ARN, the name, the description of the function, the ID and the role they might have had attached.

One scenario also had the Codebuild Projects and the AWS Systems Manager (SSM) APIs involved, but the only thing needed from them was, respectively, the codebuild project itself or listing the parameters of the SSM API. In both cases, what was important was not the metadata of those entities, but the contents themselves, which were AWS credentials.

This leads us to the next observation, which had to do with the vulnerabilities of the system. All of the scenarios had to do with either privilege escalation techniques, due to policies giving users or roles more permissions than needed, or with inline credentials and keys being in unguarded places, like environmental variables of Lambda functions and accessible objects from S3 buckets (usually a combination of both). This
meant two things: Firstly, we needed a way to gather the credentials from the functions or the objects. Those were not visible on the metadata mentioned before, so we needed a way to show more details about those Lambdas and bucket objects, in order to gather those credentials and keys. Secondly, we had to check the policy permissions for the 21 privilege escalation techniques mentioned by the Rhino security labs. Those 21 techniques cover every scenario where a vulnerability exists because of unnecessary permissions being provided to a user (or a role).

By gathering all of the aforementioned information, we had everything we needed to show in a knowledge graph all of the connections and relationships that would showcase how an attack could happen in every scenario.

It is important to mention here again that, while solving the scenarios, one could not gather all of this information from one scenario user alone. For example, a user might not have had the right permissions to list the policies that exist in the system or see the roles that are attached to him. To gather all of the available, aforementioned information, we needed a user that was independent of the scenarios and had either security audit permissions or administrator access permissions. In our case, we could use the administrator user that we had created, in order to build the cloudgoat scenarios. Also, we decided to start building HackerGraph with the "Iam-privesc-by-attachment" scenario. This scenario struck the perfect balance between not being too simple and not being too hard and it had enough AWS entities and policies to work with. It was easier to create data collection scripts and create a graph from it, since it had enough information to continue, relative to the simpler scenarios. Also, it didn’t have as many pieces of information as to become too convoluted for a methodology creation, compared to the more difficult ones.

### 3.3 Data Collection and Vulnerability Assessment

Afterward, we had to find the most appropriate tools that were most suitable to gather that information.

#### 3.3.1 AWS proprietary tools

As mentioned in Chapter 1, AWS has a lot of proprietary tools developed to handle the AWS system and its monitoring. First and foremost, the AWS
command-line interface (CLI) can be quite effective in providing us with the appropriate data. We can get the contents of an S3 bucket, list all Identity Access Management (IAM) users and roles, as well as get network-related information. Although there are some commands that are more important and used more than others\(^1\), the AWS CLI is a very complicated CLI. In order to get all of the information we need, we might have to use extremely long and convoluted commands and thus, making it a last resort of tools.

Moreover, there are other aws proprietary tools that have been developed for threat analysis and data collection. One of them is AWS Resource Groups, which group resources based on tags or other criteria, making it easy to see which resources are related to each other as well as allowing to programmatically retrieve information about the resources in a group and their relationships. AWS Cloudtrail logs all API calls made to AWS services, including calls made to create, modify, or delete resources, and can be used to identify which resources were created and modified and the relationships between themselves. This can also be done with the AWS Config technique, which also showcases the impact of a change in one resource on the rest. Finally, AWS Management Console and AWS Neptune allow for the representation and visualization of the whole system as a graph.

Additional AWS tools are presented in Table 3.2, for the sake of completeness.

### 3.3.2 Open source tools

Of course, there are open-source and free-to-use tools that have also been developed to perform those aforementioned tasks. GraphKer\(^1\) can be utilized since it represents every public record of every vulnerability that is provided by the MITRE and NIST attack frameworks, discussed in Chapter 1, and thus can provide a lot of attack methods to our graph. Prowler\(^2\) and similarly ScoutSuite\(^3\), automate the process of performing security checks.

---

1. [https://github.com/amberzovitis/GraphKer](https://github.com/amberzovitis/GraphKer)
2. [https://github.com/prowler-cloud/prowler](https://github.com/prowler-cloud/prowler)
3. [https://github.com/nccgroup/ScoutSuite](https://github.com/nccgroup/ScoutSuite)
Table 3.2: AWS Tools for Data Collection and Threat Analysis

<table>
<thead>
<tr>
<th>AWS Tools</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS CloudTrail</td>
<td>Captures API calls and events, providing detailed logs for auditing, compliance, and security analysis.</td>
</tr>
<tr>
<td>Amazon GuardDuty¹</td>
<td>Uses machine learning to detect potential security threats, unauthorized access, and network vulnerabilities in your AWS environment.</td>
</tr>
<tr>
<td>AWS Config²</td>
<td>Monitors and records configuration changes of AWS resources for compliance assessment, security analysis, and troubleshooting.</td>
</tr>
<tr>
<td>Amazon Inspector¹</td>
<td>Performs automated security assessments on EC2 instances and applications, generating detailed findings for vulnerability remediation.</td>
</tr>
<tr>
<td>Amazon Macie⁴</td>
<td>Uses machine learning to identify sensitive data and monitor access patterns, helping with data protection and privacy compliance.</td>
</tr>
<tr>
<td>AWS Security Hub⁷</td>
<td>Provides a centralized view of security alerts and compliance status, enabling continuous monitoring and streamlined remediation workflows.</td>
</tr>
<tr>
<td>Amazon Detective⁵</td>
<td>Analyzes and visualizes security data from multiple sources to aid in investigating and understanding potential security issues and threats.</td>
</tr>
</tbody>
</table>

against various AWS services, configurations, and settings to identify potential vulnerabilities and security gaps. They conduct a comprehensive assessment of AWS accounts, including checks for security best practices, detection of common misconfigurations and compliance with industry standards like NIST and the Center for Internet Security (CIS)¹ AWS benchmarks. This CIS Benchmark is the product of a community consensus process and consists of secure configuration guidelines developed for Amazon Web Services. PMapper² is a script and library for identifying risks in the configuration of AWS Identity and Access Management (IAM) for an AWS account or an AWS organization. It models the different IAM Users and Roles in an account as a directed graph, which enables checks for privilege escalation and for alternate

¹https://www.cisecurity.org/benchmark/amazon_web_services
²https://github.com/nccgroup/PMapper
paths an attacker could take to gain access to a resource or action in AWS. Moreover, for data extraction and collection, there are tools like Resource Counter\(^1\) and S3Scanner\(^2\), that scans S3 buckets, which are one of the most valuable targets to attackers. Similarly to the previous section, Table 3.3 contains a list of the most famous open-source tools.

Table 3.3: Open source Tools for Data Collection and Threat Analysis

<table>
<thead>
<tr>
<th>Free Tools</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacu(^3)</td>
<td>Penetration testing tool for cloud, similar to Metasploit. Can enumerate the environment and also look for backdoors and easy access.</td>
</tr>
<tr>
<td>Cred Scanner(^4)</td>
<td>Python script that can be used to look for AWS credentials left in files. Only looks for AWS credentials and not other kinds.</td>
</tr>
<tr>
<td>AWS PWN(^5)</td>
<td>A group of multiple scripts that can be used during each AWS cloud penetration test sep. Checks for IAM access keys and buckets’ existence, among others.</td>
</tr>
<tr>
<td>Nimbostratus(^6)</td>
<td>Can dump credentials, check permissions for them, extract instance metadata, create new users in AWS environments</td>
</tr>
<tr>
<td>Cloudbrute(^7)</td>
<td>Finds key elements of a system like open buckets, apps, and databases hosted. Can work in more cloud providers, apart from AWS</td>
</tr>
<tr>
<td>Cloudjack(^8)</td>
<td>Cloudjack is a Python script created to check a vulnerability that happened as a result of a decoupled Route53.</td>
</tr>
<tr>
<td>Mimikatz(^9)</td>
<td>Mimikatz is one of the most used tools, especially in a Windows environment. Can dump passwords, Kerberos tickets, and much more.</td>
</tr>
<tr>
<td>Nessus(^10)</td>
<td>The most classic vulnerability scanner, including cloud infrastructure. Can identify easily vulnerable components</td>
</tr>
</tbody>
</table>

---

\(^1\)https://github.com/disruptops/resource-counter
\(^2\)https://github.com/sa7mon/S3Scanner
It is important to note that, due to the rapid evolution of the cloud and the tools or even the operating systems and the languages the tools operate on, some of them might not work anymore or be compatible with certain AWS systems. This is a classic problem of open source tools that are being developed by independent, third-party developers and not big corporations that have to keep updating their tools and systems simultaneously. For example, during this work’s trial and error phase, Nimbostratus was no longer able to perform in a new version of Kali Linux. Most of the tools in that table actually were not able to perform, at least in a complete or always stable way.

