Detecting Multi-Step IAM Attacks in AWS Environments via Model Checking

Ilia Shevrin
Citi

Oded Margalit
Ben-Gurion University

Abstract

Cloud services enjoy a surging popularity among IT professionals, owing to their rapid provision of virtual infrastructure on demand. Hand-in-hand with the growing usage, there is also a growing concern about potential security vulnerabilities arising from misconfigurations, exposing resources or allowing malicious actors to escalate privileges. Model checking is a known method for verifying that a finite-state Boolean model of a system satisfies certain properties, where the model and the properties are described in formal logic. In case it doesn't, a finite trace leading to a violating state can be generated.

In this paper, we present an approach to construct a finite-state Boolean model from the Identity and Access Management (IAM) component of Amazon Web Services (AWS), and a property from an attack target, e.g., read a classified S3 bucket object. We run a model checker that detects whether some initial setup allows an attacker to escalate privileges and reach the target in one or more steps by applying IAM manipulating actions. We show that our approach can discover existing misconfigurations in real AWS environments, and that it can detect multi-step attacks in setups containing tens of AWS accounts with hundreds of resources in under a minute.

1 Introduction

Cloud services have risen in popularity in the past years thanks to their ability to effectively supply the customer with virtual infrastructure on demand. The rapid adoption inevitably led to concerns about the security posture of applications deployed on remote, provisioned hardware. AWS devised the Shared Responsibility Model [9], where security and compliance is a shared responsibility between the provider and the customer. The cloud provider is responsible for the **Security of the Cloud** – protecting the infrastructure that runs all of the services offered in the Cloud. The customer is responsible for the **Security in the Cloud** – managing and encrypting the data, classifying the assets, and using IAM tools to apply permissions.

Hence, the burden of configuring the IAM permissions for the application is left to the customer, and along with an increase in the scope and complexity of cloud-deployed systems, it should come as no surprise that according to recent surveys [42], cloud service misconfigurations and privileged user abuse are involved in 42.2 and 37.8 percent respectively of security attacks. In their words, misconfiguration of cloud resources is a pervasive issue, as evidenced by the plethora of exposed S3 (Simple Storage Service) buckets. Privileged user abuse is likely symptomatic of the complexity of IAM policies and settings that are tied to most cloud operations.

Furthermore, OWASP released in 2021 the top 10 web application security risks document [2]. At least 3 out of 10 items could be connected to the IAM configuration vulnerabilities problem. They are **identification and authentication problems**, which leads to initial cloud principal compromise; **broken access control**, which directly relates to flaws in policies leading to unwanted escalation of privileges by attackers; and **security misconfiguration**, which can serve as an umbrella term for imperfect policies containing errors and typos that have security implications. Additionally, several techniques of the influential MITRE ATT&CK framework [5] directly address this topic – Exploitation for Privilege Escalation (T1068); Domain Policy Modification (T1484); Abuse Elevation Control Mechanism (T1548).

Recently, Rhino Security Labs researchers showed that certain combinations of permissions in AWS IAM policies can be exploited in privilege escalation methods, by performing certain AWS actions under certain conditions [27]. To detect these combinations, instead of performing syntactic scans on the policy documents, we purpose a model checking approach. We construct a finite-state Boolean model, which is essentially a transition system where the AWS actions are assigned semantics from an adversarial point of view. This model is then formally checked that it satisfies certain reachability properties. These properties are derived from attack targets such as "read a classified S3 bucket object", or "get full administrator permissions". If the model checker finds a trace that satisfies the property, it produces a list of steps that describe the AWS actions an attacker must perform until they reach the target. Alternatively, the model checker may output that the setup is safe if it manages to prove that no traces exist.

Constructing such a model poses many challenges due to the richness of the AWS IAM service, and due to the fact that real-world cloud configurations are usually expected to contain hundreds of cloud resources. Hence, the modeling is essentially an exercise in careful abstraction of complicated real-world frameworks. In this paper, we attempt to provide motivation for deciding when to abstract certain elements of AWS while keeping others. For example, an IAM action that updates the inline policy of a role (**PutRolePolicy**) is crucial to model because it may serve as a means for a privilege escalation, but the JSON content of the updated policy itself can be abstracted since we assume the attacker provides the most permissive content suited for their needs (e.g., allow all actions on all resources).

Moreover, the modeling process is not a one-time effort, as at any time there may be IAM exploits that are not formalized in the model, or that are not even discovered yet by security researchers. The model checker will not be able to produce traces with these exploits. However, we envision that with time our model will cover more and more exploits, enhancing its credibility and reducing false positives and false negatives. In the meantime, our model already covers many known exploits, and our evaluation shows that our approach is able to detect existing misconfigurations in pre-deployment environments and assist security engineers in verifying their IAM policies.

Another challenge is the scalability of the model checking process. While we greatly depend on the quality of our underlying formal reasoning tools, we also provide a technique to alleviate the performance burden on the model checker. We can construct the model efficiently using only Boolean variables while leaving the original model checking interface intact. Our evaluation shows that the model checker can detect attack vectors of up to 5 steps on accounts with hundreds of resources in under a minute.

In brief, our contributions in the paper are as follows.

- We define a Boolean model of the AWS IAM component and discuss ways to model and abstract various IAM elements while maintaining correctness in the context of our threat model. (Section 5).
- We define and implement a model checking process that receives an AWS organization setup and a target property to check, and detects multi-step attacks exploiting AWS actions (Sections 6 and 7). To the best of our knowledge, this is the first work that attempts to apply model checking on the IAM component in cloud environments to detect multi-step privilege escalation attacks.
- We evaluate the scalability of our approach and the quality of the detected results (Section 8).
- We present a technique to improve model checking performance by translating non-Boolean variables (such as strings) into Boolean arrays, thus essentially reducing the problem to pure Boolean satisfiability (Appendix A).

2 Model Checking

Model checking [22] is a known method for checking whether a finite-state model of a system meets a given specification. The model and the specification (or a property), are formulated in a precise formal language, mostly using mathematical logic. A popular class of properties is safety properties, which guarantee that a bad thing *never* happens in the system.

Model checking strength lies in the premise that it explores the full state space of the model, giving it two advantages: it is capable of detecting rare bugs, by generating a trace, or a counter-example, leading to a violation of the validated property; and it can generate a proof assuring that the model satisfies the property. The exhaustive nature of model checking often leads to the state-space explosion problem as the modeled systems grow. Attempts to deal with this problem include algorithms such as Bounded Model Checking (BMC) [43] using k-induction, and IC3 [19, 25].

Many model checking techniques usually integrate satisfiability (SAT) solvers [35] in their algorithms. A SAT solver determines whether there exists an assignment to a given Boolean formula, such that the formula evaluates to true. A satisfiability modulo-theorem (SMT) solver receives a richer set of inputs, that is, formulas with respect to theories such as real arithmetic, arrays, and strings. Fast and efficient SAT/SMT solvers exist [17, 23], and are applied in areas such as program verification, static analysis, and model checking [24].

Specifically, a BMC algorithm constructs in each loop iteration k a Boolean formula unfolding the model up to step k, and then constrains it with the negation of the property in step k. The satisfiability of this formula is then checked in a SAT solver. A satisfying assignment is directly translated to a trace in step k, while an unsatisfied result signals the algorithm to proceed to the next step. A k-induction algorithm is interwoven in the loop, that is, if the algorithm proves the statement that "the last k steps are safe" implies that "step $k+1$ is safe", it may stop and conclude global safety.

3 AWS Identity and Access Management

In the context of AWS IAM, a *statement* is a permission defined by the following elements (as of 2022 [4]):

1. **Effect** is either allow or deny.
2. **Action / NotAction** defines what action(s) can / cannot be performed.
3. **Resource / NotResource** defines on what resource(s)

the action can / cannot be performed. A resource is a cloud object with a unique identifier, such as S3 bucket, IAM role, or Lambda function.

4. **Principal / NotPrincipal** defines by what principal(s) the action can / cannot be performed. Various AWS services, as well as resources such as IAM roles and IAM users may serve as principals.
5. **Condition** defines additional conditions depending on the action. A condition is composed of an operator, key and value.

The values in each element are usually represented as Amazon Resource Names (ARN). The ARN is most commonly defined by the following format.

```
arn:partition:service:region:account-id:resource-
    type/resource-id
```

For example, `arn:aws:iam::1234:role/MyRole` is the ARN of an IAM role named `MyRole` belonging to account 1234. Some of the ARN's elements can be omitted for specific resources, for instance IAM resource ARNs do not have regions. The name identifier in an ARN may contain wildcard characters – Asterisk "`*`" is a placeholder for zero or more characters; question mark "`?`" is a placeholder for a single character.

Conditions are composed of a condition operator, for example `ArnEquals` or `StringNotLike`; a condition key, for example `policyARN` or `instanceType`; and a condition value which may be of any type depending on the key. In case of `policyARN`, the value is formatted as an ARN and the accompanying operator works on ARN values; in case of `instanceType` the value is a string and the accompanying operator works on string values. To simplify presentation, we omit the service prefix that preceeds action and condition names in AWS.

For example, a statement JSON may contain the following content.

```
{"Effect": "Deny",
 "Action": ["AttachRolePolicy","DetachRolePolicy
     "],
 "Resource": "arn:aws:iam::1234:role/MyRole",
 "Condition": { "ArnNotLike": {
     "PolicyARN": "arn:aws:iam::1234:policy/Dev*"
          }}}
```

This statement denies attaching and detaching IAM policies whose name is *not* matched by the regular expression `Dev*`, to the IAM role `MyRole` in account 1234. It does not state anything about the policies that are *allowed* to be attached.

