Universiteit van Amsterdam

MSc System & Network Engineering

Automated analysis of AWS infrastructures

P.G.J. Bennink

Supervisor: Cedric van Bockhaven

August 9, 2018

Abstract

This project intends to find out what information can be gathered about an Amazon Web Services (AWS) infrastructure based on limited access gained through the infiltration of an EC2 instance and/or retrieved access keys. This type of analysis is useful in penetration testing and red teaming operations. To this end we analyzed the AWS platform as a whole, the relationships that can exist between components, and what factors influence access to information about the infrastructure. We then built a tool that automatically gathers and parses information about the infrastructure via the AWS CLI to enumerate the components and the relationships between these components. It then imports this data into a graph database that allows for visualization of the data. This tool is able to combine the information multiple access keys can retrieve into one resulting database of the infrastructure, and can also include components that can be seen, but not yet accessed. We thus conclude that this type of AWS infrastructure analysis is useful for reconnaissance and security audits. Future work includes expanding the amount of supported services and commands.

Contents

1 Introduction

AWS is "a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow."[10] On this platform, users can build their own architectures by deploying components and then connecting these components with one another in whatever way necessary. When someone successfully infiltrates a component of such an AWS infrastructure, most of the time the next step is to try and expand this access to other parts of the infrastructure. However, this is difficult without any further knowledge of the infrastructure, and a lot of trial and error would be required. So before proceeding, reconnaissance of the infrastructure should be done to get a better idea of the components the infrastructure is made up of, the relationships between these components, and the best way to ultimately obtain access to valuable data within this infrastructure. Another important factor is finding out which of the found components can already be accessed with the found credentials.

Bloodhound, which relies on Microsoft's Active Directory to "easily identify highly complex attack paths that would otherwise be impossible to quickly identify within an Active Directory environment"[12], offers solutions to the problems specified above. However, the problem Bloodhound encounters is different, as Active Directory is generally one centralized directory containing all related entities. On AWS this is not the case, and data has to be sourced using a multitude of commands, sometimes executed multiple times on different components of the same type.

The goal of this project is thus to design a tool that crawls through an AWS infrastructure, identifies as many components as possible, and shows the relationships between the different components. While all the information this tool uses is already available to the individual with these obtained keys, the vast amount of data makes quick reconnaissance difficult, and a lot of time can be spent trying gain knowledge about the infrastructure from this data. In the analysis for and development of the tool we will pay attention to finding components of the infrastructure that we currently cannot yet control. This is specifically important in red team operations, where expanding access is one of the immediate goals. Another use case would be infrastructure administrators who are looking for misconfigurations, e.g. components that have more access than necessary. Based on the results we discuss what obstacles there still are in this type of analysis, and how they could be solved in the future.

AWS uses a service called Identity & Access Management (IAM) to define the privileges of access keys that entities (users, components or services) use to interact with the infrastructure. One of the problems this project aims to solve is the lack of knowledge about the privileges an access key has.

The tool has two functions. Firstly, for every AWS access key that it receives as input it will try to obtain information about the AWS infrastructure. It will then parse this information and put it in the database for the user to analyze. Secondly, the tool tries to access a predefined amount of AWS CLI commands, and for each command return whether that access key has the permission to execute it. The first function will only use a subset of commands that can be used to gather data, while this function will also try commands used to control the infrastructure and its components. This shows the user in what way they would be able to escalate privilege and/or expand access.

1.1 Research question

The research question will be defined as follows:

Given an infiltrated AWS component, what part of the related infrastructure would an automated tool be able to index?

'Indexing' is here defined as the enumeration of components and the relationships between them. The AWS component we chose is an EC2 instance, which is one of the services AWS offers and will be further explained in Section 4. The reason we assume an EC2 instance is the fact that EC2 instances are often the public-facing component within an infrastructure, which is thus a realistic component to have infiltrated first. Shell access to an EC2 instance also allows one to retrieve that instance's security credentials, which are the credentials this project uses to gather data about the infrastructure.

While for this project we assume an infiltrated AWS component, the results can be generalized to include any situation in which someone has obtained security credentials to an AWS infrastructure.

