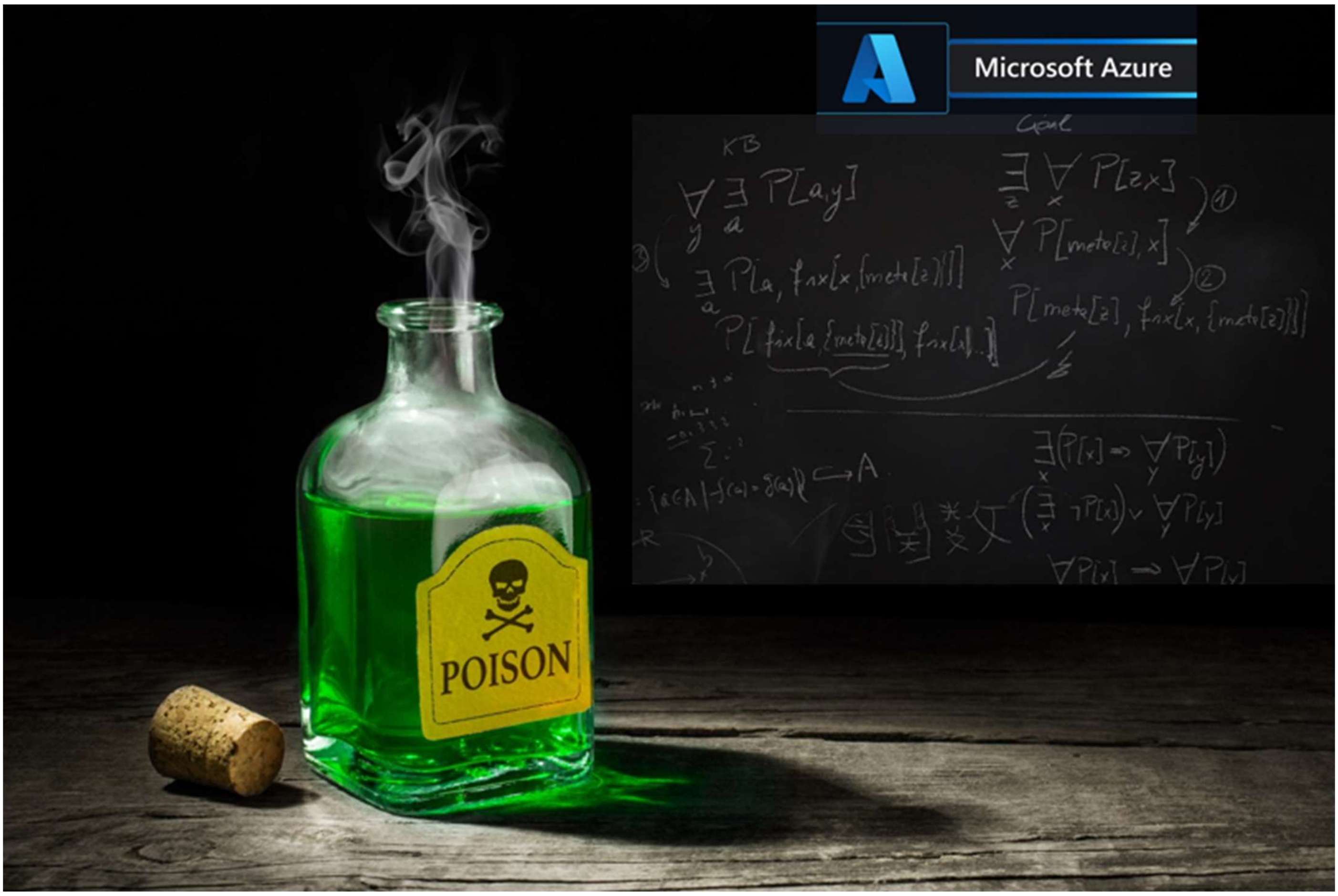
“Toxic Locks”

Monitor Just In Time (JIT) access to Azure PaaS
With automated reasoning



Version 1.0 (August 12, 2022)

Author: Christophe PARISEL

Permalink:

github.com/labyrinthinesecurity/automatedReasoning/2_ToxicLocks.pdf

Introduction

How do you oversee Just In Time (JIT) access to Azure PaaS data or control plane when cloud native tools (like PIM or Bastion) don't get you covered?