In order to choose between the plethora of tools that exist, we had to compare them based on how much of the aforementioned information they provided, as well as some other properties. In total, we picked our tools based on the following properties:

1. information provided
2. functionality
3. stability
4. accessibility

Functionality means how easily a tool can be integrated into our own system. For example, while Pacu does provide a lot of information, we were not able to pipeline it to our own system. Accessibility means basically that they are not proprietary tools and are open source and easy to use. AWS Cloudtrail is a thorough tool for example, but it is AWS proprietary and thus, not free to use. Finally, stability means that the tool performs in the same way and provides results every time it is used. Enumerate-iam\(^1\) for example was a tool that did provide all of the policies, roles, and permissions, but it didn’t work for every case and even for the same scenario, some times it produced

\(^1\)https://github.com/RhinoSecurityLabs/pacu
\(^4\)https://github.com/disruptops/cred_scanner
\(^5\)https://github.com/dagrz/aws_pwn
\(^6\)https://github.com/andresriancho/nimbostratus
\(^7\)https://github.com/0xsha/CloudBrute
\(^8\)https://github.com/prevade/cloudjack
\(^9\)https://github.com/gentilkiwi/mimikatz
\(^10\)https://www.tenable.com/products/nessus
\(^1\)https://github.com/andresriancho/enumerate-iam
results while other times resulted in an error. Based on these 4 criteria, the best two tools to use were awless \(^1\) and trufflehog \(^2\).

In contrast to the AWS CLI, which is overly convoluted, difficult to work with, and most important, proprietary, awless focuses on simplicity. In particular, awless stands out because it has small and hierarchical commands, clear and flexible terminal output, a local log of all the cloud modifications done with it, and syncing capabilities to a local graph storage of one’s cloud representation. The only thing that it requires is an aws profile to get the credentials from, which we already have by creating the Cloudgoat scenarios. As such, it is possible now to query the details of the infrastructure of AWS systems, from the most popular APIs, each with a simple command. In short, it can provide us with mostly all of the data and metadata that was discussed in the previous section. The only data that it cannot provide us with, is the data for the Codebuild and SSM APIs. Another important attribute that Awless offers, is that it can output the data in different formats, such as JSON, tables, or even RDF data.

As for trufflehog, it is similar to Cred Scanner. However, it is much more powerful and can look for different kinds of credentials and security assets, like private keys of keypairs. Trufflehog can be used to search for credentials and keys in GitHub repositories, S3 buckets, or even local files that are provided.

With just these two tools, we have most of the data that we need for the vast majority of the AWS systems that exist, as well as looking for vulnerabilities from misconfigurations with publicly accessible credentials. None of the other tools mentioned above could provide us with the same level of information that awless and trufflehog did. Of course, some of the proprietary tools could, but the accessibility property was violated, i.e. we had to pay to use them. For the sake of completeness, in Table 3.4 we mention the reasons why the other tools were not picked.

### 3.4 Graph Construction

The next step was to figure out where to store the data as a graph and how to create that graph. This is where graph databases, like Neo4j and Amazon Neptune, come in. Graph databases are basically databases that handle all of the data and their metadata in a form of a graph. Between these two aforementioned options, we chose the former over the latter, not only because it

\(^1\)https://github.com/wallix/awless

\(^2\)https://github.com/trufflesecurity/trufflehog
Table 3.4: Problems with other open source tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphKer</td>
<td>Stability issues. Could not receive data from cloudgoat scenarios in a reliable way</td>
</tr>
<tr>
<td>Prowler and Scoutsuite</td>
<td>Powerful tools but could not find the IAM misconfigurations in the example scenario</td>
</tr>
<tr>
<td>PMapper</td>
<td>Good for IAM enumeration, but awless does that and much more</td>
</tr>
<tr>
<td>Resource Counter</td>
<td>Stability issues, could not be installed</td>
</tr>
<tr>
<td>S3Scanner</td>
<td>Scans public buckets or buckets you provide the name with, so trufflehog does it better</td>
</tr>
<tr>
<td>AWS PWN</td>
<td>An old version of pacu in scripts, didn't provide as much information as awless.</td>
</tr>
<tr>
<td>Cloudbrute</td>
<td>Stability issues</td>
</tr>
<tr>
<td>Nessus and Mimikatz</td>
<td>Not cloud-specific, couldn’t provide much information</td>
</tr>
</tbody>
</table>

is open source and not proprietary to AWS, but because it is also a very popular graph database with a lot of plugins and options for us to use. One of the most famous Neo4j plugins is the apoc plugin\(^1\), that is responsible for handling json files in Neo4j. Through it, one can use the json files in order to create nodes and relationships based on the json data. Moreover, Neo4j offers capabilities like using data from URLs, Cloud storage or even local files. Finally, it uses the cypher language, which is a programming language that is made for creating and querying graphs, a property that is going to be of utmost importance to our design.

### 3.5 Graph analysis and finalization

For graph analysis, it was important to handle the data in a way that would be easier to visualize in neo4j. We chose the JSON format for the data, which offers high flexibility, ease of manipulation, and compatibility with other tools

\(^1\)https://neo4j.com/labs/apoc/4.1/installation/
and systems. Moreover, it is more widely known and used, so it was easier to find out how to properly adjust and configure for this work. As for visualizing the data and finalizing our tool, we used a combination of Python and Cypher languages. Python was chosen because it offered the easiest way to manipulate the data but most importantly, it allowed us to use the Cypher language through its libraries. As such, through Python, we were able to write commands in Cypher language, which is how we could create nodes and relationships for our knowledge graph. In order to achieve that though, we had to define what our nodes would be, and how the relationships between them would be created.

3.5.1 Policies, Users and Roles

Naturally, every single entity mentioned in Chapter 3.1 would be a node based on its unique ID. With the awless list command, we could gather all of the metadata mentioned in Chapter 3.1 in json format (and much more actually that are also included in the nodes for the sake of completeness) and use them as the metadata of the nodes in Neo4j. Firstly, we created the nodes for the Users, the Roles and the Policies. In order to form relationships between them we created an auxiliary/bridge type of node, called the Policy Attachment node. In particular, with the ARN of every policy and the awless show command, we were able to find which policy belonged to each user (or role). After that, we created the Policy Attachment node, that had as its ARN, name and ID, the metadata of the respective policy. Moreover, it contained as its username and userID metadata, the respective metadata of the user. If the policy was connected to a role, it has as username and userID the respective rolename and roleID. As such, the Policy Attachment node contained metadata from both the Policy and the User/Role nodes. We used these same metadata to create the relationships. Specifically, if a Policy Attachment Node has as its ID the same ID as a Policy Node, then these two specific nodes would be connected. Similarly, if the Attachment Node has as its userID the same ID as the User, then a relationship between those two nodes would be created. As such, we created various triplets of User(or Role)-Policy Attachment-Policy nodes, that basically showcased what permissions each user/role has, based on its policy. Then, we created a new kind of node, called the Policy version which would be connected with their respective policies based on the Policy ARN. All of these are depicted in figure 3.2
3.5.2 Other kind of AWS related Nodes

Afterwards, we created nodes for Instances, Instance profiles, Buckets, Lambda functions and RDS databases. We then checked the permissions of every policy. If a Policy has permissions to work with S3 buckets for example, then we create a relationship between every bucket there exists. If it has permissions to work with only a specific bucket, then we create a relationship only to that specific bucket, based on the name of course. We did the same for instances, instance profiles, lambda functions and rds databases. Furthermore, in regards to relationships, instance profiles and lambda functions have in their metadata the name of the role they are assigned to. As such, we create relationships between the instance profiles and the roles, based on the name of the role, as well as the lambda functions and the roles. It is important to note that, through the triplet, we can now understand what permissions the lambda functions and the instance profiles have. Also, instances, as mentioned in Chapter 3.1, might have an instance profile attached to them. If there is an instance node that has an instance profile attached and thus, its name in its metadata, then we connect these two nodes.

The final 2 kinds of nodes that were created were the S3objects and the Keypairs. If keypair nodes existed, they were connected with their respective instance (again, the instance node would have in its metadata the name of the keypair). Similarly, the S3objects were connected with their respective buckets.
For the sake of better visualization, all of those concepts are depicted in two figures, Figures 3.3 and 3.4. Figure 3.3 depicts all of the other nodes (apart from User, Policy, Policy Attachment, Policy Version, and Role) and their relationship with each other. Also, for the sake of better visualization, in Figure 3.3 we only depict the relationships of the rest of the nodes with the Policy Node. Of course, the Policy Nodes are connected to Users, Roles, etc., as depicted in Figure 3.2, but are omitted here. These two kinds of nodes are presented again in Figure 3.4, along with the Nodes from Figure 3.3 that have relationships with.

