Cloud Custodian Security Audit 2023

Security Audit Report

(Arthur) Sheung Chi Chan, Adam Korczynski, David Korczynski

17 apr 2024

Contents

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tool development.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog and maintain a YouTube channel with educational videos. You can also follow Ada Logics on Linkedin and X.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

Project dashboard

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
(Arthur) Sheung Chi Chan	Auditor	Ada Logics Ltd	arthur.chan@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Kapil Thangavelu	Cloud Custodian Maintainer	Cloud Custodian	kapil@stacklet.io
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org

Executive summary

In late 2023 Ada Logics conducted a security audit of Cloud Custodian. The audit was facilitated by the Open Source Technology Improvement Fund (OSTIF) and funded by the Cloud Native Computing Foundation. Cloud Custodian is a command line tool that allows users to manage cloud resources across multiple cloud ecosystems by way of yaml policies. Cloud vendors require developers to define policies by way of programming languages such as Python, and developers will often need to write policies specifically for a particular cloud environment. With Cloud Custodian, users can write their desired policies in a cloud-agnostic format - yaml - and depend on Cloud Custodian to translate this into effective, cloud-specific policies.

Ada Logics began the engagement by formalizing a threat model for Cloud Custodian. The threat model was helpful to us as we audited the source code. Once we had initiated the threat model, we continued iterating over it throughout the entire audit as we learned more about the project. With the first version of the threat model, we began the manual review. In this part of the audit, we looked at a range of threats to Cloud Custodian. We also began work on setting up a fuzzing suite for Cloud Custodian, which included writing fuzzers for Cloud Custodian and settig up the infrastructure for the fuzzers to run continuously.

In summary, during the engagement, we:

- Formalized a threat model of Cloud Custodian
- Audited Cloud Custodians code base for security vulnerabilities.
- Integrated Cloud Custodian into OSS-Fuzz.
- Wrote a targeted fuzzing suite for Cloud Custodian.

Threat model

In this section we present the findings from threat modelling Cloud Custodian. We first enumerate Cloud Custodians main components, tools and dependencies. We then proceed to an overview of a crucial part of Cloud Custodian: policies. After that, we introduce the data flow of a Cloud Custodian deployment at a high level. Here, we show how data and trust travels through Cloud Custodian and where trust changes. Finally, we enumerate the threat actors that could impact the security of Cloud Custodian.

Each aspect of the threat model gives a different perspective to Cloud Custodians security model; When we later consider specific security issues found during the manual auditing goal, having enumerated the threat actors allows us to consider potential security risks against an attacker. The threat actors allow us to ask questions such as “who has privileges to trigger this issue?”. The data flow findings are helpful for the process of auditing, ie. to identify particular exposed components of Cloud Custodian, as well as when considering the threat from a particular finding. The trust flow analysis locates the threat actors and attack vectors that a given vulnerability is reachable to.

The Cloud Custodian policies are a core component of Cloud Custodians security model. We therefore go into deeper detail with these in the threat model and consider them from different perspectives.

The threat model is intended to assist multiple audiences. First and foremost, the threat model helps the auditors of this particular security audit. At a higher level, that translates to be helpful to other security researchers that wish to review Cloud Custodians security posture independently of this security audit conducted by Ada Logics. For this community, the threat model gives an advanced entry into the internals of Cloud Custodian as well as a high-level view of Cloud Custodians threat model. Secondly, the threat model can be a perspective to the Cloud Custodian maintainers when assessing community-based security disclosures; It helps consider the same angles of a vulnerability report as a security researcher takes. Thirdly, the threat model can also be of interest for Cloud Custodian users that notice unexpected behavior in their use cases. The root cause of such cases can be bugs or security vulnerabilities, and becfore disclosing these publicly, users can assess them from Cloud Custodians threat model.

Main components in scope

Cloud Custodians code base is roughly split into two high-level categories: 1) a list of core packages with supporting tools and 2) providers for cloud services. The packages vary in maturity from sandbox packages or components under development to mature, production-grade packages. The sandbox and development packages were out of scope for the audit. The table below shows all the Cloud Custodian components in scope for the audit.

Core package

Components	Description
c7n	Core package for the cloud custodian project. It contains the cli.py and commands.py to accept command line calls to the cloud custodian and act as the front end of the tools. It also has different supporting classes to handle the parsing and execution of requests or queries with the supporting schemas.
c7n/actions	Part of the policy handler, handles the actions to be performed on the filtered target services or components. It can be changed or simply queried.
c7n/filters	Part of the policy handler, it handles the choice requirement to filter out the target services or components to which this policy will apply.
c7n/reports	Part of the policy handler, it displays the report showing the result of the previous policy execution with the given policy.
c7n/resources	Template schema and other configurations and controls classes for different cloud services actions, queries, and requests

Cloud service providers

Components	Description
tools/c7n_azure	Custodian provider package to support Azure
tools/c7n_gcp	Custodian provider package to support GCP
tools/c7n_kube	Custodian provider package to support Kubernetes
tools/c7n_oci	Custodian provider package to support OCI
tools/c7n_tencentcloud	Custodian provider package to support Tencent cloud
tools/c7n_left	Tool to evaluate policies with chosen cloud services without actually executing them

Supporting tools

Components	Description
tools/c7n_mailer	Tool to handle message relay and mailing service across different cloud accounts and services
tools/c7n_policystream	Tool to manage and control policies changing history with version control

Third party components

Components	Description
Python poetry	Package and dependencies management library
Cloud service API	Cloud platform service API for supporting cloud environment, like BOTO3 for AWS.

Cloud Custodian policies

The core functionality of Cloud Custodian is translation of into YAML commands applicable for different cloud enviroments. Users pass their YAML policies to Cloud Custodian, and Cloud Custodian deploys the policy using a scripting language suited for the users cloud environment.

Policies are declarative using YAML files that follow a predefined schema predefined and performs some actions on resources on the cloud servers that match certain criteria. The action can be immediate actions, scheduled activity or policy rules to be enforced. Different actions require different levels of permissions and roles depending on the users cloud environment.

Each cloud provider uses different mechanisms for deploying this type of functionality into their platforms. Learning these mechanisms takes time and effort, and users will have to learn these mechanisms for each cloud provider they use. Cloud Custodian abstracts these mechanisms into a unified approach supporting different cloud platforms, allowing Cloud Custodian users to use the same declarative approach for all support cloud platforms.

The main purpose of Cloud Custodian is to use a unified language to perform policy enforcement and actions on the cloud services without the need to develop specific scripts and execution requests through different cloud service APIs of different cloud providers. In general, Cloud Custodian provides an abstraction and abstracts basic syntax of resources, filters, actions, API calls, business logic, and other cloud features into declarative YAML-based policies.

Policy validation

An important part of Cloud Custodian is to parse policies according to schema to different resource filters and actions to the cloud service. The correctness and security of policy parameters and definitions are important factors in maintaining functional and security-relevant expectations on Cloud Custodian to properly and securely translate YAML-based policies into cloud scripts. The major interpretation and translation of Cloud Custodian are located in the PolicyLoader class. It takes care of the policy interpretation according to the schema. It also complies with the policies to the permissions and roles of the CLI user pushing the policies. In addition, it also has a validation process to ensure the policies are fulfilling the needed settings of the schema and that they do not contain illegal injections or characters in the provided settings.

Policy continuous integration

Cloud Custodian provides webhood for continuous integration from version control like git repository. It can be added to the Cloud Custodian CLI in the cloud services to activate webhook monitoring to a git repository. Whenever there are new or updated policies being merged. It will trigger the webhook and upload the policies to the policies storage in the cloud after validating the policies. It will then apply to the cloud service if the permissions and roles are allowed. The security responsibility may pass partly to the git repository management to only allow certain people to merge the policies and also requires checking of possible injections or unwanted actions in those policies. Besides, there are also additional uses of webhooks. Webhook. Periodic policies with webhook action type could be added to Cloud Custodian to point to some URL or repository location. HTTP requests are initialised to retrieve resources or filtering information from the URL or repository when the policies are processed. No new policies need to be deployed if the resource status or filtering criteria are changed. This could be applied to cloud servers with frequent policies, resources or configuration changes.

Data flow for Cloud Custodian

In this section we present the dataflow of Cloud Custodian.