What should be made clear at this point is that the tool that was built during this project does not compromise any AWS components itself. This tool is meant for reconnaissance of infrastructures to which the user of the tool has gained access in a lawful manner (whether it be their own infrastructure or that of a consenting third party). Furthermore, all access to the AWS infrastructure happens via the AWS CLI.

2 Related work

While a certain amount of AWS analysis and/or visualization tools exist, most of these are meant to be used in combination with admin-credentials. In our situation, these would be the semi-metaphorical "keys to the kingdom", which we cannot assume we have access to for this project.

Nimbostratus is a tool developed by Andres Riancho, which he presented at Black Hat USA 2014.[11] This tool automates the retrieval of access keys and metadata from an EC2 instance as well as the exploitation of a few commands (if that EC2 instance has access to those). While the focus of Nimbostratus lies more with the exploitation of an EC2 instance than with the analysis of the greater infrastructure, this was a starting point for our analysis of AWS and possible weaknesses in (configurations of) IAM.

Another very similar tool is CloudMapper[13], developed by Duo Labs. The difference between this tool and the tool we are creating in this project is the situation in which it can be used. CloudMapper can be used when accessing a specific set of `Actions` is allowed. `Actions` are command with which one can interact with the AWS infrastructure. They will be explained further in Section 4.1. If access to even one of these is not possible, this tool will refuse to work. It is not so much a tool used for penetration testing as it is meant for analysis of an infrastructure to which a relatively high (and very specific) amount of access has already been established. This is not viable in the context of our project, as we do not know beforehand what access we have with each key, and in a lot of cases we will not be able to expand the access of an access key ourselves. Furthermore, Cloudmapper does not support inputting multiple keys and creating **one** mapping based on multiple keys with different permissions.

3 Methodology

The analysis of AWS was done by using the AWS platform (via the web interface, the CLI and the Python SDK), creating multiple infrastructures and reading the documentation. The development of the tool made for this project was done in Python 3. For the testing of the tool the aforementioned infrastructures were used.

4 Analysis

AWS offers a vast amount of services. Defining what is part of 'the infrastructure' can be difficult, as a lot of the services AWS offers play a supporting (but nevertheless equally important) role in the infrastructure. Including all of these services in this project would not be possible, as each service requires a different method for interacting with it. Due to this we scoped this project to just include the following main services:

- Amazon Elastic Compute Cloud (Amazon EC2), which is "a web service that provides secure, resizable compute capacity in the cloud."[15] [1] EC2 components (or instances, as they are called on AWS) are VMs ran on a hypervisor, and are the backbone of each infrastructure. They control all the other components in the infrastructure. These VMs can run any of the more than 70,000 Community AMIs (Amazon Machine Images), which are free, or one of the AMIs offered on the AWS Marketplace, which are paid.
- Amazon S3, which is "object storage built to store and retrieve any amount of data from anywhere."[15][4]
- Amazon Relational Database Service (Amazon RDS), which "provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups."[15][3]

These are three of the most used AWS services[1], in an attempt to make the findings of this project as widely applicable as possible.

However, a setup with these services already makes use of other supporting services by default. Of these services one is the most important, namely IAM, as it is the security backbone that allows entities (users, components, services) to interact with one another. Another service, VPC (Virtual Private Cloud) offers an isolated environment in which to deploy components. This service is also used by default, and will have to be included in this project, as knowing in what VPC a component is located is useful information.

[1]Networkworld.com - Top 30 AWS Cloud Services

4.1 IAM

In Amazon's words, IAM "enables you to manage access to AWS services and resources securely. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources."[15] IAM uses a combination of an Access Key ID, a Secret Access Key and optionally a Token to allow users to interact with the infrastructure. Access Key IDs are public, and can thus, with the right privileges, be found using the AWS CLI. However, once created the Secret Access Key and Token cannot be retrieved from AWS again, meaning that if a user loses either of these keys they will have to generate new keys. Tokens are only used for temporary credentials.