Figure 3.3: This metamodel figure describes the rest of the nodes and their relationships with each other. Moreover, it is split in two parts. This figure describes the relationships of the rest of the nodes with the Policy Node. The Policy node is connected to Users, Roles, and Versions as depicted in Figure 3.2, but are ignored here for better visualization purposes.

It is vital to discuss also how the visualization happens with Cypher as well, i.e. the creation of nodes and the relationships. Firstly, the creation of a node, and particularly of an S3 object, is depicted in Figure 3.5. One can see that we create an object based on its unique ID (line 3 of the code), so no other
Figure 3.4: This metamodel figure describes the relationships of the Lambda, Instance Profile and S3objects with the User and Role Nodes. This is the second part of the two-part metamodel description between this figure and figure 3.3. Only these 3 kinds of Nodes interact with Roles and User nodes and that’s why we omitted the rest, that are of course depicted in the previous figure

object with this ID can be created again. Then we set its metadata based on the metadata of the json file (lines 4-9). If the object in the json file lacks one of those attributes, like for example the HasCredsFor attribute, then that attribute is skipped by Cypher and is not added to the Node details. The other kinds of Nodes are created in a similar fashion.

Figure 3.6 depicts how a relationship between two Nodes is created and specifically, between the Policy and Policy Version Nodes. This query works in the following way. First, it gathers all the nodes that are of the Policy and the Policy Version kind (line 1). Then, it checks every Policy node for its ARN (p.arn) and checks if that is the same as the ARN of any of the Policy Version Nodes (pv.arn). It thus cross-checks every Policy Node with every Policy Version Node and checks if they are any that have the same ARN. If it finds two nodes (one of each kind) that share the same ARN, then it connects them with a relationship from the Policy Node to the Policy Version node that
Figure 3.5: This figure depicts the query for creating an s3 object, along with its most important metadata

```
CALL apoc.load.json("json-files/s3objects.json") YIELD value
UNWIND value AS ob
MERGE (obj:S3Object {id: ob.ID})
set obj.bucket = ob.Bucket
set obj.class = ob.Class
set obj.key = ob.Key
set obj.size = ob.Size
set obj.Owner = ob.Owner
set obj.hascredsfor = ob.HasCredsFor
```

says "HAS_VERSION" (line 3). As such, it creates every relationship of this kind between all the Policy Nodes and The Policy Version Nodes that share an ARN. The relationships between other kinds of Nodes are created in a similar fashion.

Figure 3.6: This figure depicts the query for creating a relationship between two nodes, in particular a Policy and a Policy Version.

```python
match(p:Policy), (pv:PolicyVersion)
where p.arn = pv.arn
create (p)-[:HAS_VERSION]->(pv)
```

3.5.3 Privilege Escalation Nodes

One of the most important aspects of HackerGraph is the ability to trace vulnerabilities in regards to the aforementioned 21 privilege escalation techniques, inline credentials and private keys. Of course, if any of those vulnerabilities are found, a respective node is created. These kind of nodes are also depicted in Figures 3.2, 3.3 and 3.4. For the privilege escalation techniques, we begin with the code described in Figure 3.7. This code is used to find the sets of permissions that allow for privilege escalation. Those sets of permissions, along with the name of the technique they enable, are imported with the command "from dict import escalation methods" as "escalation_methods" from a dictionary called dict.py we ourselves created.

In particular, the code works as follows: First, we iterate between all of the
policy elements that exist in the json file of "policies.json". We then save their policy ID and we go through their property documents, which itself contains another list called statements. Then, if the statement property has the word action, that means that what follows is a list of the permissions the policy has. We of course compare the permissions that the policy has with the set of permissions from each escalation method. If there is a match, then we create a new json element, with id being the Policy ID, the name of the technique, the set of permissions and the escalationID. If a policy has multiple escalation techniques, the EscalationID basically indicates the number of the techniques. For example, if its value is "method4", then it is the 4th of the methods that can be used for escalation. Then, we parse it to a json file. We do this for every policy element.

```
for policy in policies:
    matched_policies = []
    policy_id = policy['ID']
    document = json.loads(policy['document'])
    statements = document['Statement']
    policy_actions = set()
    for statement in statements:
        if 'action' in statement:
            actions = statement['action']
            if isinstance(actions, str):
                policy_actions.add(actions)
            else:
                policy_actions |= set(actions)
    for key, value in escalation_methods.items():
        if all(key in policy actions for a in value.keys()):
            matched_policies[value] = set(value.keys())
    matched_policies_list = [{'ID': policy_id, 'EscalationID': f'method{index}', 'Name': key, 'Permissions': list(value.keys())} for key, value in matched_policies.items()]
    if matched_policies_list:
        with open('json/Templates/escalation_methods.json', 'a') as f:
            json.dumps(matched_policies_list, f, indent=2)
            f.write(']

print('escalation methods file created successfully!')
```

Figure 3.7: This figure depicts a snippet of our python code, specifically the part for detecting the permissions required to perform any of the 21 privilege escalation techniques

As such, we create a json file that contains a json array of elements that describe the techniques, each with the Policy they are related, the name of the technique, the set of permissions and the EscalationID.

Of course, this just creates the json file, so in order to create the nodes and the relationships we have to use cypher, specifically as it is shown in Figures 3.8 and 3.9, respectively. In order to create the node, we simply load the json file and create a node based on the aforementioned properties. Cypher checks on the properties of the json elements, and matches it to the properties we want to add in the node. For example, it takes the ID of the element (which is the policy ID the escalation refers to), and adds it as the property "id" to
the node. Same with name, EscalationID and permissions. In order to create the relationship between the privilege escalation nodes and policy nodes, we simply compare the IDs. As explained before, with the code shown in figure 3.9, cypher gathers all of the two kinds of nodes and check their id property. If it finds a policy node that has the same id as a privilege escalation node (or vice versa), then it creates a relationship between the two named "possible escalation". The word possible here means that the technique is there, but it doesn’t mean that the attacker can actually escalate. For example, if the technique has to do with picking a different version of policy and applying it, that means he has the permission to do that. However, that requires that first, the policy has more than one version and second, that one of those versions actually gives access to the attacker to more APIs or entities of the AWS system in general. In short, having the permission doesn’t mean that it is in itself a vulnerability, but an indication to one.

Figure 3.8: The cypher code used to create the privilege escalation node

```cypher
CALL apoc.load.json("json-files/escalation_methods.json") YIELD value
unwind value as esc
```

Figure 3.9: The cypher code used to create the relationship between privilege escalation nodes and the policies. The connection is named as "POSSIBLE_ESCALATION"

```cypher
MATCH (p:Policy), (ES:EscalationMethod)
WHERE p.id = ES.id
CREATE (p)-[:POSSIBLE_ESCALATION]->(ES)
```

### 3.5.4 Vulnerability Nodes

The final kind of node we create, is the vulnerability node, related to the inline credentials and the keys. This kind of node doesn’t have to be unique, i.e we can have two vulnerability nodes that are the same in a scenario. This are only used as an indication for a vulnerability that exists, to make it easier for a security engineer to analyse the results. In the case of cloudgoat scenarios, one can easily see the vulnerabilities even without this node but in bigger
scenarios with millions of nodes, this will help the engineer pinpoint exactly the vulnerable entities. For example, if he wants to find if a user has access to a vulnerability or a privilege escalation technique, he can just go to neo4j after HackerGraph has ran, use a simple cypher command like the one depicted in Figure 3.10, and a path between the user and any privilege escalation node will be shown to him. Of course, for vulnerability nodes one simply changes the "EscalationMethod" to "Vulnerability" in the second line.

```
MATCH (u:User {name: "raynor-iam_privesc_by_rollback_cgid7st2i173d3"})
MATCH (v:EscalationMethod)
MATCH path = (u)-[*]-(v)
WHERE length(path) > 0
RETURN path
```

Figure 3.10: An example of cypher code used to bring up all the paths between a user node and a privilege escalation Node

In order to create those vulnerability nodes, we first need, of course, to find the vulnerabilities. The bash code that utilizes trufflehog to do that is depicted in Figure 3.11. The code is split into two parts, one for the lambda function search and one for the s3 object search. We are going to explain only one part of the code, because the only difference between the two is the aws command to access each entity. So, we first get every lambda function (or s3object) that we have on our json file. We then use an aws command to access the details of the lambda function and save it into a file. We use trufflehog to analyse those files for secrets and save its results in a temporary file for analysis.