After the Hub & Spoke pattern prover, **Toxic Locks** is my second control for automated reasoning. (As a matter of fact, Toxic Locks went live several months before Hub & Spoke because I had an urgent need for it…)

Unlike the Hub & Spoke prover[2] which needed abstract constructs like uninterpreted functions and arithmetization, Toxic Locks only relies on elementary Satisfiability Modulo Theories (SMT): congruence closures and bitvectors theory.

If you're not familiar with SMT, I strongly recommend you take a look at my Primer on automated reasoning [3], or Microsoft Research Z3 programming [5].

The foundations of automated reasoning in the Cloud

Earlier this year, echoing Werner Vogel's call [4] for raising awareness around the benefits of automated reasoning for Cloud computing, I laid down the foundations of a framework [1] on top of which Cloud Customers automated reasoning controls could be built. I also explained how it would complement native Cloud policy-driven compliance (like Azure Governance and AWS CloudFormation hooks), software factories and devSecOps operating models to bring the highest standards of provable security in customer deployments.

Three components form the core of this framework:

1. **Authoritative sources**: these are trusted repositories from the Cloud provider and/or from the customer, from where all *axioms* required for theorems proving are distributed through API endpoints. Some authoritative sources deliver ground truth, while others deliver static reference data;
2. **Customer controls**: these are logical statements that represent the security enforcement expected by the customer;
3. A **SMT solver** able to prove customer controls by a process called *derivation*. The solver feeds from axioms and already proven logical statements. The Solver we're going to use in this paper is Microsoft's Z3.

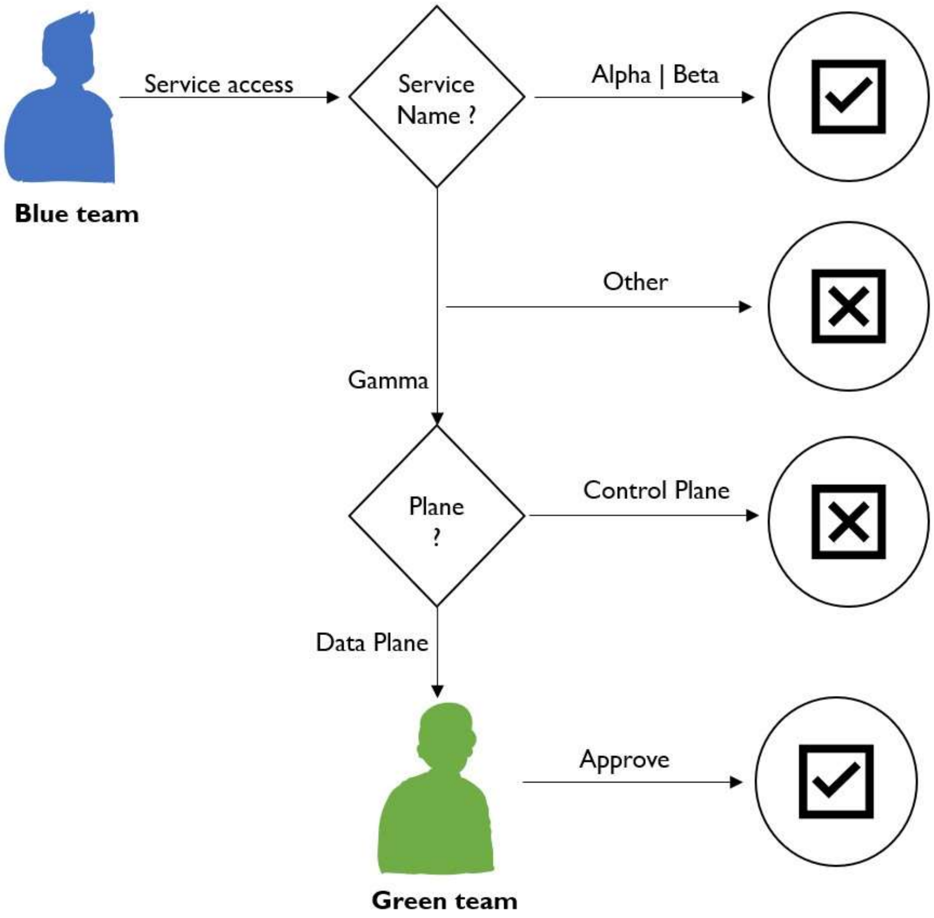
A customer control is unproven until formally derived by the solver. If the control is proven, it becomes a *theorem*. If it is disproved, it becomes a *nontheorem* and a *control anomaly* must be dispatched to the customer monitoring backend.