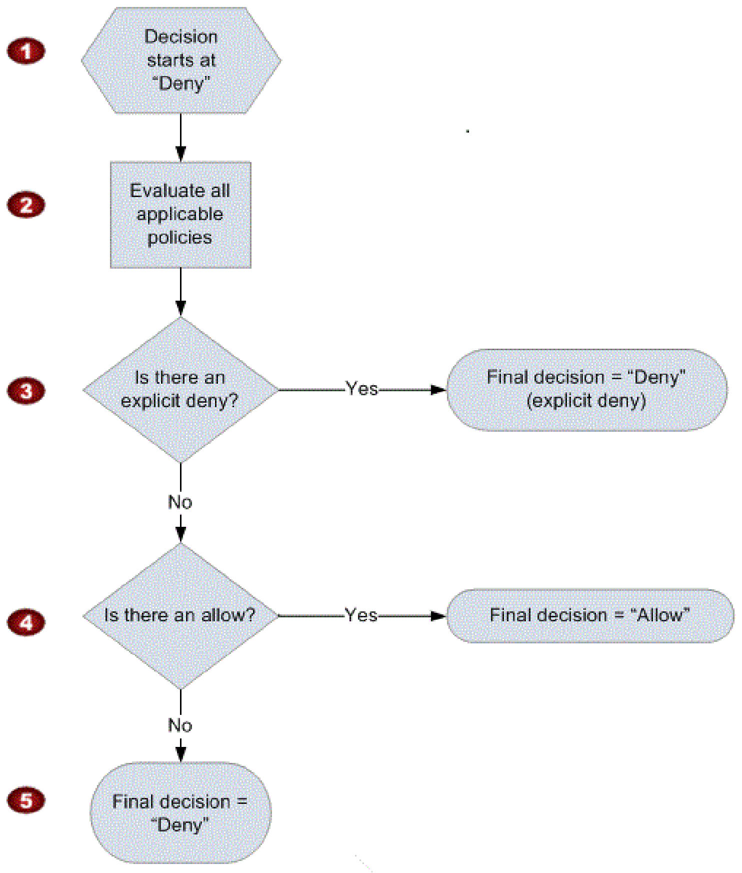
IAM allows for the creating of groups, users, roles, and policies, the latter of which is the basis for the whole platform.