A high-level overview of dataflow and trustflow in Cloud Custodian looks as such:

Policy developers develop Cloud Custodian policies according to existing schemas in the Cloud Custodian service. CLI users retrieve those policies and apply them through Cloud Custodian CLI for immediate or periodic plans, rules or actions on filtered resources. The CLI users need to have enough permissions and roles on the target resources in order to apply those policies in the cloud servers. In addition to using existing schemas, Cloud Custodian allows custom schemas for uncovered cloud

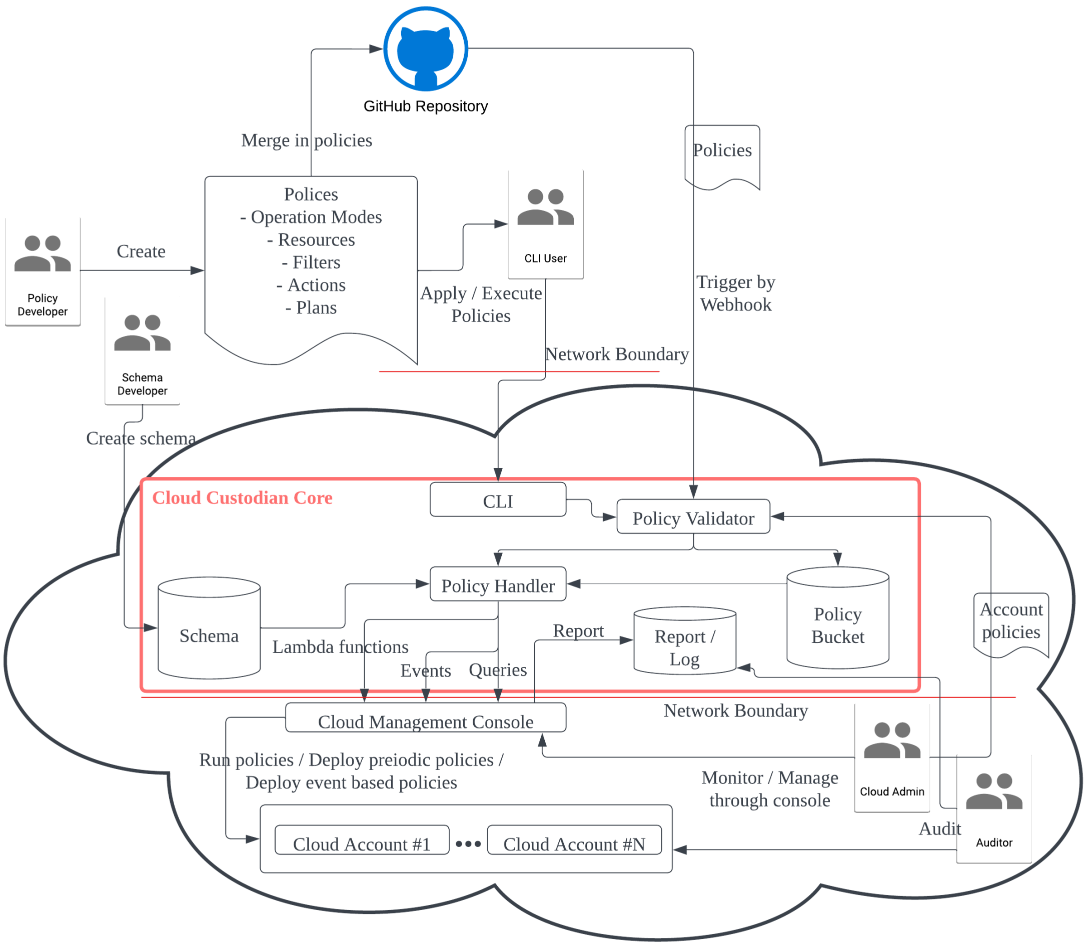


Figure 1: Data flow for Cloud Custodian from CLI entry point

services API combinations, that is, if Cloud Custodian adopters have a specific use case to their need, they can write their own schemas.

When a user or an automated service ("Github Repository" in the diagram) deploy policies, the Policy Validator validates them for correctness of the policies and that the CLI users permissions correspond to the permissions in the schemas. After the Policy Validator has validated a policy, Cloud Custodian policy handlers translate the valid policies to different types of cloud service requests, target filters or lambda functions and push them to the cloud services storage and register triggering events or execute them immediately. Depending on the actions from the policies, the cloud server may store policies as periodic events, or perform immediate actions. Sometimes it will also enable continuous logging or returning the current snapshot data of the cloud servers on request or storing them in the Cloud bucket. The cloud admin can control the stored policies and also the IAM roles for both the policies and the CLI users that could deploy policies. Lastly, If need be, auditors can audit the logs and some long term triggering events of policies to ensure it complies with the consumers internal or legal regulations.

Entry through Cloud event triggering

In this section we cover Cloud Custodians event-based triggering of policies.

At a high level, Cloud Custodian supports three types on invocations besides directly invoking a policy:

1. Event-triggered policies When an event-triggered policy runs through Cloud Custodian CLI, it will register event filterings in the given cloud service with certain patterns on the CloudWatch event bus. When a change happens on resources or services, the CloudWatch bus will stream a System event to registered Lambda services, if the patterns filter is matching, the lambda functions will be invoked. This kind of policy deployment allows Cloud Custodian to deploy translated actions into lambda functions and store them in the cloud service. The functions are then registered to be invoked when a certain CloudWatch Events API has been called. This is done by pattern filtering in the Cloud Custodian policies and is translated to lambda functions that determine if the streamed system event matches the invocation requirements of the functions.

2. Periodic-triggered policies Periodic-triggered policies are similar to event-triggered policies in which the lambda functions invocation is also triggered by CloudWatch Event API. But instead of triggering based on certain resources or service changes status, Cloud Custodian registers a periodic event in the CloudTrail to periodically send triggering requests to CloudWatch Event API and trigger a new system event. This kind of periodic registration could also be filtered by patterns and resources/services targets, which could be configured by the Cloud Custodian policies.

3. Config rules triggered policies Besides the above two types of event triggering lambda function invocation, there is an additional config rules triggered policies. These types of policies trigger the invocation of stored lambda functions when a certain service or resource configuration has been made on the cloud services. The policies register a certain config rules monitoring in the cloud services, wherever there is a configuration change, it will match the changes with the rules registered and invoke the target lambda functions if the rules match.

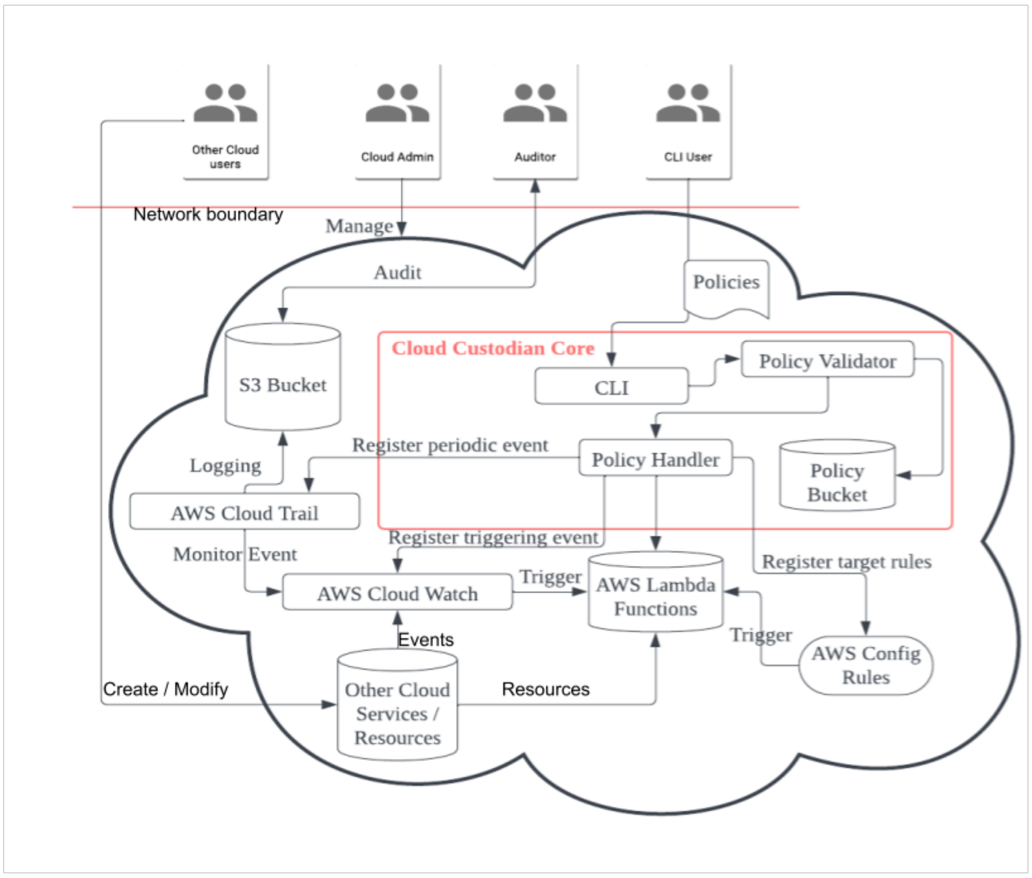


Figure 2: Data flow for Cloud Custodian to register triggering criteria for policies

Once a policy has been translated into a lambda function on a cloud platform, the Cloud Custodian user can set up their environment in such a manner that the lambda function gets invoked when an event takes place. In the case of AWS, users can for example use the AWS CloudTrail, CloudWatch and config rules services to deploy event-triggered lambda functions in their cloud. The translated lambda functions are stored in the AWS Lambda Services with roles and permission enforced on the invocation of those stored lambda functions. There are 3 types of ways supported by Cloud Custodian to trigger the events. But it is also important to remark that once the lambda functions have been created and deployed to the AWS lambda services, any IAM roles of users or triggered events match the one enforced on those lambda functions, the lambda functions would be invoked. That opens a possibility that a lambda function deployed through this setting through Cloud Custodian may trigger an unexpected event if it matches some of the events or configuration rules. Users can perform additional monitoring of the execution and triggering of these stored lambda functions can be done by specifying monitoring and reporting options to the CloudTrails through Cloud Custodian policies, then log or other metrics are stored in S3 Bucket and could be audited or viewed in a later stage.