Some of the data that trufflehog can detect are private keys and AWS credentials. Let’s begin with the AWS Credentials case. In the aforementioned temporary file, if trufflehog has found some results, it saves them in a specific form. In the case of AWS credentials it provides the access key as raw data. We then use the access key and an aws command to find the user that has this access key. Finally, we add to the element of the lambda function in which we found the credentials a new property named "hascredsfor" with value of the name of said user. As such, we update the json file for the lambda function.

In the case of private keys, we follow a similar logic. If trufflehog finds a private key, it saves it fully, from "BEGIN OPENSSH PRIVATE KEY" to "END OPENSSH PRIVATE KEY". Thus, we look for the former phrase in the temporary file and if that exists, we add to the lambda function, in which we found the key, a property named keypair with value keypair. The same applies for s3 objects.
To summarize, if there are inline credentials or private keys, we add a property to the json element of the entity that contains them, named "hascredsfor". If it’s AWS credentials, its value is their user and if it’s a private key, its value is the word keypair. Thus, there are two kinds of relationships that can be created, also depicted in Figures 3.3 and 3.4. In particular, if the value of that attribute is a "UserName", then, either the Lambda Function or the s3object will connect to the User that has this Username attribute, as depicted in Figure 3.12.

Figure 3.11: The bash code used to locate credentials and private keys. This utilizes trufflehog and checks lambda functions and s3objects. The code is separated in two parts, each for every kind of entity. If one wants to search in other entities as well, he has to add the same code and modify it slightly.

On the other hand, if they have a private key, then, as mentioned before, we cannot check to which keypair this private key belongs, and as such, we connect the Lambda Function or the S3Object with every keypair that exists in the system, as depicted in figure 3.13. This is the only case of possible false positive connections, since nodes might be connected to keypairs that have no private key of. However, because of the small size of the scenarios, when a Lambda or an S3object do have a private key pair, there is only one keypair node to connect to, which is the keypair to which the aforementioned private key belongs.
Figure 3.12: This figure depicts the two queries, separated by a blank line, for creating a relationship between a lambda function and a specific user, or a s3object and a specific user. If either the lambda or the object contains credentials for a User, then that is shown in the "HasCredsFor" attribute, that contains the username of the User, and the relationship is formed based on that User.

However, we still haven’t created the vulnerability nodes we discussed. Their creation follows a very similar logic, both for the credentials and the private key case, depicted in Figures 3.14 and 3.15, respectively. Each entity that contains inline credentials or private keys is connected with each own vulnerability node and only the relationship is different. In the cypher code we look for the "hascredsfor" attribute and check its value, which can be either the word keypair or a user. In the case of credentials, the value is not the word keypair, so then relationship is named "HAS INLINE CREDENTIALS" while in the case of private keys, the value is the word keypair, so then the relationship is named "HAS PRIVATE KEY".

With all these methods, we now have a thorough analysis of the graph. Based on the metadata, the vulnerabilities and the inline credentials that might exist, we create different kinds of nodes and connect them with different relationships between them. Thus, we have a holistic view and analysis of each scenario.

3.6 Verification of methodology

In order to verify that our methods have been properly chosen and have actually resulted in a comprehensive design, a comparison is being made with the two most important state-of-the-art tools, namely the DSL for AWS and the
Method | 39

Figure 3.13: This figure depicts the two queries, separated by a blank line, for creating a relationship between Lambda functions or S3 objects with every keypair node.

```cypher
MATCH (ob:S3Object), (k:Keypair)
where ob.hascredsfor = "keypair"
create (ob)-[:MIGHT_BE_PRIVATE_KEY_OF]-(k)

MATCH (l:Lambda), (u:User)
where l.hascredsfor = "keypair"
create (l)-[:MIGHT_BE_PRIVATE_KEY_OF]-(k)
```

Figure 3.14: This code shows how a relationship between either a lambda function or s3 object node is formed with a vulnerability node. A vulnerability node is also created simultaneously and if AWS credentials are contained, a relationship named "HAS_INLINE_CREDENTIALS" is formed.

```cypher
MATCH (n)
WHERE n.hascredsfor IS NOT NULL
WITH n
WHERE n.hascredsfor <> 'keypair'
CREATE (v2:Vulnerability {name: 'Vulnerability'})
CREATE (n)-[:HAS_INLINE_CREDENTIALS]->(v2)
```

awspx tools. We compare the cloudgoat scenarios tested in the DSL for AWS paper with our own system and discuss the results. Specifically, in the official cloudgoat github page ¹, one can find the step-by-step solution for each scenario. We modified these solutions to only showcase the AWS-related parts of the solution, and then we map each step of the solution to a part of our graph. We discuss what knowledge our graph provides, compared to the graph from awspx and also, we analyze the difference in the results between DSL and HackerGraph, for the steps that DSL provides.

¹https://github.com/RhinoSecurityLabs/cloudgoat
MATCH (n)
WHERE n.hascredsfor IS NOT NULL

WITH n
WHERE n.hascredsfor = 'keypair'
CREATE (v2:Vulnerability {name: 'Vulnerability'})
CREATE (n)-[::HAS_PRIVATE_KEY]->(v2)

Figure 3.15: This code shows how a relationship between either a lambda function or s3object node is formed with a vulnerability node. A vulnerability node is also created simultaneously and if private keys are contained, a relationship named "HAS_PRIVATE_KEY" is formed.
Chapter 4
Design

4.1 Complete Algorithm

Below a summary of HackerGraph’s algorithm is presented. The steps of our algorithm are as follows:

- Gathers all the data needed from the aws system (listall script)
- Checks for the policies and for any escalation techniques (python script)
- Creates a policy attachment JSON file, used as a bridge between policies, users, and roles (polattach script)
- Checks for different versions of policies (polversions script)
- Checks for any credentials that might exist inside our infrastructure (searchcred script)
- Creates nodes out of all the data it collected in the previous steps (cypher code)
- Creates relationships based on all of the above (cypher code)

The whole Python code, bash scripts, and dictionary of escalation methods are available on Github. Moreover, in Figure 4.1, one can see the most important modules of our tool, that work together in order to create our knowledge graph.
4.2 Scripts and Data

In order to gather the appropriate data with our tools, we create 4 different scripts. These 4 scripts basically cover the whole data collection phase of Figure 2.1. In particular, according to Chapter 3, the listall script uses awless to collect all of the data and the metadata. The searchcred script is the script described in Figure 3.11, that uses trufflehog to look through s3objects and lambda functions for credentials and keys. The polattach script creates the auxiliary Policy Attachment json file that will be used to create the node, and similarly, the polversion creates the policy versions json file. The python script is the script described in Figure 3.7. These scripts are written in both Python and Bash language, and provide these data to our main Python code.

4.3 Python code and Graph Construction

The main code of the graph constructor tool is written in Python. The main function of this Python code (apart from the escalation script) is to connect with the neo4j database through a driver. This is provided by using the GraphDatabase library imported from Neo4j, in order to connect with the database we have installed. This is possible because Neo4j uses Bolt, an application protocol for the execution of database queries via a database query language such as Cypher. Through Bolt and Python, we can use code in
Cypher now to upload directly to Neo4j Desktop. So, we can construct our graph of the AWS environment, along with any vulnerabilities from publicly accessible credentials, as well as escalation methods that the user might have already. Hence, we now have a method of efficiently constructing and visualizing the graph. It is important to note, that in order to properly upload the files created by the scripts to Neo4j, we had to create a local graph database, and use its credentials when we connect to Neo4j with the help of the driver.

### 4.4 Cypher Language and Graph analysis

After this point, we basically use cypher language (through python), to create the graph. Cypher language, with the utilization of the apoc plugin, can use the JSON files, and create nodes and relationships as discussed previously out of them, with the json element properties being used as metadata of the nodes. In short, we create every possible relationship that can exist between our different entities. This is basically the selling point of our tool, since all of these relationships happen automatically. If there are some common properties in the metadata of the different data, our code automatically creates a relationship between them.

Finally, after all the nodes and relationships are created, we go to Neo4j Desktop and return every part of the graph with a simple command. It’s important to note here that cypher language enables us to do further analysis of the graph, even after its fully constructed. In particular, the cloudgoat scenarios are small and every single entity that exists in each system scenario is demonstrated. In bigger, actual production environments though, there could be millions of nodes and relationships. It is important to mentioned again here that cypher allows us to isolate parts and check specific relationships and nodes. For example, cypher could be used to return all of the paths a specific user could take. This means, that out of all the entities of the graph, this command would return only the tree that has the user as its root.
Chapter 5

Results

Now, we can finally present the results of our graph constructor tool. Of course, we will begin with the “iam-privesc-by-attachment” scenario, where we will check for escalation techniques for the user "Kerrigan". After that, the results of all the different scenarios of our tools will be presented together. Their in-depth analysis and comparison with DSL and awspx will be provided in Chapter 6.