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "s3:ListBucket",
        "Resource": "arn:aws:s3:::example_bucket"
    }
}
```

Listing 1: An example of an AWS policy

Listing 1 shows an example of a policy. A policy is made up of statements that either Allow or Deny a user or entity to access some command or resource. The important parts of these statements are `Effect`, `Action`, and `Resource`. `Effect` can either be Allow or Deny.

Figure 1: Decision flow chart of policy evaluation.


1
Decision starts at "Deny"
2
Evaluate all applicable policies
3
Is there an explicit deny?
Yes
Final decision = "Deny" (explicit deny)
No
4
Is there an allow?
Yes
Final decision = "Allow"
No
5
Final decision = "Deny"

Source: AWS IAM Documentation

As can be seen in Figure 1, by default (if there is no policy that allows or denies a user or entity to do something) the access will be denied. If there is both an Allow and a Deny (in whatever order), access will also be denied. Only if the applicable policies solely explicitly allow an action will it be allowed.

`Action` specifies what actions the statement allows or denies. These actions are formatted as `Service:Action`, as multiple services use the same name for the same action. Wildcards are allowed, meaning that `*:List*` allows all List-actions to be used on all services. Using these `Actions` any user can be prohibited from accessing any specific command within the whole of AWS.

In Listing `Resource` we can define on what specific resource an action can be used. In 1 the `S3:ListBucket` command can only be used on the bucket with the name `example_bucket`.

In terms of information necessary for the enumeration of components of the infrastructure, the `*:List*` and `*:Describe*` actions are the most important, as these are the commands that will statically retrieve information about the infrastructure without making modifications. What makes some of these commands even more interesting, is that their output cannot be controlled at a resource-level, meaning that if for instance `EC2:DescribeInstances` gets executed, it will either show all or none of the EC2 instances, which makes this a valuable source of information for this project. Amazon keeps a list with these types of commands, which can thus be used for this project.[9]

Groups contain users, and policies can either be attached to users directly or by creating a group of users and attaching policies to that group. Roles can be used to attach policies to services or instances.

Roles are basically groups for users, and can be attached to as many instances as necessary. If for example an EC2 instance needs to interact with an S3 bucket, a user with rights to make modifications in the IAM service can attach a role to an EC2 instance. They can then attach a policy to that role that allows all entities in that role to access all (or specific) S3 buckets in various ways (depending on what the instances need to do with the bucket).

These roles are interesting, as it could very well be that two instances that show an overlap in their required permissions get attached to the same role. This means that both instances have more access than they actually need, which can be a security risk (the severeness of which is of course dependent on the `Actions` and `Resources`). This has been observed in production environments.[6]

As with all other IAM policies, the instances need access keys to make use of their permissions. These access keys get distributed via metadata servers, which each instance can access via `http://169.254.169.254`. Each instance has its own metadata server, meaning that instances cannot access information about other instances via their own metadata server. This means that if someone in whatever way has gained shell access to an EC2 instance, they also have access to very detailed metadata for that specific instance, which includes access keys with all permissions that that EC2 instance has. This is thus one way in which someone might be able to retrieve access keys from the AWS platform. However, the keys distributed to instances are temporary access keys and will thus have to be renewed after they expire. The default expiration time is 12 hours, the minimum is 15 minutes and the maximum is 15 hours.[8]

In terms of possibilities to detect the execution of these commands, by default AWS allows administrators to see the last time a user or component has executed a command via the IAM panel in the AWS console. It has been observed that this is not done often.[6] In Section 7.1 we discuss improving the undetectability of the tool.

4.2 Information