Trust Boundaries

As the Cloud Custodian is installed in the management services of the cloud servers, the CLI users certainly need to connect to the target cloud servers to execute the policies. Thus the policies are surely going through the network boundary from the local environment of the CLI users to the remote cloud management servers with Cloud Custodian installed. The other possible network boundary is the boundary between the Cloud Custodian and the target cloud services. Although they should be located in the same cloud network which could be manageable by the same cloud management console, the management server and the target cloud services may be located in different subnetworks. That creates another possible network boundary.

Threat actors

Cloud custodian is assumed to be run in a cloud console where all the actors should be either trusted or controlled by the Cloud IAM service.

Threat Actor	Description	Level of trust
CLI Users	Users that push policies to cloud services to perform actions on the filtered target resources with their own IAM roles	High
Cloud Admins	Users that monitor and manage the cloud server where Cloud Custodian is running.	Full
Policy Developers	Users that create Cloud Custodian policies to be deployed to the cloud server.	High
Schema Developers	Users that create schema or plugins to extend the functionality of Cloud Custodian.	Low

Threat Actor	Description	Level of trust
Auditor	Users to monitor and audit logs, policy status, periodic events and rules to ensure overall security and accuracy of the policy enforcement.	Low
Other cloud users	Some Cloud Custodian policies may depending on other cloud services or events. Thus other cloud users with different IAM roles that control some cloud services or events could trigger some stored Cloud Custodian policies.	Low

Attack surface

In this section we present Cloud Custodians attack surface.

The attack surface represents the entrypoints into a system that an attacker could utilize to compromise the system. The attack surface describes both the known and expected entrypoints as well as unexpected or unintended entrypoints. Delineating the attack surface is helpful in understanding the types of threats and threat actors that could negatively impact the system. In this section we present the attack surface of Cloud Custodian; Later in the audit, we use the findings from the threat modelling detailing the attack surface in the manual auditing goal of the audit.

A fundamental property of Cloud Custodians security model is that it is deployed in a cloud environment such as AWS, Azure, Google Cloud and Kubernetes. Cloud Custodian inherits substantial parts of the security model of these platforms. As such, these platforms also represent an attack surface for Cloud Custodian; If there are vulnerabilities in the underlying platform, Cloud Custodian can be vulnerable to these as well. In other words, if an attacker can compromise the underlying cloud platform, Cloud Custodian has no or few defense mechanisms between itself and the cloud platform. During our analysis of Cloud Custodian, we found several indicators that Cloud Custodian accepts this inherited attack surface. For example, Cloud Custodian decompresses the user data of EC2 instances without checking its size or buffering the output (Line 817-818 below):

https://github.com/cloud-custodian/cloud-custodian/blob/50d9f139de4a78aa32766f86f64e438cf7a8158a/c7n/resources/ec2.py#L803-L821

```
803     def process_instance_set(self, client, resources):
804         results = []
805         for r in resources:
806             if self.annotation not in r:
807                 try:
808                     result = client.describe_instance_attribute(
809                         Attribute='userData',
810                         InstanceId=r['InstanceId'])
811                 except ClientError as e:
```

```
                        if e.response['Error']['Code'] == '
                            InvalidInstanceId.NotFound':
                            continue
                    if 'Value' not in result['UserData']:
                        r[self.annotation] = None
                    else:
                        r[self.annotation] = deserialize_user_data(
                            result['UserData']['Value'])
                if self.match(r):
                    results.append(r)
            return results
```

Cluster users with lower privileges can control EC2 instances and craft a malicious EC2 instance with user data that decompresses into a large blob size would cause denial of service of Cloud Custodian, since it would read the decompressed blob into memory. Cloud Custodian does not guard against this attack but is also not vulnerable to it, because the limit of the user data of EC2 instances on the underlying platform - AWS - is 16KB. As such, to cause a denial of service, an attacker needs to find a way to circumvent the restriction on AWS's max allowed size on EC2 user data.

This is an example of how Cloud Custodian integrates into the underlying platform to use their hardening in its own security posture. There are pros and cons to this. On the pros side, Cloud Custodian benefits from the security work made on the cloud platforms. Leaving out hardening mechanisms and relying on the hardening made by the cloud platforms avoids bloat of the Cloud Custodian code base. Furthermore, Cloud Custodian avoids hardening on an attack vector that is expensive for attackers to compromise through. On the cons side, if an attacker can compromise the underlying cloud platform, they have a large attack surface available and can cause a lot of havoc for Cloud Custodian users. In addition, Cloud Custodian is susceptible to issues arising from unexpected or unnoticed changes in the resource limits and sanitization for cloud resources.

Another attack vector is by way of cluster resources, ie. by creating or modifying cluster resources in such a way that when Cloud Custodian queries them, Cloud Custodian will behave unexpectedly - possibly in an insecure way. This attack vector requires cluster privileges for the attacker, albeit minimum create or edit-privileges are sufficient. Cloud Custodian can handle resources insecurely when running policies once or periodically, and a cluster user can craft a resource that causes unexpected behavior in Cloud Custodian and triggers security vulnerabilities. This type of attack requires a CLI user to write and/or deploy policies, and the attacker would need to know which Cloud Custodian policies are deployed. Cloud Custodian users with multiple users of varying vertical permission levels are prone to this attack vector. An sample attack would progress as such:

1. Cluster admin creates a cluster user without `create` privileges but with `edit` privileges.
2. Cluser admin deploys a policy that periodically queries all EC2 instances in the cluster.
3. The cluster user created in step 1 knows that the policy created in step 2 mishandles an EC2 instance with a particular name, and instead of reading the name, it creates 100 new EC2 instances

that are not restricted by any limitations from policies in the cluster. The cluster user creates the EC2 instance with the malicious name.

4. The cluster user has managed to create 100 EC2 instances with their own specifications and has thereby succesfully escalated privileges.

From the perspective of a standard Cloud Custodian use case, this is an exposed attack surface, since it originates from an intended way of using Cloud Custodian. Cloud Custodian should be able to handle a user base with different permission levels without exposing itself to privilege escalations. We note that this attack surface also relies on authorization mechanisms of the underlying cloud platform, however, this attack surface is exposed to cluster users even when the authorization mechanisms work as intended; If the user has permissions to create EC2 instances, they do no need to escalate privileges and be able to delete EC2 instances or create S3 buckets in order to exploit a vulnerability in cluster resource handling.

Cloud Custodian allows deploying policies that have no immediate effect and are only triggered by certain cloud events, these policies are translated and stored in the cloud as stored actions (lambda function in AWS) or stored policies in the policy bucket. For example, Cloud Custodian can register event monitoring and trigger in AWS through AWS Cloud Trail surface. If the registered cloud event has happened, the AWS Cloud Trail invokes the stored actions (Lambda functions in AWS) with the details of the cloud events. If the stored actions translated from Cloud Custodian policies use some of the information from the cloud events without proper validation or sanitization, they are vulnerable to attacks like injections or Denial-of-service. The actions could also cause unexpected behaviours or privilege escalation if the stored actions are invoked with IAM roles with higher privileges. An attacker could observe or guess the stored and cloud event registered Cloud Custodian policy actions and specifically trigger a cloud event with malicious data. In this situation, the attacker only requires IAM roles that have permission to trigger that cloud event and still invoke the stored Cloud Custodian policy actions even if those policy actions are executing in different IAM roles.

Attack surface	Description
Cloud Custodian CLI	Cloud Custodian CLI is one of the entry points for accepting policies and performing actions on the cloud with the user's IAM roles. Privilege escalation or injection targeting the cloud could go passed an insecure or over-privileged CLI.
Cloud Custodian Policy	Cloud Custodian defines what actions are performed on what resources are in the cloud. Attackers could target to trick legitimate users into creating a policy to complete some unexpected actions in the cloud.

Attack surface	Description
Cloud Custodian schema	Cloud Custodian schema translates a YAML policy into actions in the Cloud environment. Adopting third-party or custom schema to Cloud Custodian could open up unexpected translations and perform unexpected actions in the cloud.
Cloud triggering events	Cloud Custodian allows deploying policies that have no immediate effect and are only triggered by certain cloud events. Changes in the existing cloud events could cause unexpected policies to be triggered.
Cloud IAM services	Cloud Custodian policies are running with the IAM roles of the user deploying the policies. Over-privileged IAM roles for Cloud Custodian could allow privilege escalation on the cloud.