A *policy* is an array of statements. Policies may belong to resources, defining *what can be performed on the resources by which principals* (resource-based policies); or to principals, defining *what the principals can perform on other resources* (identity-based policies). For the latter, policies may be defined inline for a single principal, or as IAM policy resources (AWS managed or custom), which can be attached to users, groups, or roles. Policies may be also evaluated on a global organization scope and not for a single resource (service control policies, attached to accounts). Policies may also serve as permissions boundaries, limiting effective permissions of a principal to a conjunction of their permissions boundary with their attached policies. The fundamental policy evaluation rule is that a permission requires at least one allow statement across all relevant policies and no deny statement.

Resources in AWS belong to accounts, which in turn belong to organizational units, and together form the organization hierarchy. When evaluating permissions, AWS takes into account many factors: the principals' identity-based policies, the principals' permissions boundary, the service control policies attached to the account and the organization units in the organization tree, and the resource-based policies attached to the relevant resources. Then a final allow/deny outcome is produced according to the elaborate policy evaluation logic described in the AWS documentation [3, 6]. We briefly summarize it as follows.

1. **Deny evaluation**. An explicit deny statement in *any* IAM policy that applies to the request results in a final decision of Deny.
2. **Organizations SCPs**. Absence of an allow statement in global AWS Organizations service control policies (implicit deny) results in a final decision of Deny. Otherwise, the evaluation continues.
3. **Resource-based policies**. Within the same account, the presence of an allow statement in a resource-based policy results in a final decision of Allow. For cross-account requests, absence of an allow statement in a resource-based policy results in a Deny.
4. **Identity-based policies**. The identity-based policies are evaluated and the absence of an allow statement results in a Deny.
5. **IAM permissions boundaries**. The permissions boundary is evaluated only if such exists. Absence of an allow statement results in a Deny.
6. **Session policies**. Session policies are evaluated if the principal is a session principal and such a policy exists. Absence of an allow statement results in a Deny. Otherwise, the final decision is Allow.

An IAM role resource type has both an identity-based policy and a resource-based policy. The latter is called a trust policy and it defines which principals can *assume* this role, i.e., obtain temporal credentials allowing them to operate on behalf of this role.

4 A Motivating Example

Consider an AWS account that contains the following two resources. To simplify the presentation we use names instead of full ARNs.

dept2/Admin. An IAM role with an inline policy document as follows.

```
"Effect": "allow",
"Action": "*Role*",
"Resource": "dept2/*"

"Effect": "deny",
"Action": "*Role*",
"Resource": "dept2/Admin"
```

This policy grants permission to perform a plethora of role-related IAM actions (such as creation of new roles and changing roles' policies), but only on roles with the name pattern defined by the regular expression `dept2/*`. As a precaution, the policy denies explicitly these actions on `dept2/Admin` themselves.

classified. An S3 bucket with a resource policy document as follows.

```
"Effect": "allow",
"Action": "GetObject",
"Resource": "classified/*",
"Principal": "dept1/*"

"Effect": "deny",
"Action": "GetObject",
"Resource": "classified/*",
"NotPrincipal": "dept1/*"
```

This policy grants read access to objects in the bucket only to roles with the name pattern defined by the regular expression `dept1/*` and denies access to the rest of the organization.

At a first glance, only `dept1/*` roles can access the classified bucket. However, in case `dept2/Admin`'s credentials are compromised by a malicious actor that is at least partially aware of the existing permissions in the account, a chain of actions may lead an unauthorized principal to gradually elevate privileges until they can read data from a classified bucket. The loophole lies in the fact that while `dept2/Admin` cannot alter their own permissions, they can create new roles and modify *their* permissions to achieve the desired results.

The 5-step chain of actions for the attacker is as follows.

1. Perform **CreateRole** in order to create a new role whose name adhers to `dept2/*` pattern.
2. Perform **PutRolePolicy** in order to give the new role permissions that allow it to perform S3 service actions, and in particular bucket policy modifying actions.
3. Perform **AssumeRole** in order to get temporary session credentials for the newly created role.

   At this stage, the attacker escalated their privileges to the administrator level, because they can authenticate with the new role. However, they cannot read from the classified bucket yet because the regular expression in the policy doesn't match the name.
4. Perform **DeleteBucketPolicy** to delete the restricting bucket policy of `classified` (authenticating with the credentials of the new role).
5. Perform **GetObject** and read classified objects in the target bucket.

Observe that the compromised principal is not granted full access privileges, and there is no usage of the star `"*"` wildcard. Of course, `dept2/Admin` defined permissions are sensitive as they contain IAM actions, but it might be not enough for a syntactic policy scanning tool to alert on. Instead, the attack succeeded due to a combination of actions and naming patterns that can be easily left unnoticed when examined in isolation. Assessing the regular expressions found across all policies, taking into account complex evaluation rules, requires a deeper analysis effort than simply scanning for a set of pre-defined best-practice dangerous patterns.

5 AWS IAM Model

We present a definition of a finite-state Boolean model of the AWS IAM component. The model and an attack target property serve as an input to a model checker, which outputs a finite trace if such exists, i.e., a sequence of steps as performed by an attacker leading to the target.

The model itself is a Boolean formula over operators $=$, $\wedge$, $\vee$, $\rightarrow$, and $\neg$ with their respective semantics. We wish to keep the model confined exclusively to Boolean logic, so we encode variables of data types such as enumerations, lists, and strings as arrays of Boolean variables. An enumeration is a one-hot encoding Boolean array where each variable represents equivalence to a single value. A list is a Boolean array where each variable represents the containment of a specific element in the list. A string is a Boolean array where each variable represents a regular expression match. The corresponding algorithms for translating strings to Boolean variables are provided in detail in Appendix A. We remark that this encoding dramatically improves model checking performance as it alleviates the SAT solver from the computational burden of formally reasoning about other data types, see evaluation in Section 8.

The model is logically divided into four main components, the organization state, the attacker state, the policy evaluation logic and the action semantics. Roughly speaking, the attacker executes actions that change the organization or the attacker states and affect the policy evaluation logic outcome, thus allowing them to execute other actions and get closer to their target.

The model is constructed from external sources such as the AWS documentation and security research publications, and from the initial AWS organization setup. From the external sources we define the policy evaluation logic and the semantics of the actions that the attacker can use. This information is unlikely to change often and is outside of our control. From the initial setup, we gather all the information about the cloud resources and build the organization and attacker states. This information is subject to relatively frequent changes. This conceptual division can be illustrated through a puzzle such as Sudoku or Sokoban – the description of the puzzle rules versus the puzzle setup. One may regard two different setups as producing two distinct models, but in our context, we refer to the whole construction as a single model.

5.1 Threat Model

We make the following assumptions about the attacker and the cloud environment:

- The attacker begins with at least one compromised set of credentials of an AWS principal, obtained by any means. It can be either via phishing, by exploiting a security flaw in a cloud application, or by having internal access in advance.
- The attacker can perform AWS actions via any interface, for example, the AWS web console, command line, or AWS SDK – the means of execution is transparent to the model. Also, the attacker may perform actions within any time interval, i.e., the attack may be scripted and executed automatically within seconds, or manually in the course of hours or days.
- The attacker is aware of what resources exist in the organization and their properties. For example, an attacker may know beforehand that a permissive IAM policy exists in the account, so that they can attach it to themselves and escalate privileges. Without the attacker's discovery of the available resources, the likelihood of executing an attack is perhaps reduced but still not eliminated, given that the vulnerable permissions are nonetheless there.
- The attacker is not affected by configurations outside IAM evaluation, such as the network layer. Specifically, when deciding whether an attacker can perform an action, the model does not consider elements such as firewall or router rules.
- Given the chance to update an IAM policy document (via **PutRolePolicy** for example), an attacker always chooses the most permissive contents, to escalate privileges as much as possible.
- Besides having an initial set of credentials, the attacker can acquire different credentials of principals in the organization by executing actions such as **AssumeRole**. These credentials are not expired during the attack (even though they may have an expiration date, we assume the attack is finished by that time).

Some authors [11] define *monotonicity* as the property of a model such that no action an attacker takes interferes with the attacker's ability to take any other action. Generally, we cannot assume monotonicity in the context of AWS IAM as it is possible to attach a policy with both allow and deny statements, thus forbidding the principal to perform something that they could on a previous step, but allowing something else. Observe that lack of monotonicity increases the problem complexity, as in the course of an attack, the attacker may first reduce their permissions in order to gain more later.

5.2 Modeled AWS Elements

AWS is known to offer a very rich set of services, many of which contain hundreds of actions and resource types. AWS IAM service in particular supports over one hundred different actions. We focus our modeling efforts on a subset of all AWS elements that can be regarded as *means*. *Means* actions are used by the attacker to change the organization or the attacker states in the next step via a defined transition relation, advancing them toward the target. Such actions meet one of two criteria: (1) directly affect policy evaluation logic; (2) are exploited in known IAM privilege escalation methods. The rest of the AWS actions are regarded as *ends*, i.e., potential targets for the attacker. In particular, all AWS read actions are not *means*, as we assume the attacker already knows what resources exist in the organization.

To cover actions that meet (1), we formalize in the model all the IAM policy manipulating actions, including Organizations actions that affect SCPs, and actions that affect resource-based policies such as **PutBucketPolicy** or **DeleteBucketPolicy**. The model must also recognize crucial IAM resource types, such as users, roles and policies, AWS Organizations resource types such as accounts and SCPs, and policy condition keys such as `policyARN`.

To cover actions that meet (2), we formalize in the model all the actions considered to be vulnerable by security researchers, even though they may not change the IAM state directly, e.g., **InvokeLambda** or **RunInstances**. To this end, we formalized all the AWS actions semantics from the IAM privilege escalation methods described in the Rhino Security Labs article [27]. The model must also recognize the associated resource types such as Lambda functions and EC2 instances. In Section 5.6 we describe the additional formalization that is required to support these methods.