All services and their actions can be controlled via either a CLI or one of the SDKs that Amazon offers. What becomes clear after analysis of the output of these commands is the fact that there is an overlap in information between these commands. For all types of components an `Action` exists that will enumerate all components of that type, `EC2:DescribeInstances` being an example of such a command. In most cases these types of commands also offer a lot of information about the surrounding components. In the case of `EC2:DescribeInstances` we are able to retrieve information about the availability zones, subnet IDs, VPC IDs, mounted volumes, security groups and owner ID of these instances. This is all information related to those instances, but also related to other entities in the infrastructure.

Another example is that when access to `EC2:DescribeVpcs` (which lists the different VPCs with information about each) is prohibited, the existence of certain VPCs can still be confirmed based on the data about, for instance, an EC2 instance that resides in that VPC. As this project is about reconnaissance (with the intention of expanding access) this is important information that will be used in our resulting tool.

In a situation where access to for instance `EC2:DescribeSecurityGroups` is prohibited, a decent amount of information about the existence of these security groups can still be obtained via the Actions to describe components that can be put in security groups, such as `EC2:DescribeInstances` or `RDS:DescribeDBInstances`.

4.3 Isolation

As explained in Section 4.2, the information gained from lower-level components can be used to gain knowledge about the higher-level components. Conversely, one can also use the information of higher-level components to gain information about lower-level components.

AWS also allows users to place the components of their infrastructure in a Virtual Private Cloud (VPC). Among the services that support this are RDS and EC2. These VPCs contain security groups, which can be used to limit the amount, type, source and destination of traffic for components within that group.

These security measures are meant to make malicious use more difficult, as it just completely blocks all connections that do not abide by the rules of that security group. However, with access to `EC2:DescribeSecurityGroups`, which enumerates all security groups and the rules attached to them, making an assumption about what components are attached to the security group (based on properties like the group name and the allowed ports/IPs) would still be possible, and thus decide whether gaining access to this security group and its contents would be useful.

5 Development

The tool created for this project can be split up into three parts in terms of functionality, which correspond to the subsections below. Firstly, the crawling of an EC2 instance's metadata server for all metadata, including its IAM credentials. Secondly, the enumeration of permissions of all access keys that are given as input through bruteforcing, which may or may not include access keys obtained using the previous function. And thirdly, the gathering, processing and importing of data about the AWS infrastructure to Neo4j. The fourth section below describes the resulting output, namely the visualization of the database in Neo4j.

The first part is written in bash, and should be executed on an EC2 instance to which shell access has been gained, as these metadata servers can only be reached from or via the instance itself. While there are situations in which one can remotely obtain data from the metadata server (see Nimbostratus[11]), this is not within the scope of this project.

The second and third parts are written in Python 3, and can be executed from any computer with an internet connection that can run Neo4j, Python 3, and some Python packages. The access keys have at this point already been obtained.

5.1 Crawling metadata

Once shell access to an EC2 instance has been achieved it is very easy to access the metadata server. To make it somewhat easier to crawl through the server we wrote a short bash script that crawls through the folders on the server and downloads all of the contents. It will then output the access keys found on the server to the user, who can then use those keys (among potential other found keys) as input for the second and third functions.

5.2 Brute forcing permissions