Attacker objectives

Privilege escalation in the cloud

Most of the privilege management in the cloud is done by IAM services. Attackers may make use of the Cloud Custodian to escalate its privilege in the cloud if the IAM roles for the Cloud Custodian are not correctly configured or over-privileged.

Mess up with services and resources of the cloud

With enough privilege, the Cloud Custodian could perform almost any action in the cloud. Attackers may make use of the high-privilege IAM role of Cloud Custodian to perform some unexpected actions in all other cloud services.

Gain information from the cloud services and resources

With enough privilege, the Cloud Custodian could perform almost any action in the cloud. Attackers may make use of the high-privilege IAM role of Cloud Custodian to retrieve information or data in some protected resources.

Fuzzing

As part of the audit, Ada Logics wrote a fuzzing suite for Cloud Custodian consisting of 11 fuzzers targetting primarily resource handling and validation methods for resources from multiple cloud providers that Cloud Custodian supports. In addition, Ada Logics built the infrastructure to suppor continuous fuzzing; We did that by integrating Cloud Custodian into OSS-Fuzz which is an open-source project created and run by Google that offers automation of different aspects of a healhty fuzzing workflow. This includes running the fuzzers periodically, reporting of found bugs by the fuzzers, testing for bug fixes and more - all done in an automated manner with technical details available for bugs such as stack traces and reproducer testcases. A continuous fuzzing setup is an important part of software security and upon completion of this audit, Cloud Custodian is fuzzed continuously, even after the audit has concluded.

In this audit we took the high level approach of adding coverage to validation, processing and parsing routines to resource types, actions and filters. This resulted in multiple alike fuzzers organized by provider or subdirectories. The goal was to add coverage to class methods that processing input, which for some classes has higher complexity than others. Some of the calls in the fuzzers are to methods in Cloud Custodian that are not complex compared to other classes. Nonetheless, we added coverage to those methods to add as much coverage to validation, processing and parsing as time permitted in this audit. The fuzzing work done by Ada Logics in this audit was Cloud Custodians first step into adding fuzzing its codebase, and adding coverage was the main priority of our fuzzing work.

All fuzzers can be found in Cloud Custodians OSS-Fuzz project folder: https://github.com/google/oss-fuzz/tree/master/projects/cloud-custodian. OSS-Fuzz builds builds the fuzzers using its Dockerfile and build script. The `Dockerfile` moves the fuzzers to the Docker environment at build time, and the build script builds the fuzzers.

These are the fuzzers, we have written during the audit:

fuzz_actions_parser

This fuzzers adds coverage for the `parse` method of the `ActionRegistry` class.

fuzz_actions_process

This fuzzer tests the `process` methods of multiple `actions` classes: 1) `AutoTagUser`, 2) `Notify`, 3) `LambdaInvoke`, 4) `Webhook`, 5) `AutoScalingBase` and 6) `PutMetric`. In each fuzz iteration, the fuzzer will choose the class to test, create an object of the class using the data from the fuzzer and invoke the objects `process` method. For `AutoTagUser` and `Notify`, the fuzzer will first validate the object before invoking their process method.

fuzz_actions_validate

This fuzzer targets the `validate` methods of 3 classes of the `c7n/actions` sub directory: 1) `AutoTagUser`, 2) `ModifyVpcSecurityGroupsAction` and 3) `Notify`. The fuzzer chooses which class to create an object from in each iteration and creates and object using the fuzz testcase. Finally, the fuzzer invokes the objects `validate` method.

fuzz_filters_parser

Tests the parsing routines of the `c7n/filters` subdirectory. This includes the `parse` methods of the `FilterRegistry` class, as well as multiple parsers from the offhours filter. The fuzzer decides whether to fuzz either the `FilterRegistry` `parse` method or two `parse` methods of the offhours filter in each fuzz iteration.

fuzz_filters_process

Tests the `process` methods of the core filters (the `c7n/filters` subdirectory). The fuzzer tests in total 12 classes, and picks one to test in each iteration. The fuzzer creates an object of the picked class using the fuzz testcase. When invoking the `process` method, the fuzzer will pass a dictionary The 12 classes that the fuzzer tests the `process` methods of are: 1) `MultiAttrFilter`, 2) `Missing`, 3) `OnHour`, 4) `OffHour`, 5) `SubnetFilter`, 6) `NetworkLocation`, 7) `Diff`, 8) `ConsecutiveAwsBackupsFilter`, 9) `HealthEventFilter`, 10) `HasStatementFilter`, 11) `CrossAccountAccessFilter` and 12) `AccessAnalyzer`. The fuzzer passes a series of resources to the `process` method of each object. This tests for edge cases in resources that could impact Cloud Custodian in a negative way concerning both its security and reliability. For selected classes, the fuzzer will validate the object before invoking its `process` method which mimicks Cloud Custodian production use case.

fuzz_filters_validate

Tests the `validate` methods of 10 `filters` classes located in the `c7n/filters` subdirectory of the Cloud Custodian source tree. These are the same classes that invoke their `validate` method before their `process` method in the `fuzz_filters_process`, however this fuzzer is more focused on validation.

fuzz_gcp_actions_validate_process

Tests validation and processing routines of four classes from the `c7n_gcp` tool. `c7n_gcp` enables GCP support in Cloud Custodian, and in this fuzzer we implement similar testing logic to the fuzzers testing the validation and processing of the core Cloud Custodian classes.

fuzz_gcp_filters_validate_process

Tests the validation and processing of GCP resources. The fuzzer instantiates a resource using the testcase of each iteration and invokes its `process` method. It validates four of the resource types before invoking their `process` method. The fuzzer tests the following resource types: 1) `LabelActionFilter`, 2) `RecommenderFilter`, 3) `GCPMetricsFilter`, 4) `SecurityComandCenterFindingsFilter`, 5) `TimeRangeFilter`, 6) `IamPolicyFilter` and 7) `AlertsFilter`. The fuzzer instantiates a resource manager and add an `ActionRegistry` and a `FilterRegitry` to it.

fuzz_gcp_resources_process

Tests the processing of GCP-specific classes, specifically 1) `ServerConfig`, 2) `SQLInstance`, 3) `KmsLocationKmsKeyringFilter`, 4) `EffectiveFirewall`, 5) `HierarchyAction` and 6) `AccessApprovalFilter`. In each fuzz iteration, the fuzzer creates one object of either of the 6 classes using the fuzzers testcase to instantiate the object. Next, the fuzzer sets up a `ResourceManager` and adds an `ActionRegistry` and a `FilterRegistry`. Finally, the fuzzer invokes the objects `process` method catching exceptions that the fuzzer should not report.

fuzz_query_parser

This fuzzer tests two parsers: 1) `QueryParser` and 2) `C7NJMESPathParser` - both from the `util` module. The fuzzer passes a unicode string of maximum 1024 in length.

fuzz_resources_parser

This fuzzer tests 5 parsing routines: 1) Cloud Custodians aws Arn parser, 2) the ec2 QueryFilter parser, 3) the health `QueryFilter` parser, 4) the `sagemaker` `QueryFilter` and 5) the `emr` `QueryFilter`. The fuzzer invokes one of the parsing routines in each fuzz iteration with a unicode string.

fuzz_resources_process

Tests the `process` methods of almost 60 core resource types. The fuzzer first selects a resource type, then creates an object of that type using the fuzzers input test case to generate a pseudo-random object. Finally the fuzzer invokes the objects `process` method.

fuzz_resources_validate

Tests the `validate` methods of almost 60 core resource types. The fuzzer first selects a resource type, then creates an object of that type using the fuzzers input test case to generate a pseudo-random object. Finally the fuzzer invokes the objects `validate` method.

SLSA review

ADA Logics carried out a SLSA review of Cloud Custodian. SLSA (https://github.com/slsa.dev) is a framework for assessing the security practices of a given software-project with a focus on mitigating supply-chain risk. SLSA emphasises tamper resistance of artifacts as well as ephemerality of the build and release cycle.

SLSA mitigates a series of attack vectors in the software development life cycle (SDLC) all of which have seen real-world examples of succesful attacks against open-source and proprietary software.

Below we see a diagram made by the SLSA illustrating the attack surface of the SDLC.