In Appendix C we present the full list of the modeled AWS actions and their semantics. Actions and resource types are encoded as one-hot-encoding Boolean arrays.

5.3 Organization and Attacker States

The organization state formula encodes the current state of the organization resources. All the resources are divided into accounts and are identified via constants of type, name, and the account they belong to. For resources of specific types, we encode additional information that is relevant to the policy evaluation logic. For example, for IAM policies we encode their policy document state (see Section 5.4.2); for IAM roles, users, and groups we encode their inline policy document state, and their attached IAM policies lists; for IAM roles and users we encode their permissions boundary policies. For S3 buckets we encode their encrypting KMS (Key Management

System) key if such exist. For IAM roles we encode their trust policy document state. For IAM users and groups we also encode their names as Boolean arrays because these resource types can have their names modified after creation via an update action, hence they are not constant in the model. For accounts as Organizations service objects we also store their attached SCPs and the organizational unit hierarchy.

We also take into account resources that are created during the attack (recall example in Section 4). Hence, for every resource type, the organization state contains extra variables to accommodate a fixed number of such to-be-created resources. Each of these to-be-created resources has a corresponding Boolean flag that is turned on following execution of a create action in the previous step (for instance, **CreateRole** turns on the flag for an IAM role).

The attacker state formula encodes two elements. First, the request performed by the attacker in the current step – a combination of variables representing an AWS action, the resource on which the action is performed, the principal who performs the action, and additional parameters. Second, a list (of fixed-size, but configurable length) of compromised principals whose credentials (passwords for users, or access and secret keys for roles) were acquired during the attack, for example after performing **AssumeRole** and reading the role's credentials. When the attacker executes requests, they must be authenticated as a principal from this list.

Additional variables represent additional request parameters that are utilized when needed. For example, **AttachRolePolicy** requires the IAM policy to be attached to the role as a parameter. In practice, AWS actions usually support a long list of parameters in the request, but the majority of them does not affect the outcome of potential attacks, hence they are omitted. For example, in the actual API, **AssumeRole** action requires passing over a `DurationSeconds` parameter which limits the duration in seconds of the role session. Since the concept of time is absent from our model, we may simply ignore this parameter.

5.4 Policy Evaluation Logic

The policy evaluation logic determines under what conditions to allow or deny a given request. This logic is extracted from the AWS documentation flowchart [6] as the source of truth. We model this flowchart as a formula that includes all the encoded policy documents from all the resources across the organization. We proceed by describing how a single JSON policy document is encoded as a Boolean formula.

5.4.1 Policy Document

Each element in a policy document statement is mapped to a set of assignments to request variables. Effect field divides the statements into two lists, one for allow and one for deny. Negated elements in the statement wrap the formula in a logical negation. An array of values for an element is mapped to a disjunction between the formulas.

Some elements in the policy documents are not relevant to the model and are treated as follows. Unrecognized actions and resource types are simply skipped, while unrecognized condition operators and keys are evaluated based on the effect. If the effect is allow, it is assumed that the condition holds and does not interfere with the evaluation. If the effect is deny, it is assumed the condition *does not* hold, so the deny statement is not taken into consideration. Naturally, this may impact the false positives rate, but such false positives can still be seen as valid hypothetical attacks under different extra-IAM conditions.

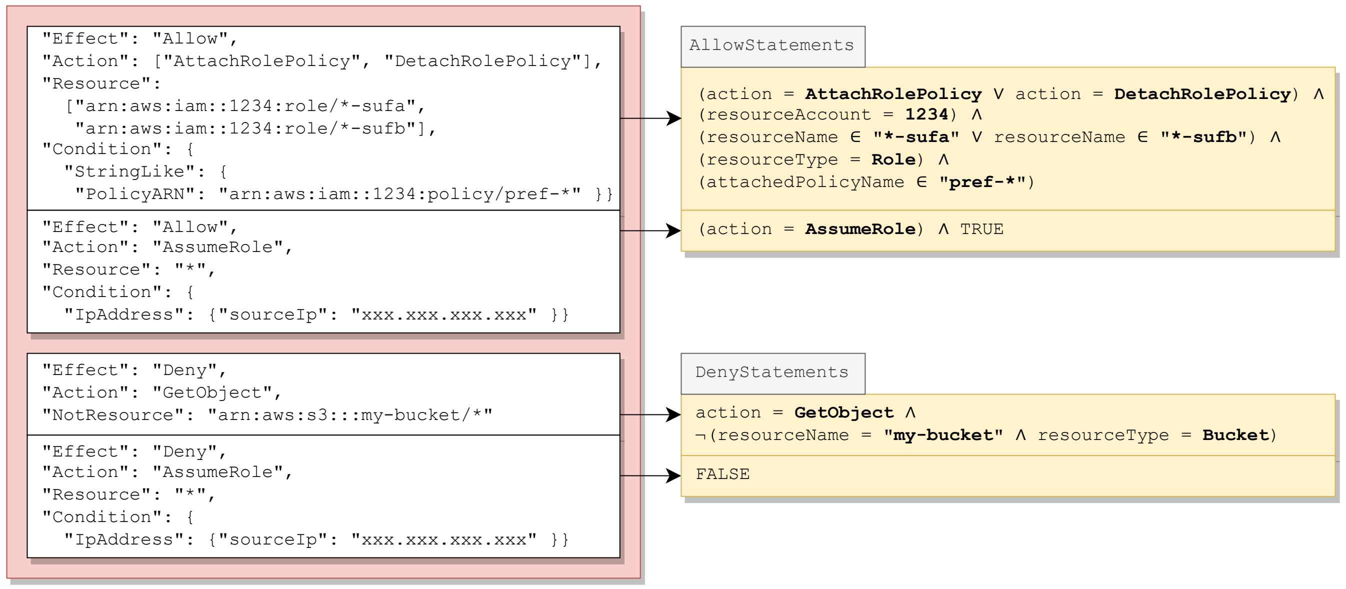
Finally, all the elements in the statement are brought together into a single conjunction. All these conjunctions in both allow and deny lists are brought together into two separate disjunctions. In Section 5.4.3 we see how these two formulas are applied in unison to form the full evaluation logic formula.

Consider the example policy and its corresponding set of constraints in Figure 1. The formula `AllowStatements` represents the permission to attach and detach policies with the prefix `pref-` from roles from account `1234`, with the suffixes being either `-sufa` or `-sufb`. Another allow statement contains a condition on the request IP range. Given that the network layer is not modeled, we decide how to deal with this statement according to the effect. We assume that the request originates within the given source IP range and evaluate the condition to true. The formula `DenyStatements` represents the restriction to get objects from all buckets except `my-bucket`. Another deny statement checks IP range, but here we assume the request originates outside the IP range, and evaluate the condition to false, which in fact renders the whole statement void.

With such encoding, the model checker may produce attack vectors exploiting the permission to perform **AssumeRole** from the allowed IP range, while in reality it may be the case that all the principals are expected to operate from a different IP range, thus rendering the attack vectors incorrect. However, such findings still hold value, if we are to assume a change in the network configuration that takes place independently of the IAM policies and inadvertently makes the attacks realizable.

5.4.2 Policy State

IAM Policy documents can be modified during an attack – an attacker may add or remove various statements in a policy to gain additional privileges. We abstract the policy document into three states: original, updated, and deleted. All policies are initially in their original state. Certain actions transform the policy to the updated state, e.g., **PutRolePolicy** for role inline policies, or **CreatePolicyVersion** for managed policies. We assume that the new updated policy is as permissive as



```
"Effect": "Allow",
"Action": ["AttachRolePolicy", "DetachRolePolicy"],
"Resource":
  ["arn:aws:iam::1234:role/*-sufa",
   "arn:aws:iam::1234:role/*-sufb"],
"Condition": {
  "StringLike": {
   "PolicyARN": "arn:aws:iam::1234:policy/pref-*" }}
```

```
"Effect": "Allow",
"Action": "AssumeRole",
"Resource": "*",
"Condition": {
  "IpAddress": {"sourceIp": "xxx.xxx.xxx.xxx" }}
```

```
"Effect": "Deny",
"Action": "GetObject",
"NotResource": "arn:aws:s3:::my-bucket/*"
```

```
"Effect": "Deny",
"Action": "AssumeRole",
"Resource": "*",
"Condition": {
  "IpAddress": {"sourceIp": "xxx.xxx.xxx.xxx" }}
```

```
AllowStatements

(action = AttachRolePolicy ∨ action = DetachRolePolicy) ∧
(resourceAccount = 1234) ∧
(resourceName ∈ "*-sufa" ∨ resourceName ∈ "*-sufb") ∧
(resourceType = Role) ∧
(attachedPolicyName ∈ "pref-*")

(action = AssumeRole) ∧ TRUE
```

```
DenyStatements

action = GetObject ∧
¬(resourceName = "my-bucket" ∧ resourceType = Bucket)

FALSE
```

Figure 1: An example of an IAM policy document (left) and its translation into a corresponding formula (right)

the attacker desires. Similarly, Certain actions transform the policy to a deleted state, e.g., **DeleteRolePolicy** for role inline policies. A deleted policy has neither its allow statements nor its deny statements evaluated.

For each policy, we represent these states with Boolean variables `updated` and `deleted`. In order to allow a request, a policy must not be deleted, and either be updated or allow the request via one of its statements. In order to deny a request, a policy must not be deleted, not be updated, and also deny the request via one of its statements. Allowing (or denying) a request via one of its statements means that `AllowStatements` (or `DenyStatements` respectively) formula evaluates to true when its variables are substituted with the request assignments.