For all of the access keys the user has obtained this part of the tool tries to execute a predefined amount of commands to offer the user some information about the value of that key in the larger scheme of the infrastructure. For this project the tool ran through 25 different commands from the services EC2, RDS and S3. These commands are listed in Appendix A. The EC2 service also includes commands about VPCs, the data volumes of the EC2 instances, and the security groups. We will discuss this further in section 7.1, but the reason we did not include more commands (both from the services specified above and other services) is mainly related to time. However, the technique used for this test can be extended to include other commands as well. Brute forcing of these permissions is done via boto3, simply by trying to execute the command. This is safe for commands that only retrieve data about the infrastructure, as the state of the infrastructure is not changed. For commands that under normal circumstances alter the infrastructure a `DryRun`-flag can be set to `True`, which results in AWS exclusively checking whether the command could be run

5.3 Data gathering and processing

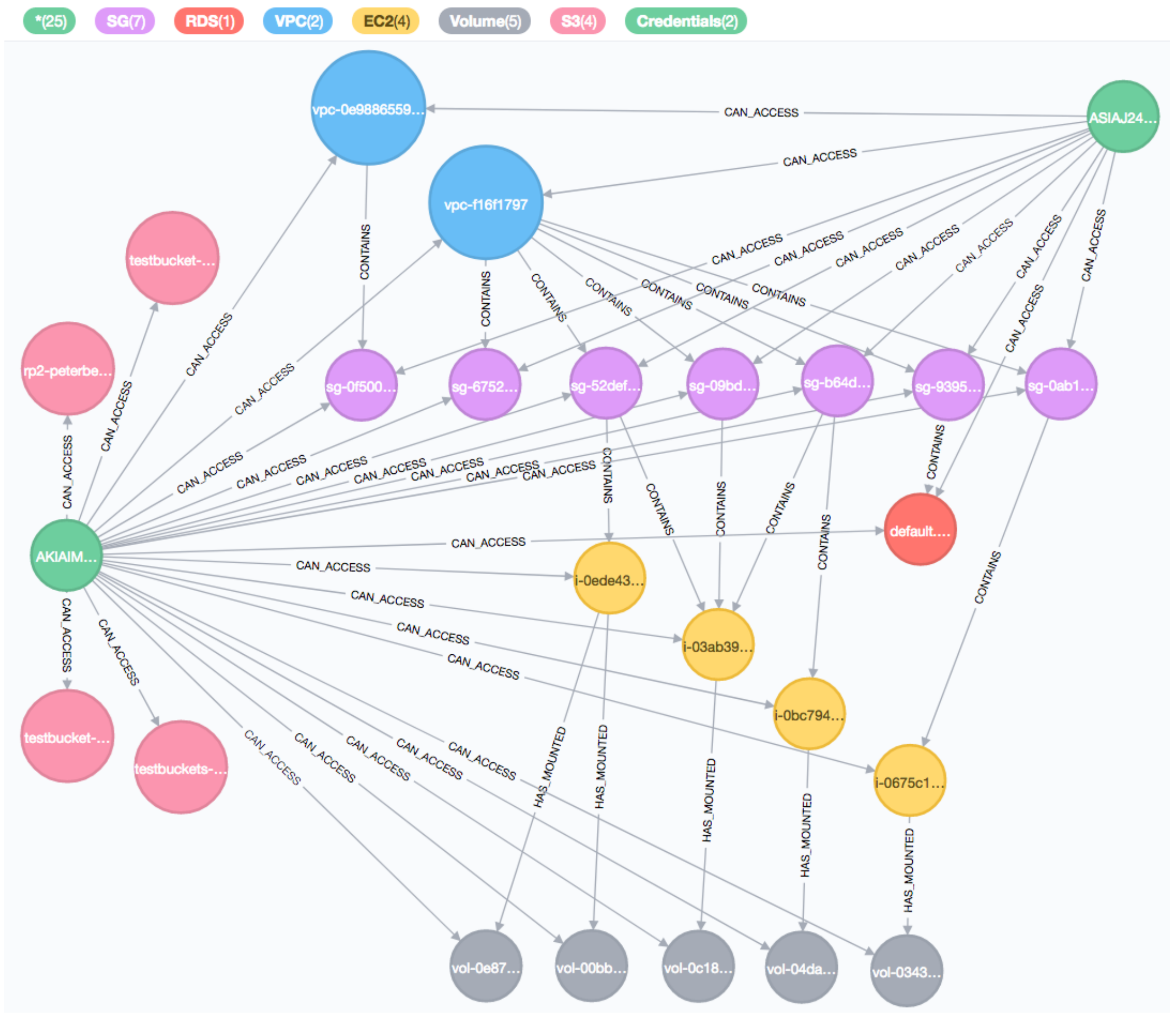
As explained in Section 4, a lot of `*:Describe*` commands contain information about surrounding components. This is used in creating the relationships in the database. For instance, each security group is contained in some VPC. Which VPC this is can be found in the output for `EC2:DescribeSecurityGroups`. In the importing of the data a top-down approach is used: first `EC2:DescribeVpcs` is executed, if this returns data it will get imported in the database. After that `EC2:DescribeSecurityGroups` is executed. If any of the groups in the output of this command are located in a VPC that is not already in the database it will get added now, and a relationship between the security group and its VPC will be added. This same process is repeated for EC2 and RDS components, with relationships added between these components and their respective security groups, and S3 buckets (which do not have security groups). All of the commands on AWS are executed using boto3, and the importing of the data to the database is done using py2neo.

Now the access keys and their permissions need to be imported in the database. For this the results of the functionality described in Section 5.2 are used. For each of the commands used in that section we specified what type of component they are related to (e.g. `EC2:CreateDefaultVpc` is related to VPC). As in the previous example, the service to which the command belongs is not always that of the type of component, so this had to be added manually. If a command can be executed from an access key, a relationship between that access key and the related components will be made.

5.4 Visualizing infrastructure

Neo4j has its own query language (Cypher[7]) and a web interface which can be used to interact with the database (similar to PhpMyAdmin, but integrated). This web interface allows you to query the database and visualize the output, which is what we use for the visualization in this project.

Figure 2: Output of Neo4j query for example infrastructure.



An example of this visualization is shown in Figure 2. The infrastructure shown here is a version of testing infrastructure used during this project. The directional edges between the component-nodes (all colors apart from the green nodes) indicate that the originating node contains the destination node, e.g. all security groups are contained in one of the two VPCs. The edges from the access keys (the green nodes) to the different components indicate that that access key can in some way access the destination node.

All nodes and edges have properties. The properties of the nodes offer more information about that component or access key. In the case of the components it will show the information that was retrieved about that component from the AWS CLI, in the case of the access keys it shows the Secret Access Key and potentially a token. The properties of the edges from the access keys indicate what commands can be executed on that component with that access key.

The figure shown above shows all types of components currently in the database. This is an example of what Neo4j can do. Just like with other query languages, you can query the database to only include certain types of nodes or relationships, based on label (e.g. 'EC2' or 'VPC') or properties (not shown in Figure, but an example would be 'VpcId' for the VPC nodes). This allows the user to query the database based on the components or relationships they are interested in. In Listing 2 the properties of such a node are shown, in this case those of one of the security groups. These are all the properties about this security group that `EC2:DescribeSecurityGroups` offers. The edges from the access keys to the components include as property with which command they were able to reach the component. The edges between components do not have further properties, as all properties about these relationships are kept inside the nodes. This is where that information is also contained in the AWS API.