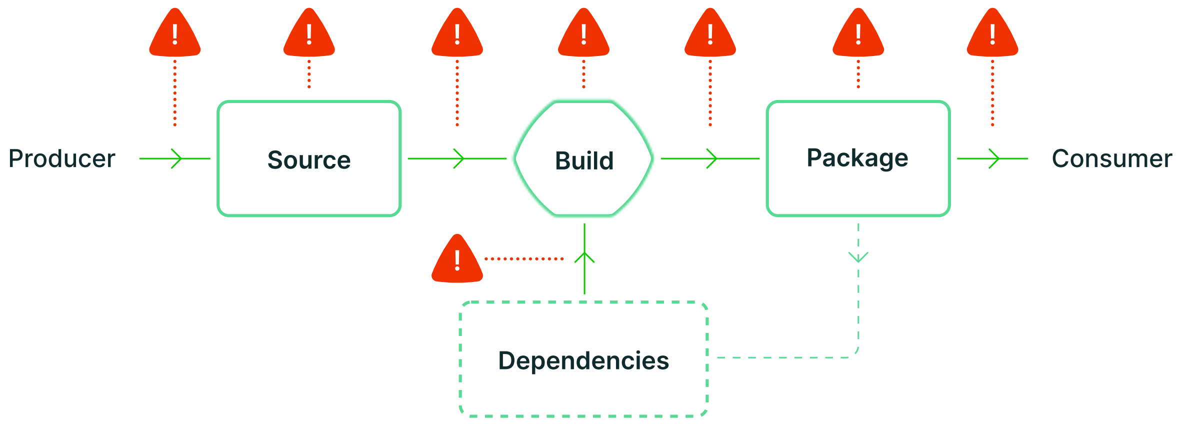


Figure 3: Data flow for Cloud Custodian to register triggering criteria for policies

Each of the red markers should different areas of possible compromise that could allow attackers to tamper with the artifact that the consumer invokes at the end of the SDLC.

SLSA splits its assessment criteria into 4, increasingly demanding levels. At a high level, the higher the level of compliance, the higher tamper-resistance the project ensures its consumers.

Cloud Custodian releases its binaries on Github Actions using .github/workflows/release.yml. Github Actions fulfills a number of criteria of SLSA. Github Actions provisions a fresh build environment for every build thereby fulfilling SLSAs requirement of isolation and hermeticity. These are great, and important features of a hardened build platform. The current version of SLSA emphasises these features of the build platform, but projects must have a provenance available to conform to SLSA Level 1. Cloud Custodian does not currently include a provenance statement with releases, and as such is currently at SLSA L0.

Cloud Custodians most important task in terms of SLSA compliance is to add a provenance statement to releases and gradually improve compliance of that provenance statement to higher levels of SLSA, such as making it verifiable. We recommend adding this using SLSAs slsa-github-generator (https://github.com/slsa-framework/slsa-github-generator).

Found issues

Here we present the issues that we identified during the audit.

#	ID	Title	Severity	Fixed
1	ADA-CC-2023-1	Use of a broken or weak cryptographic hashing algorithm for sensitive data	Low	No
2	ADA-CC-2023-2	Insecure temporary file creation	Informational	Yes
3	ADA-CC-2023-3	Missing sanitisation in using Jinja2 library	Low	No
4	ADA-CC-2023-4	Using deprecated and insecure ssl.wrap_socket	Low	No
5	ADA-CC-2023-5	Index out of range in ARN parser	Informational	Yes
6	ADA-CC-2023-6	Improper URL substring validation can leak data	Low	Yes
7	ADA-CC-2023-7	Possible DoS from attacker-controller Github account	Low	Yes
8	ADA-CC-2023-8	Possible DoS from attacker-controller Github repository	Low	Yes
9	ADA-CC-2023-9	Possible zip bomb from large S3 object	Moderate	No
10	ADA-CC-2023-10	Privilege escalation through chained Lambda functions in AWS	Moderate	No

Use of a broken or weak cryptographic hashing algorithm for sensitive data

Severity	Low
Status	Reported
id	ADA-CC-2023-1
Component	c7n resources

get_finding function and get_item_template function uses some weak cryptographic hashing algorithms like MD5 or SHA-1. Some of these usages are used for non-sensivite data, however, they might still have a minor impact of a potential attack vector. The get_item_template function uses MD5 to calculate a debup token for retrieving an item template for local storage or codebase for further processing. The weak MD5 hash allows a 2nd preimage attack which could make an attacker create another input that can produce the same hash and replace the original item template (or replace what item template is to be returned) for further processing, which may cause unexpected execution results.

get_finding function uses MD5 on policies to determine a `finding_id` in post finding operations of AWS security hubs function. Cloud Custodian can generate a policy to provision a lambda function that will process findings from AWS resources and act on them when certain criteria are matched. For example, it could process findings from AWS guard duty on all IAM users to remove their access keys if the key exists when not allowed. These policies generate a data set called `findings` to record the details of the target that needs to perform actions (or further monitored) on IAM resources. The code below uses the MD5 hashes of the policies data for the finding generation operation as its `finding_id`. This setting is vulnerable because an attacker could create different policies with the same hash in order to pretend it is the designated policy and thus result in incorrect finding target to be processed by future actions or periodic triggers.

https://github.com/cloud-custodian/cloud-custodian/blob/d458f0a/c7n/resources/securityhub.py#L525-L530

```
def get_finding(self, resources, existing_finding_id, created_at,
    updated_at):
...

        if existing_finding_id:
            finding_id = existing_finding_id
        else:
            ...
               finding_id = '{}/{}/{}/{}'.format(
                   self.manager.config.region,
                   self.manager.config.account_id,
                   # we use md5 for id, equiv to using crc32
                   hashlib.md5( # nosec nosemgrep
                       json.dumps(policy.data).encode('utf8'),
                       **params).hexdigest(),
                   hashlib.md5( # nosec nosemgrep
                       json.dumps(list(sorted([r[model.id] for r in
                         resources]))).encode('utf8'),
                       **params).hexdigest()
               )
           finding = {
            ...
               'Id': finding_id,
            ...
           }
         ...
          return filter_empty(finding).
```

https://github.com/cloud-custodian/cloud-custodian/blob/d458f0a/c7n/resources/ssm.py#L553

```
    def get_item_template(self):
...
        dedup = hashlib.md5(dedup).hexdigest()[:20]  # nosec nosemgrep
...
```

Mitagation

Change the use of `hashlib.md5` for the more secure hashing algorithms, like `sha3` or `shake` based algorithm.

Insecure temporary file creation

Severity	Informational
Status	Fixed
id	ADA-CC-2023-2
Component	ops/azure/container-host-chart

The deploy chart operation of Azure uses using deprecated code `tempfile.mktemp()` method which creates possible race condition attacks. The `tempfile.mktemp()` method is deprecated because it could create a race condition vulnerability, as mentioned by the official python tempfile library documentation. The `tempfile.mktemp()` method is divided into two steps. It first generates a random temporary filename and then creates the file with the newly generated temporary filename. Some other process could create a file in between the two steps of the `mktemp()` function with the same name generated by the first step of the `mktemp()` function. This creates a race condition situation and could result in unexpected results when later code accesses the created temp files. Attackers are able to make the chart write to a symbolic link with the same path that points to some sensitive files in the local storage with the privilege of the Cloud Custodian tool. In addition, attackers could also deny chart writing by changing the permissions of the temp files or inject malicious content into the temp files if they successfully generate a file with their own privilege before `mktemp()` function does.

Vulnerable code location

The write_values_to_file in deploy_chart.py of Azure provider uses a deprecated python functions `tempfile.mktemp()` which make the code vulnerable to race condition attack.

```
@staticmethod
    def write_values_to_file(values):
        values_file_path = tempfile.mktemp(suffix='.yaml')
        with open(values_file_path, 'w') as values_file:
            yaml.dump(values, stream=values_file)
        return values_file_path
```

Mitigation

The race condition vulnerability exists because the `tempfile.mktemp()` function divides the random temporary filename generation and the creation of the temporary file into two steps. To solve this vulnerability, just combine the two steps together by generating the file immediately. This could be done by custom logic or replacing `tempfile.mktemp()` with the newer `tempfile.mkstemp()` function which uses the steps combining approach.

Missing sanitisation in using Jinja2 library

Severity	Low
Status	Reported
id	ADA-CC-2023-3
Component	c7n mailer

In the utils.py of the c7n_mailer tool, the logics use the Jinja2 library to render an HTML email template. The functions `get_message_subject` and `get_jinja_env` take in email message parameters and path for the email template directory as input and use the Jinja2 library to create and render the HTML email template and email subject. The functions are triggered when a policy-registered event has happened and notify actions are required. The parameters passing to the functions are configurable by the policy owner and other cloud users who have access to either the environment or the resources linked to the registered event. Thus it could contain untrusted data and result in possible HTML injection and lead to possible cross-site scripting (XSS) when malicious data is being attached to the template or environment variables. According to the documentation of Jinja2, the default configuration for their Template designer does not have automatic HTML escaping because it may be a huge performance hit if it needs to escape all variables, including some variables that are not HTML. In addition, the logic in the `utils.py` of the c7n_mailer tool does not enable HTML escaping by default when using the Jinja2 package nor manually escaping the variable from the Jinja2. Thus it results in possible HTML injection.