Problem statement

Say that a **Blue** team has Contributor role over 3 Azure services scoped to a given subscription. The services are called *alpha*, *beta*, and *gamma*. **Blue** is not allowed to modify gamma's data plane without requesting just-in-time (JIT) access for a limited time. Azure PIM only supports JIT access to a service control plane, so PIM cannot be used in this situation: we have to find another solution.

A native solution to prevent auto-approval is to implement our own gatekeeping mechanism with Azure RBAC: a **Green** team is called forth for gatekeeping **Blue**'s JIT access requests to gamma's data plane.

Here are how the daily operations of **Blue** and **Green** play out in the subscription:



But there is a catch: as a pure control team, **Green** must not hold any operational role in the subscription. So, in practice, it has neither data nor control plane access to alpha, beta, or gamma.

Consequently, **Green** cannot approve or deny requests to gamma coming from **Blue** using only Azure Resource Provider permissions, as this would require some control plane access to gamma.

To get around this chicken and egg problem, the gamma service is contained in a dedicated “gamma resource group” upon which an Azure ReadOnly lock is set. Only **Green** is allowed to put or to remove the lock.

When **Blue** issues a request to make a change to gamma, **Green** removes the lock on the gamma resource group: this is functionality equivalent to a formal approval since from then on, **Blue** becomes entitled to modify gamma. When **Blue**’s operations are over, **Green** restores the lock and **Blue** cannot make any further change. (The lock may also be restored programmatically after a preset maximum duration).

Mathematical modeling of the solution

To make sure this workflow works as intended, the following properties must be always met:

1. **NO-GREEN-OPS**: **Green** must have no permissions in any Azure Resource provider scoped to the subscription (or below), except in the *Microsoft.Authorizations* Resource Provider that is in charge of handling locks,
2. **CONTAINED-GAMMA**: all gamma instances in the subscription must be either held in a locked resource group, or in an unlocked resource group which has been unlocked for no more than a grace period
3. **NO-BLUE-LOCK**: **Blue** must have no permissions of type *Microsoft.Authorizations/{*|delete|write|action}* scoped on the gamma resource group,

Note that we would also need one last property:

- **LIMITED-BLUE-OPS**: at the subscription scope (and below), all **Blue** permissions of type *{*|delete|write|action}* must be limited to the alpha, beta, and gamma Resource Providers.

We do not implement **LIMITED-BLUE-OPS** to keep this article short, keeping in mind it’s very straightforward to do.

If we can provide a mathematical proof that all properties always hold (figure 1), we will have managed to make a bullet proof oversight over the whole workflow. Such a solution is called a *provable* control.