```
Description: Created from the RDS
    Management Console: 2018/06/11 16:40:06
GroupId: sg-xxxxxxxx
GroupName: rds-launch-wizard
IpPermissionsEgress_0_IpProtocol: -1
IpPermissionsEgress_0_IpRanges_0_CidrIp: 0.0.0.0/0
IpPermissions_0_FromPort: 3306
IpPermissions_0_IpProtocol: tcp
IpPermissions_0_IpRanges_0_CidrIp: xxx.xxx.xxx.xxx/32
IpPermissions_0_ToPort: 3306
OwnerId: xxxxxxxxxxxx
VpcId: vpc-xxxxxxxx
```

Listing 2: Properties of security group in database

6 Conclusion

Our research question was: **"Given an infiltrated AWS component, what part of the related infrastructure would an automated tool be able to index?"** It would be impossible to answer this question with something like a percentage, as this all completely depends on the access key(s) acquired, what permissions they have, and whether they overlap. However, AWS does offer certain default policies to be attached to users (or other entities for that matter) in the following roles[2]:

AdministratorAccess	PowerUserAccess
Billing	SecurityAudit
DatabaseAdministrator	SupportUser
DataScientist	SystemAdministrator
NetworkAdministrator	ViewOnlyAccess

Out of those, only Billing does not give us any access to information about the infrastructure. AdministratorAccess, DataScientist, PowerUserAccess, SecurityAudit, SupportUser, SystemAdministrator and ViewOnlyAccess give us all information our current tool uses for infrastructure indexing. DatabaseAdministrator and NetworkAdministrator both still offer most information, but NetworkAdministrator falls short when it comes to listing the databases, and DatabaseAdministrator cannot list EC2 instances. However, in general all these roles would be able to list a substantial amount of the infrastructure.

The main takeaway of this project is that AWS's default policies allow for a relatively large amount of enumeration. This is among other things due to the fact that a large amount of `*:Describe*` commands, which return information about all components of a specific type, cannot be restricted on a resource-level, meaning that if someone can access `EC2:DescribeInstances` they will be able to enumerate **all** EC2 instances, including a large amount of information that can then be used in the infiltration of those instances.

In terms of the access keys and their permissions, it seems that access keys do not need to be very privileged to be useful in terms of indexing the infrastructure. While higher level privileges might be useful from an infiltration perspective, the enumeration of components and their relationships does not benefit from this per se.

The tool created during this project is open source and available on Gitlab.[2]

[2]AWS Infrastructure Analysis - gitlab.com/PeterBennink/aws-infrastructure-analysis

7 Discussion

Theoretically, every component of every infrastructure can be enumerated via the AWS CLI, which is the method of interaction with AWS that we used for this project. We conclude that indexing an infrastructure is definitely possible, and that the level of access given to the obtained access keys is of great importance. While it is difficult to make a statement about this possibility of enumeration for all infrastructures, access keys and/or components, some conclusions can be drawn, as has been shown in the previous section. Based on these conclusions we are at least able to say that it would be worthwhile to pursue the development and research behind this tool further.

7.1 Future work

The tool created during this project can be expanded, refined and improved upon in a vast number of ways. While this is mainly due to the vastness of AWS as a service, other factors have also contributed to this. The decisions made during this project were influenced by the limited time and the specific research question, and in some cases would have been different if this tool would be developed over a larger amount of time.

Path finding The most obvious improvement to this tool would be automated path finding, whereby you can input two nodes and get the shortest path between these two nodes as output, which can be used in finding out further steps in the process of infiltration. This is a feature in Linkurious[14], which allows users to interact with their graph-based data in an intuitive and graphical way. Bloodhound uses Linkurious.js for this purpose, which has been deprecated since 2016 and integrated in a new tool by the same organization called Ogma[3]. Integrating this would thus be a logical next step in the development. However, after inquiring with Linkurious it seems the minimal costs for using this library would be much more than would be affordable. Another solution with similar functionality will thus have to be found.