The get_message_subject function in `utils.py` of the c7n_mailer tool uses jinja2 library without HTML escaping.

```
def get_message_subject(sqs_message):
    ...
    jinja_template = jinja2.Template(subject)
    subject = jinja_template.render(
        account=sqs_message.get("account", ""),
        account_id=sqs_message.get("account_id", ""),
        partition=sqs_message.get("partition", ""),
        event=sqs_message.get("event", None),
        action=sqs_message["action"],
        policy=sqs_message["policy"],
        region=sqs_message.get("region", ""),
    )
    return subject
```

The get_jinja_env function in `utils.py` of the c7n_mailer tool uses jinja2 library without HTML escaping.

```
def get_jinja_env(template_folders):
    env = jinja2.Environment(trim_blocks=True, autoescape=False)
   ...
    env.loader = jinja2.FileSystemLoader(template_folders)
    return env
```

Mitigation

The main problem of this vulnerability is the missing HTML escaping and validation before using the rendered result. One possible solution is to turn on the automatic HTML escaping when using the jinja2 library but it could create a large overhead if the template is large. The other way is adding logic before the use of the Jinja2 library to escape all HTML-related parameters before passing them to the Jinja2 library to ensure all of them are correctly sanitized before using them for HTML rendering.

Using deprecated and insecure ssl.wrap_socket

Severity	Low
Status	Reported
id	ADA-CC-2023-4
Component	c7n kube

SSL versions 2 and 3 are considered insecure and completely broken. Therefore it is dangerous to use. Because of that, the `ssl.wrap_socket()` function is deprecated as it defaults to an insecure version of SSL/TLS and does not specify a minimum supporting SSL/TLS protocol version. It also does not have support for server name indication (SNI) and hostname matching. An attacker could force it to create an SSL/TLS session with broken SSL protocol versions by a downgrade attack, claiming that only an insecure version of SSL is supported during the handshake process. Besides, attackers can spoof the hostname/server name and cause the later communication and connection vulnerable to attacks. This makes the use of `ssl.wrap_socket()` for creating SSL socket connection insecure and vulnerable to different kinds of attacks on insecure and broken SSL/TLS protocol versions.

Vulnerable code location

The get_finding function in server.py of Cloud Custodian Kubernetes provider uses the deprecate function `ssl.wrap_socket()`.

```
def get_finding(self, resources, existing_finding_id, created_at,
    updated_at):
...
        server.socket = ssl.wrap_socket(
            server.socket,
            server_side=True,
            certfile=cert_path,
            keyfile=cert_key_path,
            ca_certs=ca_cert_path,
        )
```

Mitigation

The deprecated `ssl.wrap_socket()` function should be replaced by the newer `ssl.SSLContext.wrap_socket()` function. This new function returns an SSLContext object, encapsulating the settings and enforcing SNI and hostname matching with default disabling of insecure SSL/TLS version. That could deny possible downgrade attacks or hostname/server name spoofing. If no specific security policies or requirements are needed, `ssl.create_default_context()` function could be used to create a default SSLContext object without the need to specify the security configurations.

Remark

As the new `ssl.SSLContext.wrap_socket()` only supports Stream type socket, thus transferring the use of `ssl.wrap_socket()` may not be a trivial task to do so.

Index out of range in ARN parser

Severity	Informational
Status	Fixed
id	ADA-CC-2023-5
Component	arn parser

Cloud Custodians ARN is susceptible to an index out of range issue from a string split and an assumed array-length. This is a cosmetic issue, since ARNs are never user supplied. Nevertheless, fixing the issue avoids issues in the future in case the use case of the code containing the issue changes. In addition, fixing the issue also allows the fuzzer to reason about the code without getting blocked by a cosmetic bug.

The issue exists on the following lines:

https://github.com/cloud-custodian/cloud-custodian/blob/e20591d4203257652e2de29e237479e81e958ab4/c7n/resources/aws.py#L298-L305

```
298     def parse(cls, arn):
299         if isinstance(arn, Arn):
300             return arn
301         parts = arn.split(':', 5)
302         # a few resources use qualifiers without specifying type
303         if parts[2] in ('s3', 'apigateway', 'execute-api', 'emr-
                serverless'):
304             parts.append(None)
305             parts.append(None)
```

On line 303, the parser reads the third index of the array, however, there can be fewer than three elements in the array.

This issue was found by the ARN parser fuzzer.

Mitigation

We recommend checking that there is a third element in the array before reading it.

Improper URL substring validation can leak data

Severity	Low
Status	Fixed
id	ADA-CC-2023-6
Component	c7n

Cloud Custodians base notifier is susceptible to an improper URL string validation which may allow an attacker to trick Cloud Custodian into leaking data by sending messages to URLs under the attackers control. The issue allows an attacker who can add items to the transport queue to control the URL that the base notifier sends payloads to. Besides the potential for leaking data, an attacker could also manipulate subsequent workflow by crafting a malicious response to Cloud Custodian.

The issue requires high privileges to exploit: Permissions to deploy policies are necessary.

The issue exists in `send_sqs()` of `BaseNotifier`:

https://github.com/cloud-custodian/cloud-custodian/blob/d458f0a24629b5f01160568ef96e728744dc9bbc/c7n/actions/notify.py#L383-L419

```
    def send_sqs(self, message, payload):
        queue = self.data['transport']['queue'].format(**message)
        if queue.startswith('https://queue.amazonaws.com'):
            region = 'us-east-1'
            queue_url = queue
        elif 'queue.amazonaws.com' in queue:
            region = queue[len('https://'):].split('.', 1)[0]
            queue_url = queue
        elif queue.startswith('https://sqs.'):
            region = queue.split('.', 2)[1]
            queue_url = queue
        elif queue.startswith('arn:'):
            queue_arn_split = queue.split(':', 5)
            region = queue_arn_split[3]
            owner_id = queue_arn_split[4]
            queue_name = queue_arn_split[5]
            queue_url = "https://sqs.%s.amazonaws.com/%s/%s" % (
                region, owner_id, queue_name)
        else:
            region = self.manager.config.region
            owner_id = self.manager.config.account_id
            queue_name = queue
```

```
405                 queue_url = "https://sqs.%s.amazonaws.com/%s/%s" % (
406                     region, owner_id, queue_name)
407             client = self.manager.session_factory(
408                 region=region, assume=self.assume_role).client('sqs')
409             attrs = {
410                 'mtype': {
411                     'DataType': 'String',
412                     'StringValue': self.C7N_DATA_MESSAGE,
413                 },
414             }
415             result = client.send_message(
416                 QueueUrl=queue_url,
417                 MessageBody=payload,
418                 MessageAttributes=attrs)
419             return result['MessageId']
```

The issue lies in the two first branches of `send_sqs`. Assuming that an attacker can control the `queue` variable defined on line 384, they can get past the first conditional check on line 385-387 if `queue` does not start with the string `https://queue.amazonaws.com`. The second conditional checks whether `queue.amazonaws.com` is a substring of `queue`. An attacker can bypass that in a number of ways, either by crafting a subdomain of their own domain, for example `queue.amazonaws.com.malicious-url.cc`, or by including a URL parameter, for example: `malicious-url.cc?queue.amazonaws.com`. The conditional check on line 388 will return true and `queue_url` will be assigned the attackers URL. `send_sqs` will proceed to line 407, and the client will sent the message to the attacker-controlled URL on line 416.

Mitigation

Improve URL sanitization of the `queue` value. For example, remove the labeling part of the URL to ensure that only the needed domain is included in `send_sqs`. Splitting the string into domain and path before checking could also help.

Possible DoS from attacker-controller Github account

Severity	Low
Status	Fixed
id	ADA-CC-2023-7
Component	c7n policystream

Cloud Custodians PolicyStream tool is vulnerable to an infinity loop from an attacker-controlled limit in a for loop. The root cause is that the Policy Stream tool loops through all repositories belonging to an organization without setting a limit to the number of iterations. The repositories are remote and an potential attacker has numerous ways to control the list of repositories returned to Cloud Custodian. If they can achieve a position where they control the list of repositories returned to Cloud Custodian, they could cause Cloud Custodian to go into an infinity loop and thereby cause a denial of service.

The root cause of the issue is in the `org_checkout` CLI call. This API loops through all repositories of an organization and applies filtering in each iteration:

https://github.com/cloud-custodian/cloud-custodian/blob/e425311da975e9d03e69f4d33bfeec7ebe65f932/tools/c7n_policystream/policystream.py#L743-L786

```
def org_checkout(organization, github_url, github_token, clone_dir,
                 verbose, filter, exclude):
    """Checkout repositories from a GitHub organization."""
    logging.basicConfig(
        format="%(asctime)s: %(name)s:%(levelname)s %(message)s",
        level=(verbose and logging.DEBUG or logging.INFO))

    callbacks = pygit2.RemoteCallbacks(
        pygit2.UserPass(github_token, 'x-oauth-basic'))

    repos = []
    for r in github_repos(organization, github_url, github_token):
        if filter:
            found = False
            for f in filter:
                if fnmatch(r['name'], f):
                    found = True
                    break
            if not found:
                continue

        if exclude:
            found = False
            for e in exclude:
                if fnmatch(r['name'], e):
                    found = True
                    break
            if found:
                continue

        repo_path = os.path.join(clone_dir, r['name'])
        repos.append(repo_path)
        if not os.path.exists(repo_path):
            log.debug("Cloning repo: %s/%s" % (organization, r['name']))
            repo = pygit2.clone_repository(
                r['url'], repo_path, callbacks=callbacks)
        else:
            repo = pygit2.Repository(repo_path)
            if repo.status():
                log.warning('repo %s not clean skipping update', r['name'])
                continue
            log.debug("Syncing repo: %s/%s" % (organization, r['name']))
            pull(repo, callbacks)
    return repos
```

An attacker could obtain control over the Github organization in a number of ways; For example, the owner of the github organization may not be using proper configuration such as 2FA and permission levels, and an attacker could compromise the Github organization and launch the attack. Alternatively, an attacker pretend to be a legitimate contributor over a longer time, make legitimate contributions, review pull requests etc to build up credibility. The attacker could then choose to launch the attack at a the most lucrative time. The attack could be against a competitor with the goal of achieving a time-advantage in certain operations such as research, business, trading or military operations. These are examples that illustrate the Cloud Custodian does not control the data coming from the remote github repository, whereas fully untrusted users may be able to control it. As such, the data from the remote repositories can be fully untrusted.