5.4.3 Policy Evaluation Flowchart

The formula for the policy evaluation logic is constructed according to the flowcharts published by AWS [3, 6]. This formula is ultimately composed of all the sub-formulas of all the resource and identity policy documents across the organization.

We define the formula for the policy evaluation logic for same-account requests and for cross-account requests as presented in Figure 2. In the figure, AWS policy evaluation flowchart is translated into a formula with parts corresponding to each evaluation stage. Each part contains the formulas of the relevant IAM policy documents. For example, `IdentityBasedPoliciesAllow` formula contains the allow sub-formulas for IAM policies and user/role/group inline policies. Service control policies are additionaly evaluated according to the inheritance rules described in [8].

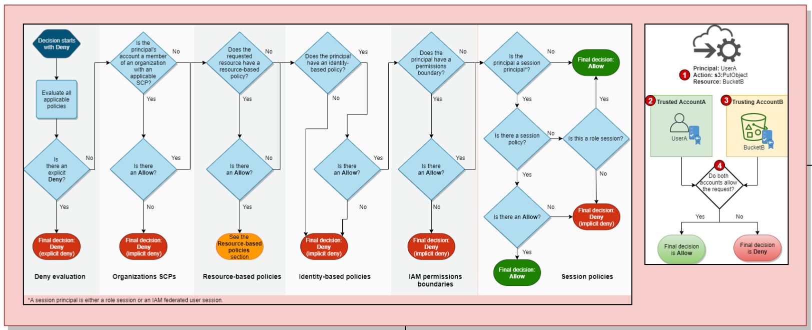
For cross-account evaluation, *both* the resource-based policy and the identity-based policy must allow the request, instead of *at least* one of them [3]. `resourceAccount = principalAccount` formalizes a request within the same account. When true, the policy evaluation formula is reduced to the single account flowchart from [6]. Otherwise, an extra condition is formalized as well.

Observe that from an adversarial point of view, it is not beneficial to manually add session policies when assuming a role, because they do not grant more permissions, but only limit existing permissions by triggering an additional condition in the flowchart. Therefore we may discard the handling of session policies in our model, assuming that an adversary always has a more efficient attack without considering this choice.

5.5 Action Semantics

The requests executed by the attacker directly affect the organization state. We define for each AWS action that serves as an attack *means* unique semantics that describes the transition relation, i.e., the result of the execution in the next step. Each action that we model was carefully studied in the context of the security model, to decide upon its effect on the organization. We base our semantics on hands-on experimentation in real AWS environments and AWS documentation.

In Figure 3 we present an example of an action and its semantics – **PutRolePolicy**, which transfers the inline policy of a given IAM role to an updated state. In the formula, the `updated` variable for the role which is the resource of the current request is turned on in the next step. Also, we add constraints that keep this variable unchanged between steps for all other roles in all accounts, as obviously other resources



```
SameAcc :=
  resourceAccount = principalAccount

PolicyEvaluationLogic :=
  ServiceControlPoliciesAllow ∧
  ((ResourceBasedPoliciesAllow ∧ SameAcc) ∨
   (IdentityBasedPoliciesAllow ∧
    PermissionsBoundariesAllow)) ∧
  (ResourceBasedPoliciesAllow ∨ SameAcc) ∧
  ¬ServiceControlPoliciesDeny ∧
  ¬ResourceBasedPoliciesDeny ∧
  ¬PermissionsBoundariesDeny ∧
  ¬IdentityBasedPoliciesDeny
```

```
Allow(policy) := ¬policy.deleted ∧ (policy.updated ∨ policy.AllowStatements)

AllowForResource(account, resource) :=
  ∃policy ∈ account.IAMPolicies: (policy ∈ resource.AttachedPolicies ∧ Allow(policy)) ∨ Allow(resource.InlinePolicy)

IdentityBasedPoliciesAllow :=
  ∃account ∈ Accounts: principalAccount = account.id ∧
    (∃role ∈ account.IAMRoles: principalName = role.name ∧ AllowForResource(account, role)) ∨
    (∃user ∈ account.IAMUsers: principalName = user.name ∧ (AllowForResource(account, user) ∨
     ∃group ∈ account.IAMGroups: (group ∈ user.GroupsUserBelongsTo ∧ AllowForResource(account, group))))
```

Figure 2: The policy evaluation logic flowcharts, taken from AWS 2022 documentation [3, 6], are translated into a formula that encompasses all IAM policy documents in the organization. In this figure we also show the subformula for `IdentityBasedPoliciesAllow`

should not be affected by this request.

Other action semantics are defined in a similar fashion – according to their effect on the organization from an adversarial point of view. **AttachRolePolicy** modifies the attached policies list for an IAM role; **UpdateAssumeRolePolicy** alters the trust policy for an IAM role; **CreateRole** updates the variables of a to-be-created IAM role and turns on its flag; **AssumeRole** adds the IAM role to the credential list of the attacker. The list of modeled actions and their semantics is presented in Appendix C.

5.6 Modeling Different IAM Exploits

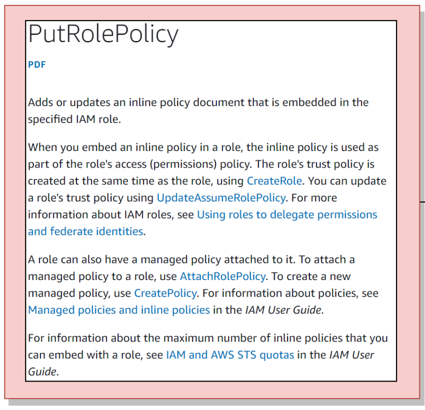
As stated above, the modeling process is an ongoing effort. In order to enrich the model, we must be continuously aware of new IAM exploits. These exploits serve as additional *means* for an attacker to realize their goals, hence modeling them enhances the quality of the model checker findings by detecting a broader range of attacks and reducing false negatives.

We briefly describe several known privilege escalation methods for AWS IAM and how we attempt to model them. These methods are described in the article by Rhino Security Labs [27]. In many cases, elements in the attack can be abstracted if their independence from the IAM policy evaluation logic is identified, e.g. abstract coding details or network layer properties.

Lambda Function Invocation. Lambda functions in AWS are serverless compute objects that provide the user with the ability to run code without provisioning servers. A function runs on behalf of an execution role, which sets the effective permissions for the code execution. An attacker with permission to invoke lambda functions can steal the credentials of any function's role by maliciously overriding the function's original code and instead printing the execution role credentials to any output source accessible to the attacker. Once the attacker gets hold of the role credentials, they can perform requests as if they had already assumed the role. Below is a piece of code in NodeJS that the attacker can configure inside the Lambda function to output the role's access and secret key once the function is invoked.

```
exports.handler = async (event) => {
  const response = {
    statusCode: 200,
    body: {
      "AWS_ACCESS_KEY_ID": process.env.
          AWS_ACCESS_KEY_ID,
      "AWS_SECRET_ACCESS_KEY": process.env.
          AWS_SECRET_ACCESS_KEY,
      "AWS_SESSION_TOKEN": process.env.
          AWS_SESSION_TOKEN
    }};
  return response;
};
```

We model a lambda function as a resource with an execution role, and a Boolean variable representing the state of the function's code – whether it was overridden by an attacker to reveal execution role credentials as a consequence of a



PutRolePolicy

PDF

Adds or updates an inline policy document that is embedded in the specified IAM role.

When you embed an inline policy in a role, the inline policy is used as part of the role's access (permissions) policy. The role's trust policy is created at the same time as the role, using CreateRole. You can update a role's trust policy using UpdateAssumeRolePolicy. For more information about IAM roles, see Using roles to delegate permissions and federate identities.

A role can also have a managed policy attached to it. To attach a managed policy to a role, use AttachRolePolicy. To create a new managed policy, use CreatePolicy. For information about policies, see Managed policies and inline policies in the *IAM User Guide*.

For information about the maximum number of inline policies that you can embed with a role, see IAM and AWS STS quotas in the *IAM User Guide*.

```
Keep(v) := v = v'

action = PutRolePolicy →
  ∀account ∈ accounts:
    (resourceAccount = account.id →
      ∀role ∈ account.IAMRoles:
        (resourceName = role.name → role.InlinePolicy.updated') ∧
        (resourceName != role.name → Keep(role.InlinePolicy.updated))) ∧
    (resourceAccount != account.id →
      ∀role ∈ account.IAMRoles: Keep(role.InlinePolicy.updated)

action != PutRolePolicy →
  ∀account ∈ accounts:
    ∀role ∈ account.IAMRoles: Keep(role.InlinePolicy.updated)
```

Figure 3: PutRolePolicy action API reference, taken from AWS 2022 documentation [7], is translated into a formula that defines the transition relation, i.e., the effect on the organization state. The prime notation x' is used to denote a variable x at the next step

call to **UpdateFunctionCode**. We define the semantics of **InvokeFunction** similarly to **AssumeRole**, only in this case the function's execution role is added to the attacker's credentials list. Observe that an attacker might as well simply override the function with code that performs the next steps on behalf of the execution role, but from our point of view as the model designers, it is more convenient to treat **InvokeFunction** as a "credential reading" action.

EC2 Instance SSH Access. Amazon Elastic Compute Cloud (EC2) instances provide scalable cloud compute abilities on-demand. Each instance is associated with an instance profile that holds a role. An attacker with Secure Shell (SSH) access to a specific instance with IP `IP_ADDRESS` can run a metadata query and retrieve this role's credentials with the following command, assuming the name `ROLE_NAME` is known.

```
TOKEN='curl -X PUT "http://[IP_ADDRESS]/latest/
    api/token" -H "X-aws-ec2-metadata-token-ttl-
    seconds: 21600"' \
&& curl -H "X-aws-ec2-metadata-token: $TOKEN" -v
    http://[IP_ADDRESS]/latest/meta-data/iam/
    security-credentials/[ROLE_NAME]
```

We model an EC2 instance as a resource with an IAM instance profile, and an IAM instance profile as a resource with an IAM role. Since the resolution of whether a given IP source is allowed SSH access to an instance is out of the scope of the policy evaluation logic, we assume global SSH access. We define a "pseudo-action" **"SSH Into Instance"** with semantics similar to **AssumeRole** and **InvokeFunction** – add the associated instance profile role to the attacker's role credentials list. By "pseudo-action" we mean that it is associated with a transition relation, but does not depend on the IAM policy evaluation logic and instead allowed during every step.