5.1 Results of IAM-privesc-by-attachment

In Figure 5.1, one can see the whole system of the scenario. As discussed in the two previous chapters, one can see the successful result of our methodology and design. In particular, we can see the User node of kerrigan in yellow, which is connected with its policy attachment and policy node triplet (in pink and blue respectively). Moreover, based on the permissions in kerrigan’s policy, one can see that there is a privilege escalation method attached to the policy, that is described in Figure 5.2. That technique is the ”CreateEC2WithExistingInstanceProfile”, as indicated by its name and is indeed one of the three cases we described in our delimitations. It also becomes evident that the policy has permissions to work both with instances and instance profiles, since it is connected with the instance node and the instance profile node. The instance profile has a role connected to it, which in turn is connected to its own triplet, describing thus its permissions. This in short means that the user kerrigan has the permissions of the instance profile, since her policy allows her to work with the instance profile, and thus its own permissions. One can understand that intuitively because of the continuous path from her until the profile’s permissions. If she can use it, either by
assuming its role or by creating a new instance (where she has access to) and attaching that profile to it, she can do whatever the permissions of the profile allow her to do. Furthermore, on the top of Figure 5.1, one can see another triplet of Policy-Policy Attachment-Role, that for now is not used anywhere.

![Diagram](image)

Figure 5.1: "Attachment" scenario. User "Kerrigan" on the far right has one possible privilege escalation. It is attached to her policy, that can access the instance profile and thus, through its role, can use the instance profiles permissions on her own. On the top another role with its permissions is depicted.

## 5.2 Results of IAM-privesc-by-rollback

After we developed the code of the system for the "iam-privesc-by-attachment" scenario, we had to backtrack and try it out for easier scenarios as well. As such, we continue with the "iam-privesc-by-rollback" scenario, which is shown in Figure 5.3. This is a basic scenario, that even pacu could immediately escalate the user’s privileges with one command. However, for the sake of completeness, we have to assess it as well. As one can easily see that again, there is one user with an escalation method attached to him, which is the "Set default Policy Version" technique. This is the same case as the previous scenario, in regard to our delimitations. In Figure 5.4, the policy version v4 is presented.

In regards to the possibility of further analysis after the creation of
Figure 5.2: Here one can see Kerrigan’s Escalation Method, the "CreateEC2WithExistingInstanceProfile". This method happens when a user has the "ec2:RunInstances" and the "iam:PassRole" permissions within his policy. The "id" property is the same as the ID of the policy that has the permissions.

Hackergraph that was mentioned in Chapter 3.5, we run the code mentioned in Figure 3.10. In particular, by running that code, we get the result depicted in Figure 5.5. This is only one example of what cypher can do, in order to help a security engineer to analyse the security of each graph, and see if there are any connections between entities that shouldn’t exist.

5.3 Results of cloud-breach-s3

This scenario is relatively simple as well. However, this doesn’t provide a user, because it has to do with accessing the metadata server, which of course, a knowledge graph cannot assess. According to our delimitations in Chapter 1.4, we should not bother with that scenario. However, since it is used in DSL for AWS, we do present it here for the sake of completeness. Even in this case though, as seen in Figure 5.6, the whole system can be visualized, including the ec2 instance whose IP is used to access the metadata server. In particular, again according to our analysis methodology discussed in Chapter 3.5, we see the instance being connected to an instance profile (and a keypair).
Figure 5.3: "Rollback" scenario. User Raynor can set different policy versions as default. This is shown in the escalation method node. He is connected to a policy, that itself, is connected to other versions. Thus, he can set any of those as a default.

The instance profile also has a role attached to it. This role has each own triplet, which gives him some permissions through its policy. The policy has two different versions, as can be seen by the two light blue connections, and has permissions to work with buckets, since it is connected to a bucket node. The bucket itself, contains 4 different objects.

5.4 Results of ec2-ssrf

This is a very important scenario since it demonstrates the use of trufflehog, and thus the power of our system. In particular, in Figure 5.7, one can see a full circular path that starts from the user on the far left, goes counterclockwise and ends in the lambda function again. In particular, the searchcred script does find credentials in both the lambda function variables and in a bucket object. The relationships of "has-credentials-for" that are connecting the lambda function with a user and the bucket object with another user, are demonstrated with red arrows, to emphasize its significance. Also, one can
Figure 5.4: Policy version v4 of Raynor’s policy. This policy provides administrator access. It can be seen from ”action: *”, ”effect: allow” and ”resources: *”

see two different vulnerability nodes connected with the lambda function and the s3object respectively. This is of course the AWS credentials case in our delimitations, that trufflehog spots.

5.5 Results of rce-web-app

This scenario has two different users and two different paths to follow for a solution. The whole system, along with both of the users is provided in Figure 5.8. This is also the first scenario that our tool cannot completely solve, due to trufflehog limitations, and can only showcase only one path. Even in that path though, it reaches only the penultimate step and not the final step of the solution. The vulnerability node that exists here has to do with the private key case described in our delimitations. The other vulnerabilities that this scenario has are logs and database credentials that trufflehog cannot trace.

On a side note, it is important to note here that if a policy has a specified bucket linked to her, as is shown in Figure 5.9, then that policy is connected only to that bucket. If it doesn’t, it connects to all buckets, as is shown in the policy in the middle of Figure 5.8.
Figure 5.5: This is the path that results from the code mentioned in Figure 3.10, and shows a path between the User "Raynor" and its escalation technique, ignoring the rest of the nodes of the graph

5.6 Results of codebuild-secrets

The final scenario is the codebuild secrets scenario. In this case, we also come up with another problem, which is the data Awless can offer. As already mentioned, awless offers data for the most important AWS APIs. However, neither the codebuild nor SSM API that is showcased here is one of them. As such, we can’t find the data needed to properly showcase what the scenario needs. Also, trufflehog has the same problem as before and cannot trace the database credentials in the lambda function. Of course though, as can be seen in Figure 5.10, the connections between the entities of the other, common, APIs are still properly showcased.
Figure 5.6: “s3 cloud breach” scenario. The instance has an Instance profile attached. The profile has a role assigned, with a specific policy that defines it. Through it, the instance can access a secret bucket and its objects, which is our goal.

Figure 5.7: “ec2-ssrf” scenario. User Solus (bottom-left), can get a lambda function and its configurations. In there, there are credentials for user wrex (middle), noted with a red relationship. Through his policy, the instance, the instance profile, and its role, he can access the bucket object (light pink) which has credentials for user shepherd (top-middle). He can invoke the lambda function, which is the goal.
Figure 5.8: "Rce-web-app" scenario with two paths. **a)** User McDuck (bottom-left) can access object (light pink, bottom-middle) that has the private key (orange) for the EC2 instance (red). Through it, its profile and the role attached to it, he can access two policies, one that accesses the database (beige) and one that accesses an object with the database’s credentials (light pink-top left). No connection between db and credentials is shown. **b)** User Lara can access the bucket object (light pink-top right), where important logs are. HackerGraph doesn’t find those logs, and thus, no relationship is formed from that object.
Figure 5.9: Specific bucket information in rce-web-app

Figure 5.10: In this scenario, the topology can be shown for the most important APIs, but nothing from the SSM or Codebuild APIs. So, HackerGraph can’t solve this scenario.
Chapter 6

Analysis

Table 6.1 shows a summary of the comparison between the 3 tools, Hackergraph, DSL and awspx. Here, we need to clarify how exactly the comparison is made. The analysis below is also a continuation of the analysis done in the beginning of Chapter 3.