The attack is also possible without creating the repositories, if the attacker can intercept communication and control the response returned to Cloud Custodian on these lines: https://github.com/cloud-custodian/cloud-custodian/blob/e425311da975e9d03e69f4d33bfeec7ebe65f932/tools/c7n_policystream/policystream.py#L665C9-L666

In the attacker can control these lines, they can control the JSON that Cloud Custodian uses to create the `repos` variable. Cloud Custodian does not validate whether the repo belongs to the organization when invoking the loop:

https://github.com/cloud-custodian/cloud-custodian/blob/e425311da975e9d03e69f4d33bfeec7ebe65f932/tools/c7n_policystream/policystream.py#L754-L771

```
    for r in github_repos(organization, github_url, github_token):
        if filter:
            found = False
            for f in filter:
                if fnmatch(r['name'], f):
                    found = True
                    break
            if not found:
                continue

        if exclude:
            found = False
            for e in exclude:
                if fnmatch(r['name'], e):
                    found = True
                    break
            if found:
                continue
```

An attacker could therefore return a JSON response pointing to repositories from other Github accounts and achieve the same goal.

Mitigation

We recommend resolving all of the below: 1. Either add a counter to each loop iteration and stop after a `maxAllowed` limit or check the number of repositories before starting the loop. 2. Validate in the beginning of the loop whether each repository belongs to the organization.

Possible DoS from attacker-controller Github repository

Severity	Low
Status	Fixed
id	ADA-CC-2023-8
Component	c7n policystream

Cloud Custodians PolicyStream tool is vulnerable to an infinity loop from an attacker-controlled limit in a loop. The root cause is that the Policy Stream tool loops through all commits in a repository without setting a limit to the number of iterations. The repository is cloned from remote and a potential attacker has numerous ways of controlling it and thereby the number of commits it has. If they can achieve a position where they control the repository and create a high number of commits, they could cause Cloud Custodian to go into an infinity loop and thereby cause a denial of service, since Cloud Custodian would not terminate its operation and would not be able to process subsequent operations.

The root cause of the issue is in the `delta_stream` CLI call. This API loops through all commits of a branch of a repository:

https://github.com/cloud-custodian/cloud-custodian/blob/e425311da975e9d03e69f4d33bfeec7ebe65f932/tools/c7n_policystream/policystream.py#L316-L345

```
    def delta_stream(self, target='HEAD', limit=None,
                     sort=pygit2.GIT_SORT_TIME | pygit2.
                         GIT_SORT_REVERSE,
                     after=None, before=None):
        """Return an iterator of policy changes along a commit lineage
            in a repo.
        """
        if target == 'HEAD':
            target = self.repo.head.target

        commits = []
        for commit in self.repo.walk(target, sort):
            cdate = commit_date(commit)
            log.debug(
                "processing commit id:%s date:%s parents:%d msg:%s",
                str(commit.id)[:6], cdate.isoformat(),
                len(commit.parents), commit.message)
            if after and cdate > after:
                continue
            if before and cdate < before:
                continue
            commits.append(commit)
            if limit and len(commits) > limit:
                break

        if limit:
            self.initialize_tree(commits[limit].tree)
            commits.pop(-1)

        for commit in commits:
            for policy_change in self._process_stream_commit(commit):
                yield policy_change
```

An attacker could obtain control over the Github repository in a number of ways; For example, the owner of the github repository may not be using proper configuration such as 2FA, and an attacker could compromise the owners account and launch the attack. Alternatively, an attacker can pretend to be a legitimate contributor over a longer time, make legitimate contributions, review pull requests etc to build up credibility. The attacker could then choose to launch the attack at a the most lucrative time. The attack could be against a competitor with the goal of achieving a time-advantage in certain operations such as research, business, trading or military operations. These are examples that illustrate the Cloud Custodian does not control the data coming from the remote github repository, whereas fully untrusted users may be able to control it. As such, the data from the remote repositories can be fully untrusted.

`delta_stream` has an option to limit the number of iterations, however, this is off per default.

Mitigation

We recommend resolving all of the below: 1. Change the default `limit` parameter to a number instead of `None`

Possible zip bomb from large S3 object

Severity	Moderate
Status	Reported
id	ADA-FASTIFY-2023-9
Component	c7n

Cloud Custodian is susceptible to a possible Denial-of-Service from a maliciously crafted S3 object. The attack would allow an attacker to exhaust memory of the machine and prevent Cloud Custodian from performing subsequent operations after the vulnerable code part has been invoked. The root cause is that Cloud Custodian decompresses compressesd S3 objects without hardening against large objects. As such, an attacker with permissions to create S3 objects can create a malicious S3 object that will cause Cloud Custodian to exhaust memory when Cloud Custodian resolves it.

The issue exists in two places in the Cloud Custodian URIResolver:

https://github.com/cloud-custodian/cloud-custodian/blob/61ad56cc748e19230cf5794db1dfd3364bc3e66b/c7n/resolver.py#L44-L50

```
    def handle_response_encoding(self, response):
        if response.info().get('Content-Encoding') != 'gzip':
            return response.read().decode('utf-8')

        data = zlib.decompress(response.read(),
                               ZIP_OR_GZIP_HEADER_DETECT).decode('utf8'
                                  )
        return data
```

https://github.com/cloud-custodian/cloud-custodian/blob/61ad56cc748e19230cf5794db1dfd3364bc3e66b/c7n/resolver.py#L52-L68

~~~
```python
    def get_s3_uri(self, uri):
        parsed = urlparse(uri)
        params = dict(
            Bucket=parsed.netloc,
            Key=parsed.path[1:])
        if parsed.query:
            params.update(dict(parse_qsl(parsed.query)))
        region = params.pop('region', None)
        client = self.session_factory().client('s3', region_name=region
           )
~~~