6 Model Checking Process

Once we have the model of the AWS IAM component at hand, we run model checking on it in order to detect multi-step attack vectors. We define an initial state formula that reflects the initial organization setup and the initial assumptions for the attacker. While all the constraints discussed in the previous section are safety constraints, i.e., assertions that occur *in every step*, the initial state formula applies *only in the first step*. Every resource is initialized with specific properties according to type. For example, the initial formula for an IAM role defines the initial values for its name, attached policies, and permissions boundary, and asserts that its inline and trust policies are not updated and not deleted. Also, all flags for the to-be-created resources are set to false.

In our motivating example, we get the organization setup that contains a single account with at least one IAM role and one S3 bucket. The role and the bucket each have their respective inline policy documents. There may be other resources in the account that do not participate in the attack.

For the attacker, we assert facts about the initial compromised credentials list. We can assign specific values and detect attacks starting only from specific principals, or we can skip the initial assignment altogether. The latter allows the model checking process to discover all the principals in the organization that can reach the target, by providing different assignments for the initial step. Note the high degree of flexibility – we can exclude certain principals from the list, or come up with theoretical combinations of several compromised principals working in cooperation in order to execute an attack.

In our motivating example, we either explicitly assume that the attacker starts with credentials to role `dept2/admin`, or alternatively, we leave the list unassigned and discover `dept2/admin` upon running the model checker and examining the satisfying assignments of the SAT solver.

Next, we translate an attack target to a property that the model checker attempts to satisfy in one or more iterations. An attack target is best seen as a reachability goal for an attacker, for instance, "perform **GetObject** on bucket `classified`". The property itself is theoretically any proposition expressible in Boolean logic. It can be a simple combination of action and resource as in the motivating example and in our evaluations, or a more complex assertion such as "authenticate as a user from account *a* whose name matches some regular expression and perform a cross-account action on any bucket in account *b* that matches another regular expression". The scope and richness of potential properties can greatly benefit threat modeling teams in executing complex breach and attack simulations.

After the model checker detects a trace, it is translated to an attack vector by extracting the variable assignments to the attacker state in each step. In our example, the assignment in the first step corresponds to the **CreateRole** action on a role resource in the special slot. In the second step, it corresponds to **PutRolePolicy** on the IAM role that was created in the previous step (that has the creation flag set to true), and so forth. Another potential output of the model checker is that no traces were discovered up to a certain depth, or a proof that no traces exist and the system is safe (see future work in Section 10).

Additionally, the model checker may exhaust all results by logically negating a discovered trace and re-running reachability with an extra constraint. This enhances the usability of the approach since often security engineers do not settle for detecting a single attack, but rather seek to gain maximum visibility of the organization's security posture. This enhancement works effectively well when the initial attacker credentials are left unassigned, and all the principals in the organization that can maliciously reach a target in one or more steps are detected. The diagram of the full model checking process is presented in Figure 4.

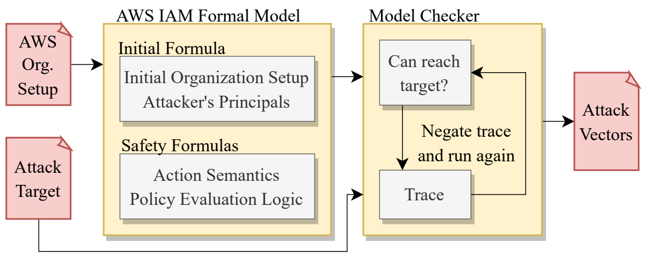


Figure 4: Model Checking Process

7 Implementation

We implement the model construction and model checking in Java, using the Z3 theorem prover open source library [23]. Our implementation consists of reading the AWS organization initial setup and an attack target from a JSON file, constructing the model using the Z3 Java API and admitting it to a custom variant of the bounded model checking (BMC) algorithm from Sheeran et al [43]. Getting the up-to-date organization setup is a technical detail and can be automated by running the **GetAccountAuthorizationDetails** AWS action or other IAM read actions, periodically or on-demand. We implement a Boolean-only model integrated with the optimization presented in Appendix A, along with a model version that also uses string variables. We wish to focus on trace detection, so we strip the original BMC algorithm from the *k*-induction part, leaving only the reachability part.

Observe that the execution of the model checker is done offline, i.e., we can apply any manual modifications to the organization setup file and dry-run a theoretical setup without having to apply the changes in AWS. This characteristic resembles AWS Policy Simulator [10], a tool that lets users simulate and troubleshoot IAM policies before applying them.

Furthermore, to enhance the validity and confidence in our model, we simulated each detected attack vector as a script in a testing AWS environment to ascertain that it indeed produces the expected outcome. An invalid result would mean that the model does not represent correctly the AWS IAM logic. Usually, this is caused by a bug in the formalization that can be immediately fixed, but also due to environment differences in extra-IAM factors such as networking, see Subsection 5.4.1.

8 Evaluation

In order to evaluate our approach we define two research questions examining two different aspects.

- **Quantitative**. How does the approach scale w.r.t. organization volume and length of detected attack vectors?
- **Qualitative**. How does the approach carry out in detecting existing misconfigurations in real-world setups?

8.1 Quantitative Evaluation

To evaluate the performance we devise several simple misconfiguration scenarios that the model checker is expected to detect, similar to the example in section 4. We use real-world AWS organization data supplied to us by a large financial institution. The data consists of nearly 100 accounts with a varying number of IAM resources in each, ranging from tens to thousands. We test our scenarios in the context of two user stories:

- The user chooses a **single account** and a principal belonging to this account, and wishes to detect multi-step attacks assuming the principal is compromised.
- The user chooses a **subset of the organization accounts** and without specifying a principal, wishes to detect privilege escalation attacks, perhaps involving cross-account actions, that may materialize in these accounts.

acc. size range	200-300		300-400		400-500		500-600		600-700	
Model	Bool-Only	Bool+Str	Bool-Only	Bool+Str	Bool-Only	Bool+Str	Bool-Only	Bool+Str	Bool-Only	Bool+Str
Scn. 1	3.79	4.69	4.52	5.10	6.33	6.38	4.67	5.56	6.10	6.52
Scn. 2	5.21	8.00	5.72	8.15	8.52	27.92	6.50	14.93	7.64	14.71
Scn. 3	6.47	20.18	7.22	27.11	12.14	79.98	8.36	39.62	11.37	123.02
Scn. 4	8.67	56.37	13.34	94.30	15.00	280.62	18.02	timeout	23.13	timeout
Scn. 5	16.81	timeout	13.66	timeout	37.60	timeout	19.84	timeout	37.87	timeout

Table 1: Trace detection times in seconds for the single account use case

# acc.	5		10		20		40		80	
Model	Bool-Only	Bool+Str	Bool-Only	Bool+Str	Bool-Only	Bool+Str	Bool-Only	Bool+Str	Bool-Only	Bool+Str
	26.34	149.12	63.09	485.40	84.47	timeout	148.95	timeout	389.42	timeout

Table 2: Trace detection times in seconds for the cross-account use case

The scenarios are injected into the existing setup by modifying the input JSON file to append special resources and policy documents. The names and conditions in the policies are chosen in such a way so that they do not interfere with the existing organization policies and produce unexpected output. The additional setup for all the scenarios is presented in Appendix B. The executions are conducted on a standard Intel Core i7-8650U 1.90GHz laptop with 32GB RAM, running on 64-bit Windows 10 enterprise edition.

For the first, single account test, we choose 5 accounts of varying total IAM resource size ranges (ranging from 200 to 700), and for each account we run 5 scenarios with a fixed initial principal and an expected attack vector of length 1 to 5. For the second, cross-account test, we choose the n largest accounts in terms of IAM resources, where $n = 5, 10, 20, 40, 80$, making for an average of approximately 200 resources per account. We verify the detection of a pre-defined 2-step vector for all n. In this use case, we leave the attacker's principal credentials array unassigned in the initial stage.

We define the total running time as the time for reading the organization file, constructing the model, and performing the model checking until the expected trace is discovered. For each run, we measure the total running time in seconds, once for the model with both Boolean and string variables, and once for the Boolean-only model with the encoding of Appendix A. We set a global timeout limit of 1,200 seconds.

Results. We present the results for the single-account use case in table 1 and for the cross-account use case in table 2. From the results we gather that execution time grows with the model size and with the trace length, which is expected for both approaches. Also, we observe that a larger model does not necessarily predict longer running times, probably due to the heuristic nature of the underlying SAT solver. Note that according to recent reports [1] the majority of real-world attack vectors take a relatively short number of steps, e.g., not more than 5 steps, and furthermore, it would be reasonable to assume that our tested data volumes already cover the scope of many real-world companies' cloud infrastructures.

The Boolean-only model outperforms the model with string variables in all tests. The Boolean-only model takes less than 40 seconds in all the single account scenarios and less than 9 minutes for the cross-account scenarios. In the largest scenarios w.r.t. account size and vector length, the model with string variables often reaches a timeout. This outcome can be explained by the reliance on additional reasoning about string theory in the underlying SMT solver, which is more computationally heavy than Boolean-only algorithms in SAT solvers. In general, one should confine the reasoning to Boolean logic instead of using more complicated data types in order to improve performance, but such effort is not always trivial and depends on careful examination of the domain specific problem.