6.0.1 Comparison with DSL

Firstly, between Hackergraph and DSL, we basically compare a knowledge graph (Hackergraph) to an attack graph (DSL). In particular, DSL essentially uses a knowledge graph to create an attack graph. However, DSL is not focused on visualizing the knowledge base of the vulnerabilities and the AWS system, which is the purpose of a knowledge graph, i.e. a knowledge base with a directed graph structure [30]. On the contrary, it is designed to use those knowledge bases and ontologies to model attack vectors and act on them, once a system is given to it. It thus tries to perform a penetration test on the system and reach the goal of each scenario, by following an attack path. Moreover, the DSL paper [6] specifically states that it ”does not include supporting features, such as automated information collection and visualizations”, and uses securiCAD Vanguard 1 to supply the AWS information and the vulnerabilities. As such, we do not treat it as a knowledge graph, although it does use different ontologies and knowledge bases to perform the attacks, but rather as an attack graph. On the other hand, Hackergraph does not perform any attacks but rather focuses on providing information automatically about the system and its vulnerabilities (contrary to DSL) and displaying them, along with their relationships. It tries to provide

1https://www.foreseeti.com/securicad-vanguard-for-aws/
enough information that would be enough for a security and cloud engineer to spot possible attack paths that might exist and act accordingly to protect his system. Of course, it also provides a method, specifically Cypher, to further analyse the graph, based on his needs.

What this means is that we essentially compare the graph analysis and visualization parts of Hackergraph and DSL. Table 6.1 essentially describes until which step of the attack DSL can reach, and similarly, until which step of the attack Hackergraph provides enough information for an engineer to prevent. The purpose of this table is to showcase that, between choosing a state-of-the-art attack graph or a state-of-the-art knowledge graph, choosing the knowledge graph would provide more information to protect a system, rather than choosing an attack graph. In short, which of the two options, showcasing the knowledge of the system or statically performing attacks through attack vectors (based on a knowledge graph), helps a security engineer more with the vulnerability assessment of his environment. Of course, there is the trade-off of the knowledge graph not showing exactly how the attack is performed (in this case at least), however, it provides enough information to stop the attack completely in most cases. On the other hand, the dynamic and complicated nature of AWS hinders DSL in most cases from performing a full attack path and revealing the end goal. On the contrary, it will be demonstrated below that DSL can create a fully linked attack path only once! In short, with this analysis, we essentially show how much each tool helps with the vulnerability assessment of each scenario, without focusing on how the assessment is happening.

6.0.2 Comparison with awspx

The comparison between Hackergraph and awspx is easier to comprehend. Both are knowledge graphs and thus, Table 6.1 shows for both tools until which step enough information is given for a security engineer to spot an attack path and protect his system. Awspx is a state-of-the-art knowledge graph construction tool, in regards to displaying information about AWS systems and vulnerabilities and it is easy to see that our tool performs better. This makes sense, since our tool provides more information. Specifically, awspx provides information only about IAM, S3, EC2 and Lambda APIs and, vulnerabilities-wise, it only checks for the 21 possible escalation techniques. As such, it cannot display information or create relationships between entities that have credentials for users and the users themselves, between entities that contain private keys and instance keypairs, or even information about RDS databases.
On a side note, our tool provides better scalability and modularity as well, compared to awspx. If we want to gather more information, we could just add the results of one more tool in the form of a python or bash script. Awless, trufflehog and the check for the 21 escalation techniques happen with independent scripts within our code. On the other hand, awspx uses the complicated AWS CLI to collect information, so expanding it and adding more knowledge extraction capabilities would be difficult. For example, to collect just the RDS database information that we do, lots of lines of code would need to be added, compared to our one ’awless list databases’ command.

Table 6.1: Comparison of tools for each scenario, in regards to finding attack paths

<table>
<thead>
<tr>
<th></th>
<th>hackerGraph</th>
<th>DSL</th>
<th>awspx</th>
</tr>
</thead>
<tbody>
<tr>
<td>attachment</td>
<td>final step</td>
<td>not supported</td>
<td>final step</td>
</tr>
<tr>
<td>rollback</td>
<td>final step</td>
<td>not supported</td>
<td>final step</td>
</tr>
<tr>
<td>cloud breach</td>
<td>final step</td>
<td>final step</td>
<td>final step</td>
</tr>
<tr>
<td>ec2-ssrf</td>
<td>final step</td>
<td>second step</td>
<td>second step</td>
</tr>
<tr>
<td>rce-webapp-1</td>
<td>penultimate step</td>
<td>first step</td>
<td>first step</td>
</tr>
<tr>
<td>rce-webapp-2</td>
<td>third step</td>
<td>third step</td>
<td>third step</td>
</tr>
<tr>
<td>codebuild</td>
<td>second step</td>
<td>second step</td>
<td>second step</td>
</tr>
</tbody>
</table>

What follows is a detailed explanation of each scenario. The steps that are described in Table 6.1 are according to the modified flowcharts of the solutions of each scenario. The official scenario solutions can be found in the official cloudgoat github page.

6.1 Result Analysis of the attachment scenario

In Figure 6.1, the important thing to notice is that Kerrigan should switch the roles of the instance profiles and create a new instance with that stronger instance profile, in order to get administrator access. By comparing that with our result in Figure 5.1, one can immediately see that we do show that Kerrigan can access the instance, the instance profile, and its role. Also, the stronger role is depicted as well. Finally, in Figure 5.2, one can see that Kerrigan does have the permissions (and thus the privilege escalation technique) to both switch the roles that are depicted and create a new instance to use that role. Moreover, by

1https://github.com/RhinoSecurityLabs/cloudgoat
as such, we can verify that our knowledge graph does demonstrate everything that is required to reach the final step. Kerrigan can find the instance and the instance profile, it has the "iam:PassRole" to switch the role triplet that is attached to the instance profile with the stronger triplet, it has the permission to create a new instance (with the stronger instance profile) and since she can access the instance, she can get administrator access. While we chose not to show it in the graph as a separate node, if one checks the full list of permissions from the Kerrigan policy, he can verify that Kerrigan also has the CreateKeyPair policy, in order to create a new key pair and have access to the new instance.

Awspx also demonstrates the same things, because it can showcase all of the API details as well as the privilege escalation techniques. As such, it performs the same function as our tool. DSL however, excludes this scenario after all because it could not model the tactics required (switching roles in an instance profile) to show an attack path for user Kerrigan.

---

Figure 6.1: Steps for the solution of "attachment" scenario

---

### 6.2 Result Analysis of the rollback scenario

Here, it is a simple scenario, with user Raynor only having to check its policy, check its versions, and use the privilege escalation technique of changing the default policy version. Thus, he finds a policy version that gives administrator
access, as shown in Figure 5.4, and sets it as default. In Figure 5.3, one can clearly see that we showcase all of this information, i.e. the policy, its versions in detail, as well as the escalation technique that is required. So again, our knowledge graph does demonstrate everything that is required to reach the final step. Raynor can check his policy (he is attached to the triplet) and through it, he can check its versions. He has the “iam:SetDefaultPolicyVersion” permission, which is shown in the privilege escalation node and as such, he can switch the default policy of his version.

Awspx and DSL have exactly the same results as before. Awspx shows the same results as ours, since it can demonstrate IAM API details and privilege escalation techniques. DSL cannot model the tactics required, because according to its paper, the scenario is too small with too few entities to actually model.

Figure 6.2: Steps for the solution of rollback scenario

6.3 Result Analysis of the cloud breach scenario

Here, the attacker exploits the misconfigured metadata server to get the instance profile credentials and access the buckets and their secrets. Although we do not take into account the metadata server, one can still see in Figure 5.6, that the whole system topology is properly shown. One can see that the instance has an instance profile, that has a role that can access the secret bucket and its objects. As such, we consider that if we had a different way of accessing
the instance that doesn’t involve the metadata server, our scenario would show
the full solution as well.

AWSpx again shows the same results as ours. The difference here is
about the DSL server. This is the scenario that the DSL did provide a full
and linked solution, as an attack path. It assumed a request forgery to an
Unknown application (the metadata server) and obtained and used the correct
privilege[6].

![Diagram showing steps for the solution of cloud breach scenario]

Figure 6.3: Steps for the solution of cloud breach scenario

### 6.4 Result Analysis of the ec2 ssrf scenario

This scenario is the most important one because it shows the difference
between Hackergraph and both of the others. Moreover, it demonstrates that
the more data/knowledge a system collects, the more information and attack
paths it can show. In Figure 6.4, one can see that by starting as user solus,
one can get the credentials of another user, by getting the lambda function
configuration. Solus’s triplet allows him to access the lambda function, which
trufflehog shows that it contains credentials. By configuring a new profile for
that new user, one can discover a new instance and connect to it. Through its
instance profile, they can access a secret bucket and also get credentials for
a third user, called Shepard. Through that user, one can invoke the lambda
function and win the scenario. On a side note, because we care only about
cloud-specific attacks, we do not take into account the SSRF part of the
solution. Instead, we consider that we can just access the ec2 instance. By
comparing these steps with Figure 5.7, one can see that we present the full
attack path. By starting from the user on the far left, we can see that through
its triplet, we can access the lambda function. Trufflehog finds credentials in its environmental variables, and connects the function to the user to whom the credentials belong (steps 1-3). Through his triplet, he can access the instance and its profile (step 4). The profile again has its own triplet, which can access the bucket and the object. Trufflehog again finds credentials in the object, and thus connects the object with the final user, Sheperd (step 5-6). He, through his triplet, can access (invoke in this case through his permissions) the lambda function (final step).