```
        result = client.get_object(**params)
        body = result['Body'].read()
        if params['Key'].lower().endswith(('.gz', '.zip', '.gzip')):
            return zlib.decompress(body, ZIP_OR_GZIP_HEADER_DETECT).
                decode('utf-8')
        elif isinstance(body, str):
            return body
        else:
            return body.decode('utf-8')
```

A denial-of-service scenario can arise purposefully or accidentally; An attacker can specifically craft a malicious S3 object that allocates a lot of memory when Cloud Custodian decompresses it, or the AWS admin can accidentally set no limit to the object, and a large one will accidentally be created during normal business operations and exhaust memory. At the time of this audit, an S3 object can be up to 5TB in size which will be enough to exhaust memory in the majority of use cases.

Mitigation

- Decompress in chunks.

Privilege escalation through chained Lambda functions in AWS

Severity	Moderate
Status	Reported
id	ADA-CC-2023-10
Component	c7n

Cloud Custodian is susceptible to a privilege escalation issue on AWS from an underlying prioritization of privileges in AWS between user privileges and Lambda privileges. This is an issue in AWS that Cloud Custodian inherits. We consider the impact of the issue to be High, however, Cloud Custodian may opinionatedly reject the root cause of this issue to exist in Cloud Custodian. The reason for this is that AWS users are exposed to the same issue.

The issue exists because users in AWS inherit the privileges of the Lambda functions that the users have privileges to invoke, even if the user does not have privileges to carry the actions against the resources that the Lambda functions have privileges to. For example, consider a scenario where a user has privileges to read resources of type “A”, and invoke Lambda function “B”, and Lambda function “B” has privileges to create resources of type “A”. In this case, the user will also have privileges to create resources of type “A”, even if the cluster admin has not excplicitly assigned those privileges.

Below we exmplify the issue with a user that has read-only privileges against EC2 instances, access to the cloud shell, read-only access to Lambda functions and read-only access to IAM. In addition, the user has permissions to invoke Lambda functions. The same example Cloud Custodian deployment has a function called `Invoker()` which has permissions to invoke another Lambda function called `terminate_instance`. `terminate_instance` has privileges to terminate EC2 instances - a privilege that the user does not have.

In this case, the user has permissions to terminate EC2 instances, even though the cluster admin has not assigned these to the user.

Cluster assets and permissions

User Permissions

- AWS managed permissions:

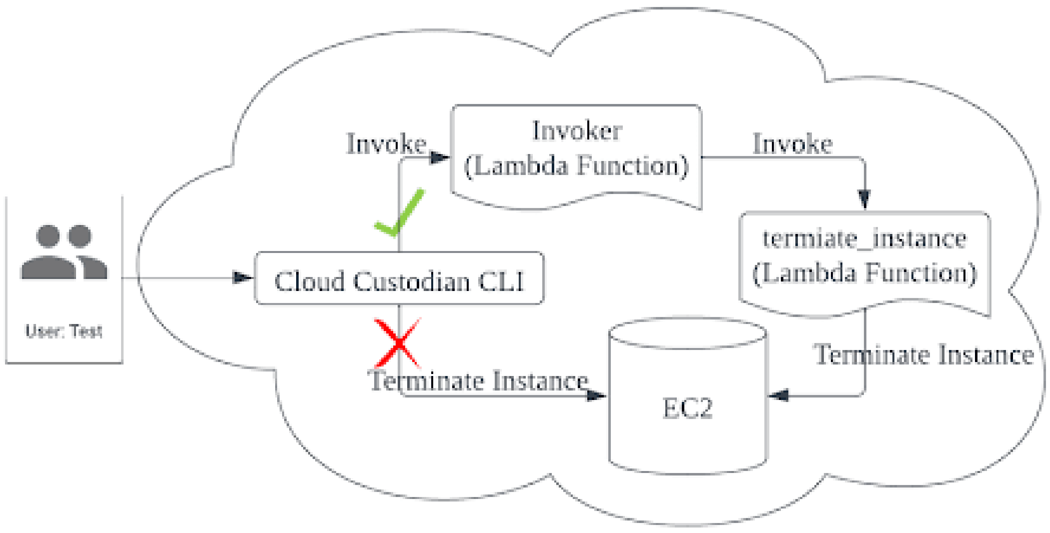


Figure 4: Privilege escalation illustration

- AmazonEC2ReadOnlyAccess
- AWSCloudShellFullAccess
- AWSLambda_ReadOnlyAccess
- IAMReadOnlyAccess

- Custom permissions:
  - lambda:InvokeFunction
  - lambda:InvokeAsync ##### Lambda Function `Invoker` ###### Permissions
- Custom permissions:
  - lambda:InvokeFunction
  - lambda:InvokeAsync ##### Lambda Function "terminate_instance" ###### permissions
- Custom permissions:
  - ec2:terminate_instance

Code for lambda function Invoker If the users attempts to run a Cloud Custodian policy directly to terminate an EC2 instance, they are denied with the following error:

```
botocore.exceptions.ClientError: An error occurred (
    UnauthorizedOperation) when calling the TerminateInstances operation
    : You are not authorised to perform this operation.
```

However, by invoking the chain of the two Lambda functions, the user can still terminate EC2 instances. The user can invoke the Lambda function `Invoker` which invokes `terminate_instance` which terminates an EC2 instance. Consider the following proof concept:

Code for lambda function Invoker

```python
import json
import boto3

client = boto3.client('lambda')

def lambda_handler(event, context):
    response = client.invoke(
        FunctionName='arn:aws:lambda:eu-north-1:[ACCOUNTID]:function:
          terminate_instance',
        InvocationType='RequestResponse',
        Payload=json.dumps(inputForInvoker)
    )

    print(json.load(response['Payload']))
```

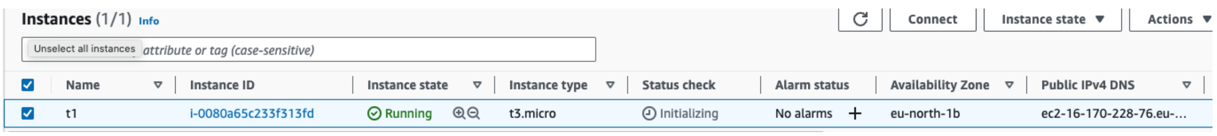
Code for lambda function terminate_instance

```python
import json
import boto3

def lambda_handler(event, context):
    ec2 = boto3.client('ec2', region_name='eu-north-1')
    ec2.terminate_instances(InstanceIds=['[INSTANCEID]'])
```

Proof of concept

Assume there is one EC2 instance in the cloud account. With `tag:Name` = t1 and instance id = `i-0080a65c233f313fd`.



☑	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
☑	t1	i-0080a65c233f313fd	Running	t3.micro	Initializing	No alarms	eu-north-1b	ec2-16-170-228-76.eu-...

Figure 5: Instance List

In this situation, the user "Test" has a limited set of permissions as stated above. He/she cannot execute a Cloud Custodian policy directly to EC2 in order to terminate an instance, because he/she only has read-only access to the EC2 resources. The following Cloud Custodian policy will fail.

```
policies:
  - name: ec2-delete-marked
    resource: ec2
    filters:
      - "tag:Name": "t1"
    actions:
      - type: terminate
        force: true
```

The error message from the above Cloud Custodian policy execution is as follows.

```
botocore.exceptions.ClientError: An error occurred (
    UnauthorizedOperation) when calling the TerminateInstances operation
    : You are not authorized to perform this operation.
```

However it is found that the user Test could still be able to terminate an instance with a detour path. The user Test can run a Cloud Custodian policy that invokes a lambda function "Invoker", where "Invoker" has permission and logic to invoke another lambda function "terminate_instance". If "terminate_instance" does have the EC2 instance termination permission. The policy to invoke "Invoker" could end up terminating an instance in EC2, given that neither the user "Test" nor the lambda function "Invoker" has the EC2 instance termination right. Cloud Custodian did not mitigate this kind of attack. The following is a policy for invoking the lambda function "Invoker"

Cloud custodian policy that user Test could deploy and terminate an instance

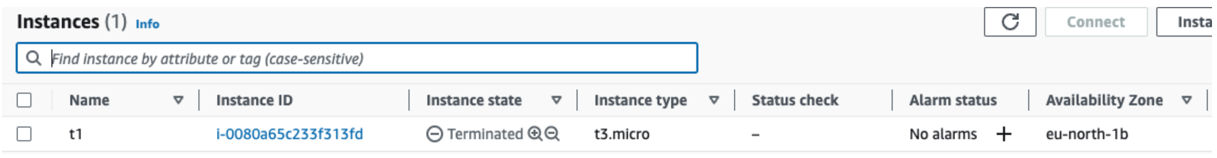
```
policies:
  - name: invoke
    resource: ec2
    filters:
      - "tag:Name": "t1"
    actions:
    - type: invoke-lambda
      function: Invoker
```

The following result shows that the policies has run successfully.

```
2023-10-09 18:08:22,470: custodian.commands:DEBUG Loaded file invoke.
    yml. Contains 1 policies
2023-10-09 18:08:22,486: custodian.aws:DEBUG using default region:eu-
    north-1 from boto
2023-10-09 18:08:23,089: custodian.output:DEBUG Storing output with <
    LogFile file://./invoke/custodian-run.log>
2023-10-09 18:08:23,100: custodian.policy:DEBUG Running policy:invoke
    resource:ec2 region:eu-north-1 c7n:0.9.29
2023-10-09 18:08:23,100: custodian.cache:DEBUG expiring stale cache
    entries
2023-10-09 18:08:23,101: custodian.resources.ec2:DEBUG Using cached c7n
    .resources.ec2.EC2: 1
```

```
2023-10-09 18:08:23,101: custodian.resources.ec2:DEBUG Filtered from 1
    to 1 ec2
2023-10-09 18:08:23,101: custodian.policy:INFO policy:invoke resource:
    ec2 region:eu-north-1 count:1 time:0.00
2023-10-09 18:08:23,706: custodian.policy:INFO policy:invoke action:
    lambdainvoke resources:1 execution_time:0.60
2023-10-09 18:08:23,708: custodian.output:DEBUG metric:ResourceCount
    Count:1 policy:invoke restype:ec2 scope:policy
2023-10-09 18:08:23,708: custodian.output:DEBUG metric:ApiCalls Count:2
     policy:invoke restype:ec2
```

The instance is being terminated.



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
t1	i-0080a65c233f313fd	Terminated	t3.micro	–	No alarms	eu-north-1b

Figure 6: Terminated Instance

Impact

All Cloud Custodian can be impacted by this, but will not be by default when using Cloud Custodian. Users need will need to configure their Cloud Custodian deployment in a similar manner exemplified above. Users will not easily accidentally configure their deployments in such a manner, however, the more complex the deployment regardingnumber of users users, permissions and Lambda functions, the more likely a mistake can happen that allows for this privilege escalation. As such, users must enable a configuration that allows this privilege escalation.

The issue can result in escalation to the highest level of privileges that the cluster admin has enabled, however, this is highly dependent on a particular Cloud Custodian deployment. As such, users should consider this on a case-by-case basis.