8.2 Qualitative Evaluation

To evaluate the quality of detected traces, we use real pre-production AWS organization data maintained by a cloud security team in a large financial institution. The organization contains approximately 100 accounts with an average number of around 300 IAM resources per account. The security team was offered an interface to the model checker and was instructed on how to create requests with different targets and organization setups, and how to read the results. Additionally, we holistically reviewed all the findings and attempted to recognize common patterns.

The security team was interested in targets that were simple combinations of action and resource – can an attacker gain read/write permissions on S3 objects or SQS (Simple Queue Service) queues. All the targets were limited to a single account. All the targets left the initial attacker credential list free to discover which principals can reach the target in one or more steps. In figure 5 we show the actions that the security engineers checked. We gathered a total of 141 different requests, of which 42 did not return any results up to a certain depth limit (which in our data was arbitrarily set to 10), while 99 contained results. Recall that we keep only the reachability part in the BMC algorithm, so a "no results" answer is not a proof that there are no traces of greater length, and each such request should be further examined.

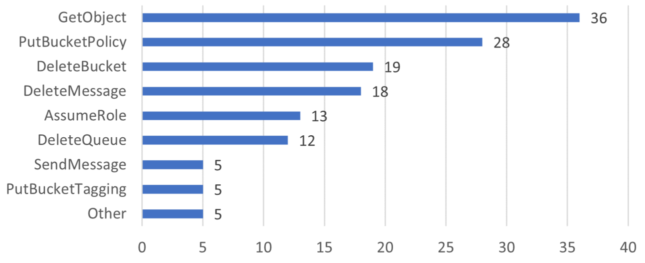


Figure 5: Breakdown of the target actions in the requests gathered from the security team

Results. In Figure 6 we show the average number of detected attack vectors per request (and total), by vector length (number of steps). Observe that the values decrease with length, which is expected and backed up by recent surveys [1]. Only 2 vectors of length 5 were detected. 60% of the results were 1-step vectors, i.e., principals that can immediately reach the target, of which some are principals that are supposed to have the target permissions. The unexpected 1-step vectors are in fact instant IAM misconfigurations in the system rather than multi-step attacks. All of the results were classified as correct – none were affected by extra-IAM conditions. All the requests that didn't contain results were due to the fact that the target was indeed unreachable in the account.

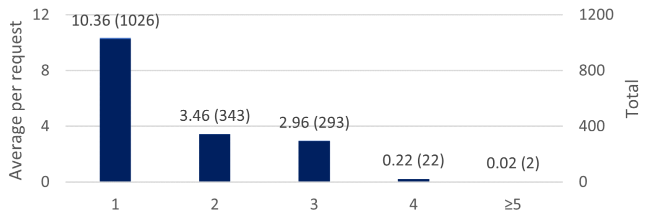


Figure 6: Average number of detected attack vectors per request (and total), by vector length

Additionally, we examined all the inner steps (except the last step – the target itself) in all the attack vectors, for all the requests. In Figure 7 we show a breakdown by the action performed in each step during the attack. We see that some actions are remarkably common, for instance **UpdateAssumeRolePolicy** and **AssumeRole**, which indicates that many potential attacks involve switching between roles to gain privileges. Other common actions are the policy document updating actions such as **PutRolePolicy**. We also observe several recurring patterns or "attack building blocks", namely, a combination of 1-3 steps that are often executed together and result in a privilege escalation, e.g., **CreateFunction** followed by **InvokeFunction**. However, these figures only reflect the cloud infrastructure under test, and cannot indicate the general ubiquity of different AWS actions or vector lengths.

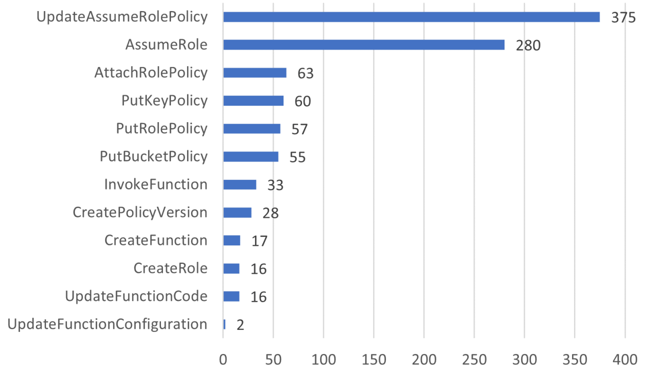


Figure 7: Breakdown of the actions used in all the attack vectors that the model checker detected for all the requests

We also stumbled upon a curious case during the evaluation. We observed an IAM policy whose contents were presumably copy-pasted from a different account, mistakenly leaving the original account identifiers in several locations. Both accounts contained a role with an identical name. The original document included an explicit deny statement prohibiting the role in the first account to perform certain actions. However, in the second account this statement was ineffectual unless the account identifiers were corrected. Hence we were able to detect attack vectors in the second account that exploited this misconfiguration, by allowing the second account role to perform actions that otherwise should have been denied.

In conclusion, the tool helped the security team gain better visibility on their IAM configuration. The security engineers were pleased that many misconfigurations were spotted before the deployment of the policies to production. Considering all the $\geq$2-step traces as real attacks, and that at least a certain percentage of the 1-step vectors was caused by misconfigurations, we estimate that approximately 50% of the findings amounted to cases that should be fixed. Moreover, several attack vectors revealed sneaky bugs in policy documents that could not be detected by other tools or by a human eye. The analysis of each finding w.r.t. metrics such as severity, impact, and remediation is still left to the IAM engineer. Nonetheless, in practice, a change to a single policy document was usually enough to remediate the majority of the attack vectors for a single target.

9 Related Work

Detection of multi-step attacks. Various formal approaches are applied in the domain of software security, specifically the network layer. Ritchey et al. [40] show how to construct a model from a given network configuration, and then apply model checking to detect attack paths in several steps. Similarly to our approach, their model describes the transition from the current state to the next state, thus allowing generation of multi-step attack scenarios. Sheyner et al. [44] use model checking to automatically generate attack graphs that map all the possible counter-examples leading to an undesirable state in the network configuration. These works model a pre-defined list of exploits that the attacker can use in order to progress toward their goal, comparably to our modeling of IAM exploits and actions semantics.

Ammann et al. [11] further discuss the scalability challenges that model checking poses due to the state-explosion problem. In other works [38, 39] the authors utilize a Datalog reasoning engine, rather than model checking, to detect security vulnerabilities in entire networks and generate attack graphs. For these works, the authors assume the *monotonicity* property (see 5.1) for the attacker. More recently, Celik et al. [21] extracted state models from IoT devices and applied model checking to verify security properties.

Verification of access control policies. Attempts to model and verify security configurations in the form of access control policies are proposed in various works [28–31]. These works attempt to verify policy implementations against the original intentions of the authors by expressing various properties in a specification language. These works serve as good evidence of the fact that access control policies profit well from formal treatment. Other works [16, 26, 45] proposed approaches to model and verify properties of firewall and network configurations using formal reasoning. However, these works verify security properties on the existing state of the policies and do not attempt to detect multi-step attack vectors by modeling transitions between states.

Security verification in the cloud. Recently, major cloud providers attempted to use formal methods to validate various security aspects of their cloud offering. For example, Microsoft uses Z3 theorem prover [23] to validate global properties of data center networks in Azure [18, 32]. Amazon pushed the topic forward and revealed TIROS [12], which formally validates network reachability, and ZELKOVA [14, 15], which detects misconfigurations in IAM policies. Specifically, ZELKOVA attempts to verify whether an S3 bucket has public access. To answer this question, they formalize the "having public access" property along with the policy evaluation logic and the network layer, and evaluate it using an SMT solver. Amazon also applied formal methods in an attempt to explain correctness of policies [13]. A recent open source tool IAMSPY [34] also models AWS policy evaluation logic, and using Z3 theorem prover answers whether an action is permitted on a resource. Cauli et al. [20] suggested formal reasoning to assess the security of cloud infrastructure before deployment. However, these works do not model the transition relation between steps, but focus on analyzing the current state of the organization.

Static analysis tools for cloud security. Finally, configuration scanners and static code analysis tools (often called linters) exist in abundance as open source and proprietary libraries. For example, Rhino Security Labs released PACU [33] following an article [27] listing multiple privilege escalation methods exploiting AWS IAM. Similar tools [37] operate on a variety of cloud environments. However, these approaches do not analyze the policies in a formal manner, but rather scan them for occurrences of dangerous permissions according to common best-practice recommendations.

10 Discussion and Future Work

Recent efforts by the major cloud providers demonstrate that formal reasoning can indeed be suited to security concerns, yet, to quote from a recent survey [41], formal methods will not answer questions that are not posed and models are limited reflections of reality. Our formal depiction of AWS IAM is neither *complete*, as the modeling of all the vulnerable AWS elements is a long process, nor *sound*, as we consciously abstract away certain elements while risking false positives. Yet, we envision that with time, the formal model will consolidate into a proven and reliable representation of AWS IAM as judged by domain professionals. It will happen via a process where security researchers discover new IAM exploits, and developers integrate these exploits into the model by formalizing the relevant action semantics. Eventually, the model will become an integral part of a security compliance solution for cloud customers.

For future research, we suggest the following directions. In this paper we focused only on detecting vulnerabilities in the IAM configuration, while model checking is also capable of providing a proof that a configuration is safe. An interesting direction would be to try to apply different model checking algorithms such as Property Directed Reachability (PDR) [25] or interpolation based techniques [36], in order to provide formal assurance that organizations are hermetically secure from specific attack targets. Moreover, we can employ advanced APIs of SAT solvers, such as unsatisfiable core computation, to find a minimal set of policies effective for allowing or denying a certain permission.