On the other hand, awspx doesn’t show these connections. Since it lacks a way of showcasing credentials, awspx shows different distinct paths that do not connect with each other. As such, since it cannot get credentials from the lambda function, awspx stops on the second step. Of course, it does showcase the rest of the connections, but in contrast to our tool, there is not a direct path from user Solus to user Sheperd. So, a security engineer would assume that Solus would not be able to invoke the lambda function, which, through Sheperd, he actually can.

In regards to DSL, [6], it supported many of the required steps but did not combine them. Going from user A to user B required the Lambda list operation that was not implemented at the time. As such, it stops on the first step. However, even if it could list the lambda functions, DSL has no way to find inline credentials. As such, it would have the same problem as awspx.

Figure 6.4: Steps for the solution of ec2 ssrf scenario
6.5 Result Analysis of the rce web app scenario

This scenario has two different paths for each solution. The first path is depicted in Figure 6.5 and the second in Figure 6.6. For the first path, a user lists buckets, finds a private key, and uses it to connect to the instance. From its instance profile, the user lists databases and also lists the contents of a second bucket, which happen to contain the credentials for the database. As such, he uses these to connect to that database and get the information inside. This is the first scenario that Hackergraph does not solve completely, however, it reaches the penultimate step. In particular, trufflehog can read the contents of the bucket’s object, however, it does not realize it contains credentials. As such, it does not provide a way to connect the bucket object to the database.

In particular, according to Figure 6.5, we can see that user Mcduck on the bottom left, can access through its triplet a bucket object on the bottom middle, that contains according to trufflehog the private key of the instance’s keypair (steps 1-3). The instance, is connected to an instance profile, that has a role that is part of two triplets. A role can have multiple policies. One of the policies allows him to access another bucket object on the top left (step 4). The other policy allows him to access the RDS database on the right (step 5).

Nevertheless, it still performs better than awspx, which doesn’t have a method to find the private key pair. As such, awspx stops at the first step. Moreover, awspx doesn’t provide information for databases, so the database is not visible either. As for DSL, while it did cover a lot of connections between entities from the scenario, it still had a problem with the inline credentials. As such, it still didn’t go past the first step. However, it was able to connect some of the later steps together. In particular, DSL can act as the McDuck user and list the buckets in the first two steps (without finding the credentials). Also, DSL can act as the instance profile and use its role to list the bucket, not the database though.

For the second path, all three of the tools could not go past the third step. Both HackerGraph and AWS do not have a way of going through logs and showcasing URLs or other important information, that can link entities between each other. As for DSL, it did again connect some steps with each other however, it mostly created independent paths that do not connect with each other and do not present a full and clear attack path to our end goal, which was the contents of the database. In particular, DSL can assume the policies of lara and list the buckets, as well as act again as the instance profile and get
Figure 6.5: Steps of the first path for the solution of the rce-web-app scenario

the buckets. However, those are presented as two independent events that are not correlated with each other.

6.6 Result Analysis of the codebuild-secrets scenario

The final scenario is the codebuild-secrets scenario, with its solutions shown in Figures 6.7 and 6.8 respectively. Unfortunately, HackerGraph cannot perform either of those solutions. This scenario uses the Codebuild secrets and the SSM APIs, which are still not as popular as all of the aforementioned ones. As such, awless does not have a command to showcase either of each content. However, we can note two things. Firstly, one can see in Figure 5.10 that the rest of the APIs are still properly shown and interconnected. Secondly, one should note that had awless been able to demonstrate these data and metadata, HackerGraph would be able to provide its full solution, for at least one of its paths.

Of course, awspx has the same problem, since it still cannot show the contents of these APIs and get the credentials. Again, it doesn’t display the database as well. As for DSL, it has the same problem as we do. For the first path, it cannot model the codebuild secrets API and cannot go past the first step. However, it should be noted that after that it can model almost completely the steps up until the second to last step. In particular, it can act as the user, list the database and read it thus getting virtual access to the penultimate step.
However, it bypasses step 5 and 6, because it cannot create a snapshot of a database, or a new accessible database from that snapshot. As for the second path, it was mostly missed due to unsupported services. It could only assume the role of the instance profile and read the database, but it could not access the credentials.

Figure 6.6: Steps of the second path for the solution of the rce-web-app scenario

Figure 6.7: Steps of the first path for the solution of the codebuild-secrets scenario
Figure 6.8: Steps of the second path for the solution of the codebuild-secrets scenario
Chapter 7

Discussion

Based on the previous analysis of the scenarios, one can realize the difference between knowledge and attack graphs, both in their functionality and their purpose. Depending on the context of their use, they have distinct advantages, proving what we discussed in the beginning of Chapter 6.

Attack graphs, like the DSL in our case, can perform sophisticated modeling and simulations on systems, like assuming roles and completing tasks like accessing databases and buckets. An attack graph that has a sophisticated database of attack techniques, especially cloud-specific ones, could perform better in the aforementioned systems. However, with the complexity of the AWS system or the AWS CLI commands, that could be hard to achieve.

Knowledge graphs, on the other hand, do not perform attack methods in themselves, but rather provide information on the system and possible vulnerabilities. They can represent rich context, like the metadata of cloud entities, in order to demonstrate information about them better. Also, knowledge graphs allow the incorporation of domain-specific knowledge and expertise. They can integrate information from diverse sources, such as security logs, threat intelligence, and expert knowledge, enriching the understanding of the system and enabling more informed decision-making. From the various figures shown in Chapter 5, one can see that even with information about just the 5 popular APIs, almost the entirety of the system can be demonstrated. This, allowed us to show most connections and relationships an attacker could abuse to pivot through each system. Of course, for now, the method with which an attacker could perform this task wasn’t explained. We were examining what happened when an attacker could access a specific entity, not how it could access that.
Finally, as far as analytical capabilities are concerned, knowledge graphs support a wide range of them that enables complex analysis, pattern detection, and identification of potential risks or mitigations. In particular, knowledge graphs themselves could be used as the database for a digital twin of a system, that would give information about how to modify or improve a system. A simple example in our case would be to use HackerGraph to create new rules and policies that follow the least privilege principle. This could even be achieved by using Awless again, which also allows for the creation of new AWS entities. Attack graphs, while helpful for visualizing attack paths, may have limited analytical capabilities beyond path enumeration.

It is important to note also, that Cloudgoat specialises in showing AWS-specific methods of attack for AWS systems. Since our HackerGraph can solve most of these scenarios, or rather display enough information to help protect those scenarios, one can assume that this methodology can actually cover real-world systems and setups. Any problem that is related to a misconfiguration in the AWS system, be it a permission or credentials in plain sight, HackerGraph can spot. This is further supported by the fact that HackerGraph can analyse the most popular APIs used, which means that it can analyse the majority of real-life application systems, operating on AWS.

One can therefore see that knowledge graphs are, for the most part, a better option than attack graphs. While they cannot perform attack techniques themselves, they can show more information. They could even be used as a database for another attack graph construction tool, and assist in finding complex attack paths, that would not be found before.

7.1 Limitations

Of course, while our method creates a comprehensive knowledge graph, it has also some disadvantages. For example, if awless doesn’t get upgraded, we will not be able to get the data for the other APIs, like in the "codebuild-secrets" scenario, and we will have to incorporate another tool to perform a similar function. Moreover, there is no perfect tool, for example trufflehog does find some credentials but in the case of the second rce-web-app path, it couldn’t identify the credentials of the rdf that existed in a bucket object, even though it was written exactly with the words "username" and "password". Other ways must be found to gather even more data and make a truly fully cohesive graph.

Another problem that should be discussed has to do with the supply chain aspects of relying on other tools. With the AWS platform and the tools themselves constantly updating, compatibility issues could arise. Even in our
case, in the span of 6 months, we had issues that had to be resolved, due to AWS updating its policies. Firstly, in order to increase the security of their systems, they did not allow buckets to automatically create and add Access Control Lists (ACLs) anymore. As such, every cloudgoat scenario that contained buckets couldn’t be automatically installed, without us manually allowing ACLs in buckets in the AWS console beforehand. Also, tools like Nimbostratus, due to the constant updating of the rest of the tools, are no longer compatible with the cloudgoat scenarios. In short, finding tools that can cooperate with each other appropriately is hard, and becomes even harder the more tools one wishes to add.