Further testing A second potential improvement would simply be further testing of this tool on larger infrastructures. Currently we have only tested this on our own testing infrastructure, which we kept expanding during the infrastructure, but further testing could reveal new ways in which the tool should be improved. While not in time for this project we are in talks with 3rd parties to test this tool on their infrastructure.

[3]Ogma - Linkurious

Penetration testing toolkit The tool created for this project is, in its current form, exclusively meant for reconnaissance. However, one way in which this tool could become part of a larger toolkit is by creating something comparable to Metasploit, but for AWS. Functionality in this toolkit could include scanning for and exploiting commands that can be used in privilege escalation, automatically scanning S3 buckets or EC2 volumes for high entropy contents (which could contain access keys or SSH private keys), or performing dictionary attacks to find usernames or bucket names.

Expansion An obvious improvement would be making the tool compatible with more AWS services. While the services in the scope of our project are among the most often used services[4], a lot of infrastructures will also use other services, and even our testing infrastructure uses more services than we analyzed for this project. An example of an interesting service to include is STS (Security Token Service)[5] which among other things allows an entity with an access key to assume the role of another entity, which might have more or different privileges.

Intelligent bruteforcing The permission bruteforcer created for this project checks a list of commands specified beforehand. It will check this whole list for each access key given. A more silent way to do this would be to prioritize the commands, and let the user of the tool choose how 'loud' the tool can be, much like the T-flag in Nmap allows its user to indicate the timing intervals between the different probes to stay under the radar of, for example, intrusion detection systems.[5] AWS allows an administrator to see which commands are executed using which access keys. If suddenly every possible command gets executed from multiple access keys this should raise some alarms. Prioritization could help prevent these alarms, making the reconnaissance more silent.

Resource-level permissions In its current form the bruteforcer only tries commands that do not support resource-level policies. Adding commands that can be restricted at resource-level would involve trying each command, for each access key, for each component that that command can be used on.

[4]Most popular AWS products of 2017 - globenewswire.com
[5]Nmap manual: "Timing and Performance" - nmap.org

References

[1] Aws ec2 documentation. https://aws.amazon.com/documentation/ec2/.

[2] AWS Managed Policies for Job Functions. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_job-functions.html.

[3] Aws rds documentation. https://aws.amazon.com/documentation/rds/.

[4] Aws s3 documentation. https://aws.amazon.com/documentation/s3/.

[5] Aws sts documentation. https://docs.aws.amazon.com/STS/latest/APIReference/Welcome.html.

[6] Cedric van bockhaven, private communication.

[7] Cypher Query Language Developer Guides & Tutorials. https://neo4j.com/developer/cypher/.

[8] EC2 Instance Metadata and User Data. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html.

[9] Granting IAM Users Required Permissions for Amazon EC2 Resources. https://docs.aws.amazon.com/AWSEC2/latest/APIReference/ec2-api-permissions.html#ec2-api-unsupported-resource-permissions.

[10] What is AWS? Amazon Web Services. https://aws.amazon.com/what-is-aws/.

[11] Andresriancho. andresriancho/nimbostratus. https://github.com/andresriancho/nimbostratus, Feb 2014.

[12] BloodHoundAD. BloodHound - Github. https://github.com/BloodHoundAD/Bloodhound/wiki.

[13] duo labs. Cloudmapper - Github. https://github.com/duo-labs/cloudmapper, Jun 2018.

[14] Linkurious. The graph intelligence platform. https://linkurio.us/.

[15] Jinesh Varia and Sajee Mathew. Overview of amazon web services. *Amazon Web Services*, 2014.

A List of the commands bruteforced

```
EC2:AssociateVpcCidrblock
EC2:CreateDefaultVpc
EC2:DescribeImages
EC2:DescribeSecurityGroups
EC2:DescribeVolumes
EC2:DescribeVolumesModifications
EC2:DescribeVolumeStatus
EC2:DescribeVpcClassicLink
EC2:DescribeVpcClassicLinkDnsSupport
EC2:DescribeVpcEndpointConnectionNotifications
EC2:DescribeVpcEndpointConnections
EC2:DescribeVpcEndpoints
EC2:DescribeVpcEndpointServiceConfigurations
EC2:DescribeVpcEndpointServices
EC2:DescribeVpcPeeringConnections
EC2:DescribeVpcs
RDS:DescribeDBInstances
RDS:DescribeAccountAttributes
RDS:DescribeCertificates
RDS:DescribeDBClusterSnapshots
RDS:DescribeDBInstances
RDS:DescribeEventCategories
RDS:DescribeEvents
RDS:DownloadCompleteDBLogFile
S3:ListBuckets
```