Additional direction is to integrate numerical weights into the model based on external security assessments and be able to rank results by severity or likelihood. Various SMT solvers are known to support such optimization problems. Then, attacks involving sensitive principals may precede potential low severity attacks. Also, one may explore different ways to automatically sort and repair potential false positive traces.

Another natural direction is to extend the support to other environments and domains. For example, Google and Microsoft offer cloud services as well, and IAM misconfigurations are a prevailing concern there as much as in AWS.

Furthermore, network solutions and technologies have greatly developed since the days of Ammann et al. [11], so one may try to reevaluate the idea of finding network security issues using model checking, inspired by the approaches described in this paper.

11 Conclusion

We presented a novel approach to construct a Boolean model of the AWS IAM component. This model is formally checked for the presence of multi-step attacks involving privilege escalation by exploiting IAM misconfigurations. We implemented the model checking process and evaluated the performance of trace detection, obtaining very encouraging results. We also evaluated the ability to discover real attacks in a cloud environment belonging to a large institution. Overall, we can conclude that applying formal reasoning to detect IAM multi-step attacks in cloud environments is indeed a feasible task and can serve as a reliable mechanism to provide additional assurance to security experts.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their constructive feedback. This research was funded by the Citi TLV Cyber Security Innovation Center. We thank Uri Kahana, Max Leibovich, Revital Gavriel, Gil Fitussi, and Mickey Hovel from the Citi TLV Cyber Security Innovation Center for the implementation and deployment efforts, and we thank Jeanette Cheng, Yong Hang Lin, and David Gubitosi from the Citi Cloud Security Engineering team for providing the testing pre-production organization data and participating in the evaluation efforts.

References

[1] 2020 dbir results and analysis - verizon enterprise solutions. https://www.verizon.com/business/resources/reports/dbir/2020/results-and-analysis/, 2020.

[2] Owasp top ten web application security risks. https://owasp.org/www-project-top-ten, 2021.

[3] Cross-account policy evaluation logic. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic-cross-account.html, 2022.

[4] Iam json policy elements reference - aws documentation. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html, 2022.

[5] Mitre att&ck®. https://attack.mitre.org/, 2022.

[6] Policy evaluation logic - aws identity and access management. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic.html, 2022.

[7] Putrolepolicy api reference. https://docs.aws.amazon.com/IAM/latest/APIReference/API_PutRolePolicy.html, 2022.

[8] Service control policies (scps) - aws organizations. https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html, 2022.

[9] Shared responsibility model - amazon web services (aws). https://aws.amazon.com/compliance/shared-responsibility-model/, 2022.

[10] Testing iam policies with the iam policy simulator. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_testing-policies.html, 2022.

[11] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In Vijayalakshmi Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 217–224. ACM, 2002.

[12] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark A. Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. Reachability analysis for aws-based networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 231–241. Springer, 2019.

[13] John Backes, Ulises Berrueco, Tyler Bray, Daniel Brim, Byron Cook, Andrew Gacek, Ranjit Jhala, Kasper Søe Luckow, Sean McLaughlin, Madhav Menon, Daniel Peebles, Ujjwal Pugalia, Neha Rungta, Cole Schlesinger, Adam Schodde, Anvesh Tanuku, Carsten Varming, and Deepa Viswanathan. Stratified abstraction of access control policies. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2020.

[14] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[15] John Backes, Pauline Bolignano, Byron Cook, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Martin Schäf, Cole Schlesinger, Rima Tanash, Carsten Varming, and Michael W. Whalen. One-click formal methods. *IEEE Softw.*, 36(6):61–65, 2019.

[16] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 155–168. ACM, 2017.

[17] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[18] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in microsoft azure. In Raja Natarajan, Gautam Barua, and Manas Ranjan Patra, editors, *Distributed Computing and Internet Technology - 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings*, volume 8956 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2015.

[19] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[20] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. Pre-deployment security assessment for cloud services through semantic reasoning. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 767–780. Springer, 2021.

[21] Z. Berkay Celik, Patrick D. McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 147–158. USENIX Association, 2018.

[22] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[23] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[25] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134. FMCAD Inc., 2011.

[26] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 469–483. USENIX Association, 2015.

[27] Spencer Gietzen. Aws iam privilege escalation – methods and mitigation. https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation/, Jun 2021.

[28] Antonios Gouglidis, Ioannis Mavridis, and Vincent C. Hu. Security policy verification for multi-domains in cloud systems. *Int. J. Inf. Sec.*, 13(2):97–111, 2014.

[29] Dimitar P. Guelev, Mark Ryan, and Pierre-Yves Schobbens. Model-checking access control policies. In Kan Zhang and Yuliang Zheng, editors, *Information Security, 7th International Conference, ISC 2004, Palo Alto, CA, USA, September 27-29, 2004, Proceedings*, volume 3225 of *Lecture Notes in Computer Science*, pages 219–230. Springer, 2004.

[30] Vincent C. Hu, D. Richard Kuhn, Tao Xie, and JeeHyun Hwang. Model checking for verification of mandatory access control models and properties. *Int. J. Softw. Eng. Knowl. Eng.*, 21(1):103–127, 2011.

[31] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a SAT solver. *Int. J. Softw. Tools Technol. Transf.*, 10(6):503–520, 2008.

[32] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C. Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 200–213. ACM, 2019.

[33] Rhino Security Labs. pacu: The aws exploitation framework. https://github.com/RhinoSecurityLabs/pacu, 2022.

[34] With Secure Labs. Withsecurelabs/iamspy. https://github.com/WithSecureLabs/IAMSpy, 2022.

[35] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[36] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[37] nccgroup. nccgroup/pmapper: A tool for quickly evaluating iam. https://github.com/nccgroup/PMapper, 2022.

[38] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A scalable approach to attack graph generation. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 336–345. ACM, 2006.

[39] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[40] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pages 156–165. IEEE Computer Society, 2000.

[41] Kim Schaffer and Jeffrey M. Voas. What happened to formal methods for security? *Computer*, 49(8):70–79, 2016.

[42] Dave Shackleford. Sans 2019 cloud security survey. 2019.

[43] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods*

in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[44] Oleg Sheyner, Joshua W. Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *2002 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 12-15, 2002*, pages 273–284. IEEE Computer Society, 2002.

[45] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 199–213. IEEE Computer Society, 2006.

A Translating Theories to Boolean Formulas

We describe a technique to translate variables of certain data types such as strings into an array of Booleans, taking advantage of the fact that for the purpose of policy evaluation we only need to know whether a resource name matches a regular expression. Many SMT solvers keep extra information about a string instance to enrich their reasoning power. For instance, SMT solvers may provide support for operators such as "length", "indexOf", and so on, all of which we do not need in our model. Recall Figure 1 where a policy document contained conditions based on regular expression matching, and our translation that contained `resourceName ∈ "*-suf"`. The variable `resourceName` is in fact a Boolean array, where each Boolean represents whether the name matches a specific regular expression. We focus on strings, but this technique can be applied to other types such as integers, arrays, dates or IP ranges.

During the policy document parsing, we collect the regular expressions that affect resources by type. Going back to the example in Sect. 4, we collect "`dept1/*`", "`*dept2/*`", and "`dept2/admin`" for the IAM role resource type. We treat constant strings as a regular expression that is matched by a single element. The length of the underlying Boolean array is the number of distinct regular expressions, so for a role in the motivating example it is 3.

An assignment to this array represents a set of strings that match the relevant regular expressions. The order of the assignments is irrelevant, so we use the notation $name[regex]$ to refer to the Boolean that encodes $name \in regex$. Logical equivalence between two string variables, i.e., $name_1 = name_2$ is modeled as $\bigwedge(name_1[regex] = name_2[regex])$ over all regular expressions according to resource type. Updating the string in the next step is similarly modeled as $\bigwedge(name_1[regex] = name'_2[regex])$. Observe that setting $name[dept1/*] = true$, can be interpreted as having the name "`dept1/Alice`", but also "`dept1/Bob`" – these names are equivalent in the sense that an attack conducted using one name can be replicated using the other name.

An enumeration or a list in our model are Boolean arrays where each variable represents equivalence to a single value (or containment of the value in the list). For enumerations we add a mutual exclusion formula between its Boolean variables, whereas for strings we must pre-compute the constraints because not all the regular expression combinations are legal for a given resource type. For example, a string cannot match both "`A*`" and "`B*`", so we must add a formula stating that $name[A*] \rightarrow \neg name[B*]$. Likewise, if we have "`A*`" and "`AA*`" we add $name[AA*] \rightarrow name[A*]$. These restrictions are in fact pre-computed using an SMT-solver that supports string theory, which is employed during model construction solely for this purpose but not for the main model checking algorithm. The computed constraints formula is added to the semantics of the create action, in order to prevent attacks where an attacker creates a resource with an illegal name that bypasses regular expression conditions. We show in Alg. 1 a procedure to compute the constraints formula for a Boolean array *str* given a regular expression list *regexes*.

We remark that this algorithm is not complete, as only pairs of regular expressions are checked in order to determine the relationship between them, and not the full power-set in order to discover constraints tied to groups of three and more variables. However, it can be shown that going over all the pairs is enough when the regular expressions contain only "`*`" (and not the single character wildcard "`?`").