Finally, a problem that also arises is that there are not many vulnerable-by-design scenarios outside of cloudgoat, to test our systems. That means, that there might be other hidden techniques or data that we should use, in order to further expand our tool. However, without having easy access to other vulnerable systems, we will not be able to know how to handle these data. For example, our system takes for granted that all of the resources are in the same AWS region. We do not know how relationships can be formed between entities that exist in different regions, or how an attacker can pivot from one region to another.

7.1.1 Dynamic Solutions

As mentioned before, the more tools and data we collect, the more complicated and unreadable the graph becomes. One might suggest that the best solution for achieving a properly complete view of a system and its weaknesses is to create a dynamic system that can perform what a hacker might do. As a result, a security engineer could easily use this tool and find every possible attack path in his system. Of course, that would go beyond the scope of a thesis project and would need very deep knowledge of cloud systems and attack methods. Again though, one could utilize our HackerGraph as one of the many databases required to perform such a task.
Chapter 8

Conclusions and Future work

8.1 Conclusions

One can truly see that knowledge graphs are a definite improvement over simple attack graphs since attack graph construction tools are not developed enough to handle all of the intricacies of even a simple cloud system. The more knowledge and data we can gather, the more connections we can demonstrate in a graph, and as such, the easier it becomes for a security engineer to audit his system. However, a knowledge graph cannot really showcase how an attacker can get into a system, but what can happen if he gets into the system. As of this moment, there aren’t tools that can show attacks that have to do with cloud systems in a satisfying way. As such, it is not easy to integrate that kind of knowledge into our graph. Moreover, adding more tools to gather data, while effective, adds a lot more complexity to our system, with compatibility issues becoming more difficult to solve or bypass. Nonetheless, as far as our resources and tools go, we have created an improved tool that gathers and showcases most of the important data of an AWS system, alongside its vulnerabilities and important keys and credentials that might exist. Moreover, it is always possible to integrate more tools into our system. This project just showcases the capabilities that Neo4j, python, Cypher, and different CLI tools can provide in order to answer our research question, which is if it is possible to provide a holistic view of a complicated cloud system.

8.2 Future work

There are multiple things that can be done as future work. First of all, the tool can always be expanded with new tools that might perform even more specific
tasks in finding vulnerabilities. One might even integrate even non-cloud-specific tools, like Nessus, for example, to have an even better view of the system. Moreover, one can focus on developing more specific tools, that can be later connected to our own. It became evident that a lot of credentials could not be identified, even though a security engineer could easily understand that they were credentials if they were presented to him. One tool that could be developed would focus on finding these kinds of credentials. Also, as far as tools go, we could design a system with the AWS proprietary tools mentioned in Chapter 3.3.1, and compare the results. Thus, one could study if the benefits that they provide cancel out the money they require.

Finally, another aspect of future work is to actually develop more vulnerable by-design scenarios, that take into account what was discussed in the previous chapter. Creating scenarios with multiple security groups, Virtual Private Clouds (VPCs) and on different regions could show vulnerabilities and attack paths not demonstrated before.
References


[16] M. Chu, K. Ingols, R. Lippmann, S. Webster, and S. Boyer, “Visualizing attack graphs, reachability, and trust relationships with navigator,” in


With the rapid adoption of cloud technologies, organizations have benefited from improved scalability, cost efficiency, and flexibility. However, this shift towards cloud computing has raised concerns about the safety and security of sensitive data and applications. Security engineers face significant challenges in protecting cloud environments due to their dynamic nature and complex infrastructures. Traditional security approaches, such as attack graphs that showcase attack vectors in given network topologies, often fall short of capturing the intricate relationships and dependencies of cloud environments. Knowledge graphs, essentially a knowledge base with a directed graph structure, are an alternative to attack graphs. They comprehensively represent contextual information such as network topology information and vulnerabilities, as well as the relationships between all of the entities. By leveraging knowledge graphs’ inherent flexibility and scalability, security engineers can gain deeper insights into the complex interconnections within cloud systems, enabling more effective threat analysis and mitigation strategies. This thesis involves the development of a new tool, HackerGraph, specifically designed to utilize knowledge graphs for cloud security. The tool integrates data from various other tools, gathering information about the cloud system’s architecture and its vulnerabilities and weaknesses. By analyzing and modeling the information using a knowledge graph, the tool provides a holistic view of the cloud ecosystem, identifying potential vulnerabilities, attack vectors, and areas of concern. The results are compared to modern state-of-the-art tools, both in the area of attack graphs and knowledge graphs, and we prove that more information and more attack paths are vulnerable by-design scenarios can be provided. We also discuss how this technology can evolve, to better handle the intricacies of cloud systems and help security engineers in fully protecting their complicated cloud systems.
Why was this problem worth a Bachelor’s/Master’s thesis project? (i.e., why is the problem both significant and of a suitable degree of difficulty for a Bachelor’s/Master’s thesis project? Why has no one else solved it yet?)

How did you solve the problem? What was your method/insight?

Results/Conclusions/Consequences/Impact: What are your key results/conclusions? What will others do based on your results? What can be done now that you have finished - that could not be done before your thesis project was completed?

*Keywords*: Cloud security, Knowledge graph, Attack graph, Vulnerability assessment, Attack paths, Vulnerable-by-design systems, Cloudgoat

Cloud security, Knowledge graph, Attack graph, Vulnerability assessment, Attack paths, Vulnerable-by-design systems, Cloudgoat

"Abstract":


"Keywords": Molnsäkerhet, Kunskapsgraf, Attackgraf, Sårbarhetsanalys, Sårbara miljöer, Cloudgoat

"Abstract":

Älla avhandlingar vid KTH måste ha ett abstrakt på både engelska och svenska. Om du skriver din avhandling på svenska ska detta göras först (och placera det som det första abstrakten) och du bör revidera det vid behov.

If you are writing your thesis in English, you can leave this until the draft version that goes to your opponent for the written opposition. In this way, you can provide the English and Swedish abstract/summary information that can be used in the announcement for your oral presentation. If you are writing your thesis in Swedish, then this section can be a summary targeted at a more general reader. However, if you are writing your thesis in Swedish, then the reverse is true - your abstract should be for your target audience, while an English summary can be written targeted at a more general audience. (This means that the English abstract and Swedish sammanfattning or Swedish abstract and English summary need not be literal translations of each other.)

Do not use the \textbackslash glspl\{\} command in an abstract that is not in English, as my programs do not know how to generate plurals in other languages. Instead, you will need to spell these terms out or give the proper plural form. In fact, it is a good idea not to use the glossary commands at all in an abstract/summary in a language other than the language used in the \texttt{acronym.tex} file - since the glossary package does \textbf{not} support use of more than one language.

The abstract in the language used for the thesis should be the first abstract, while the Summary/Sammanfattning in the other language can follow.

"Keywords": Molnsäkerhet, Kunskapsgraf, Attackgraf, Sårbarhetsanalys, Sårbara miljöer, Cloudgoat


acronyms.tex

\begin{verbatim}
$\texttt{Local Variables:}$
$\texttt{mode: }\texttt{Latex}$
$\texttt{TeX-master: t}$
$\texttt{End}$
% The following command is used with \texttt{glossaries-extra}
\setabbreviationstyle[acronym]{long-short}
% The form of the entries in this file is \newacronym[\textit{label}]{\textit{acronym}}{\textit{phrase}}
% or \newacronym[\textit{options}]{\textit{label}]{\textit{acronym}}{\textit{phrase}}
% see "User Manual for \texttt{glossaries.sty}" for the details about the options, one example is shown below
% note the specification of the long form plural in the line below
\newacronym[longplural={Debugging Information Entities}]{DIE}{DIE}{Debugging Information Entity}
% The following example also uses options
\newacronym[shortplural={OSes}, firstplural={operating systems (OSes)}]{OS}{OS}{operating system}
% note the use of a non-breaking dash in long text for the following acronym
\newacronym{IQL}{IQL}{Independent -QLearning}
\newacronym{KTH}{KTH}{KTH Royal Institute of Technology}
\newacronym{LAN}{LAN}{Local Area Network}
\newacronym{VM}{VM}{virtual machine}
% note the use of a non-breaking dash in the following acronym
\newacronym{WiFi}{‑WiFi}{Wireless Fidelity}
\newacronym{WLAN}{WLAN}{Wireless Local Area Network}
\newacronym{UN}{UN}{United Nations}
\newacronym{SDG}{SDG}{Sustainable Development Goal}
\end{verbatim}