Algorithm 1 Compute string constraints according to an array of regular expressions

```
1:  procedure GetConstraints(str, regexes)
2:      constr ← []
3:      s ← new SMT-Solver
4:      tester ← new SMT-String
5:      for reg1, reg2 over distinct regex pairs do
6:          formula ← tester ∈ reg1 ∧ tester ∈ reg2
7:          if s.check(formula) == unsat then
8:              constr.add(str[reg1]→ ¬str[reg2])
9:              break
10:         formula ← tester ∉ reg1 ∧ tester ∈ reg2
11:         if s.check(formula) == unsat then
12:             constr.add(str[reg2]→str[reg1])
13:             break
14:         formula ← tester ∈ reg1 ∧ tester ∉ reg2
15:         if s.check(formula) == unsat then
16:             constr.add(str[reg1]→str[reg2])
17:             break
18:     return conjunction over all constr
```

After a trace is found, an attack is constructed from the satisfying assignment. For string variables, a matching string is invented from the assignment to the corresponding Boolean array. This is done in the procedure described in Alg. 2. The function receives a Boolean assignment *asgn* from the model checking algorithm and the corresponding *regexes* list of the same length, and extracts a string from the SMT-solver. The assertion in line 10 checks that the formulas being added to the solver are satisfiable. In case this assertion fails, there might be a bug in the constraints computed in Alg. 2, and the assignment to the variables in *asgn* cannot produce a legal string.

Observe how an SMT-solver is utilized only in Alg. 1 and Alg. 2 to perform the necessary formal reasoning on strings, leaving the IAM model to depend solely on Boolean variables. This allows us to utilize a fast Boolean SAT-solver for the heavy-duty task of model checking and trace detection.

Algorithm 2 Propose string according to the assignment from the model checking

```
1:  procedure ProposeString(asgn, regexes)
2:      s ← new SMT-Solver
3:      tester ← new SMT-String
4:      for i ← 0 to regexes.length do
5:          reg ← regexes[i]
6:          if asgn[i] == true then
7:              s.add(tester ∈ reg)
8:          else
9:              s.add(tester ∉ reg)
10:     assert(s.check() == sat)
11:     return s.evaluate(tester)
```

B Benchmark Setup

In every scenario we add a non-encrypted test S3 bucket named `classified`, a test S3 object in this bucket, and a test IAM role. The attack target for all the scenarios is defined as **GetObject** on any object in `classified`. In the single account scenarios we add the resources to the same account. In the cross-account scenario we add the bucket to the largest account, and the IAM role to the second largest account, in order to deliberately create a cross-account attack. Observe that essentially only two accounts are required for the attack to be realized, and the other accounts are present solely for examining the scalability factor. The expected attack vector length for the single account scenarios is 1-5 respectively.

Scenario 1. `dept2/Role` is immediately allowed to access the classified bucket by virtue of its name. This scenario serves as a one-step sanity check. There is no inline policy for `dept2/Role` and the resource policy for `classified` is defined as follows.

```
"Effect": "allow",
"Action": "GetObject",
"Principal": "dept2/*",
"Resource": "classified/*"

"Effect": "deny",
"Action": "GetObject",
"NotPrincipal": "dept2/*",
"Resource": "classified/*"
```

Scenario 2. `dept1/Admin` executes **PutBucketPolicy** on themselves, then **GetObject**. There is no resource policy for `classified` and the inline policy for `dept1/Admin` is defined as follows.

```
"Effect": "allow",
"Action": "*Role*",
"Resource": "dept1/*"
```

Scenario 3. `dept1/Admin` creates a role with any name, in particular with a name adhering to `dept2/*`. So **CreateRole** is followed by **AssumeRole** and then **GetObject** via the new role. The resource policy for `classified` is defined as in Scenario 1 and the inline policy for `dept1/Admin` is defined as follows.

```
"Effect": "allow",
"Action": "*Role*",
"Resource": "*"

"Effect": "deny",
"Action": "*Role*",
"Resource": "dept1/Admin"
```

Scenario 4. `dept1/Admin` creates a role with a name adhering to `dept1/*`. So **CreateRole** is followed by **PutRolePolicy** to elevate the privileges of the created role, then **AssumeRole**, and finally **GetObject** via the new role. There is no resource policy for `classified` and the inline policy for `dept1/Admin` is defined as follows.

```
"Effect": "allow",
"Action": "*Role*",
"Resource": "dept1/*"

"Effect": "deny",
"Action": "*Role*",
"Resource": "dept1/Admin"
```

Scenario 5. `dept1/Admin` creates a role with a name adhering to `dept1/*`, but now there is a resource-based explicit deny. So **CreateRole** is followed by **PutRolePolicy** to elevate the privileges of the created role, then **AssumeRole**, then **PutBucketPolicy** or **DeleteBucketPolicy** to remove the deny statements of the bucket, and finally **GetObject**. The resource policy for `classified` is defined as in Scenario 1, and the inline policy for `dept1/Admin` is defined as in Scenario 4.

Cross-Account Scenario. `dept2/Role` executes **PutRolePolicy** on themselves, granting them **GetObject** allow permission needed in cross-account requests for the identity-based policy (in addition to the already existing allow permission in the resource-based policy) and then accesses the bucket. This is a two-step vector. The resource policy for `classified` is defined as in Scenario 1, and the inline policy for `dept2/Role` is defined as follows.

```
"Effect": "allow",
"Action": "*Role*",
"Resource": "*"
```

C Modeled AWS Elements List

Table 3 lists the *means* actions whose semantics are formalized in the model, and their associated resource types.

Service	Resource Type	Action	Semantics
IAM	User	CreateUser	Turns on a to-be-created user resource flag
		UpdateUser	Updates the name attribute of the user
		CreateLoginProfile	Adds the user to the attacker's credentials list
		UpdateLoginProfile	Same as above
		PutUserPolicy	Updates user's inline policy to fully privileged (updated state)
		DeleteUserPolicy	Removes users's inline policy (deleted state)
		AttachUserPolicy	Adds a given managed policy to user's attached policies list
		DetachUserPolicy	Removes a given managed policy from user's attached policies list
		PutUserPermissionsBoundary	Updates user's permissions boundary policy
		DeleteUserPermissionsBoundary	Removes user's permissions boundary policy
	Group	CreateGroup	Turns on a to-be-created group resource flag
		UpdateGroup	Updates the name attribute of group
		PutGroupPolicy	Updates group's inline policy to fully privileged (updated state)
		DeleteGroupPolicy	Removes group's inline policy (deleted state)
		AttachGroupPolicy	Adds a managed policy to group's attached policies list
		DetachGroupPolicy	Removes a managed policy from group's attached policies list
		AddUserToGroup	Adds the group to a user's groups list
		RemoveUserFromGroup	Removes the group from a user's groups list
	Role	CreateRole	Turns on a to-be-created role resource flag
		AssumeRole (STS service)	Adds the role to the attacker's credentials list with a new session name
		UpdateAssumeRolePolicy	Updates role's trust policy to fully privileged (updated state)
		PassRole	Auxiliary action that must be allowed on several occasions, such as when setting a lambda execution role, or an instance profile role
		PutRolePolicy	Updates role's inline policy to fully privileged (updated state)
		DeleteRolePolicy	Removes role's inline policy (deleted state)
		AttachRolePolicy	Adds a managed policy to role's attached policies list
		DetachRolePolicy	Removes a managed policy from role's attached policies list
		PutRolePermissionsBoundary	Updates role's permissions boundary policy
		DeleteRolePermissionsBoundary	Removes role's permissions boundary policy
	Policy	CreatePolicy	Turns on a to-be-created policy resource flag
		CreatePolicyVersion	Updates the policy to fully privileged (must have less than 5 versions)
		DeletePolicyVersion	Deletes an arbitrary policy version (to have less than 5 versions)
		SetDefaultPolicyVersion	Updates the default version of the policy
	Instance Profile	CreateInstanceProfile	Turns on a to-be-created instance profile resource flag
		AddRoleToInstanceProfile	Updates instance profile's role
		RemoveRoleFromInstanceProfile	Removes instance profile's role
Organizations	SCP	UpdatePolicy	Updates the service control policy to fully privileged (updated state)
	Account/OU	AttachPolicy	Adds an SCP to account's or to organizational unit's SCP list
		DetachPolicy	Removes an SCP from account's or from organizational unit's SCP list
Lambda	Function	CreateFunction	Turns on a to-be-created function resource flag
		InvokeFunction	Adds the function's execution role to the attacker's credentials list
		AddPermission	Updates functions's resource policy to fully privileged (updated state)
		UpdateFunctionConfiguration	Updates function's execution role
		UpdateFunctionCode	Updates function's code to reveal role credentials on execution
	Event Source Mapping	CreateEventSourceMapping	Adds the EventSourceMapping's execution role to the credentials list
		UpdateEventSourceMapping	Same as CreateEventSourceMapping
EC2	Instance	RunInstances	Turns on a to-be-created instance resource flag
		"SSH into instance"	Adds the instance's role to the attacker's credentials list
		AssociateInstanceProfile	Updates instance's instance role in case it was empty
		DisassociateInstanceProfile	Removes instance's instance role
		ReplaceInstanceProfile	Updates instance's instance role in case it was already set
S3	Bucket	PutBucketPolicy	Updates bucket's resource policy to fully privileged (updated state)
		DeleteBucketPolicy	Removes bucket's resource policy (deleted state)
KMS	Key	PutKeyPolicy	Updates key's resource policy to fully privileged (updated state)
		Decrypt	Must be allowed when accessing encrypted resources such as S3 buckets
Glue		CreateDevEndpoint	Adds the DevEndpoint's role to the attacker's credentials list
CloudFormation	Stack	CreateStack	Adds the stack role to the attacker's credentials list
		UpdateStack	Same as CreateStack
DataPipeline	Pipeline	CreatePipeline	Turns on a to-be-created pipeline resource flag
		PutPipelineDefinition	Updates pipeline's role
		ActivatePipeline	Adds the pipeline's role to the attacker's credentials list

Table 3: Modeled AWS actions and their respective semantics