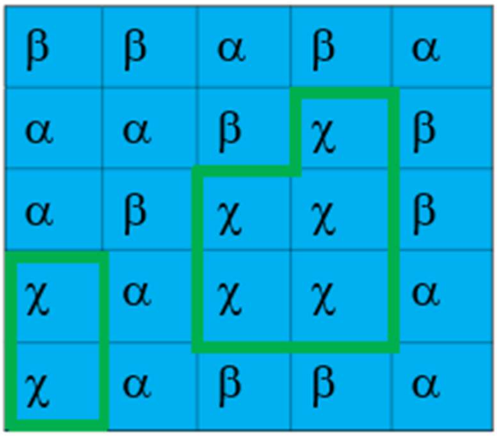
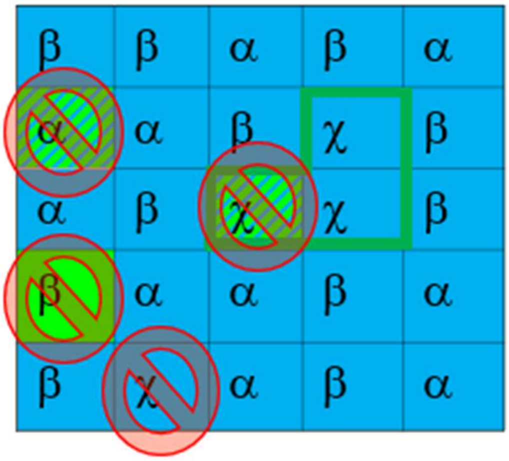


Figure 1: nominal case. All properties are “True”.

Figure 2 depicts a few examples where the first two properties are false:

- One alpha instance data or control plane can be modified by **Blue** and **Green** teams (**NO-GREEN-OPS** is False);
- One beta instance data or control plane can be modified by the **Green** team (**NO-GREEN-OPS** is False);
- One gamma instance is not protected by a lock (**CONTAINED-GAMMA** is False);
- One gamma instance control or data pane can be modified by **Blue** and **Green** teams (**NO-GREEN-OPS** is False).

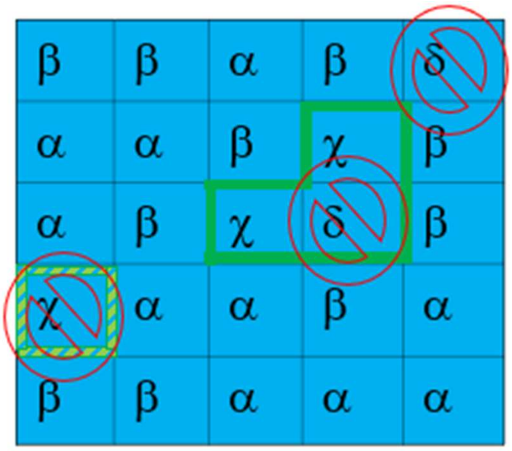


β	β	α	β	α
α	α	β	χ	β
α	β	χ	χ	β
β	α	α	β	α
β	χ	α	β	α

Figure 2: anomalies. CONTAINED-GAMMA and NO-GREEN-OPS are “False”.

Figure 3 depicts examples where the last two properties are False:

- Two delta instances data or control plane can be modified by **Blue** (**LIMITED-BLUE-OPS** is False);
- One gamma resource group lock can be modified by the **Blue** and **Green** teams (**NO-BLUE-LOCKS** is False).



β	β	α	β	δ
α	α	β	χ	β
α	β	χ	δ	β
χ	α	α	β	α
β	β	α	α	α

Figure 3: yet another group of anomalies. NO-BLUE-LOCKS and LIMITED-BLUE-OPS are “False”.

Provable control design

To translate our properties into logical statements to be derived by Z3, we are going to break down the problem:

1. The “unlocked sensitive resources” subproblem: Identify resource groups with sensitive assets, and make sure they are either locked or unlocked since less than an upper bound
2. The “toxic combinations” subproblem: look for principals who have permissions on these locks, and who have permissions on the sensitive assets at the same time

Unlike the Hub & Spoke pattern prover, we are going to need not just one but three authoritative sources:

- *Azure Resource Graph* will be required to pull resource groups and sensitive assets;
- *Azure RBAC (in Azure Active Directory)* will be required to pull locks, principals and role assignments.
- *Azure Activity Logs* of the Microsoft.Authorization resources provider will be required to know when management locks are added or deleted.

The idea to tackle both subproblems is to leverage the power of Z3 to reason about congruence closures to partition the space into equivalence classes: a first partition splits the space into sensitive and non-sensitive resource groups, a second partition splits the space into lockmasters and non-lockmaster principals.

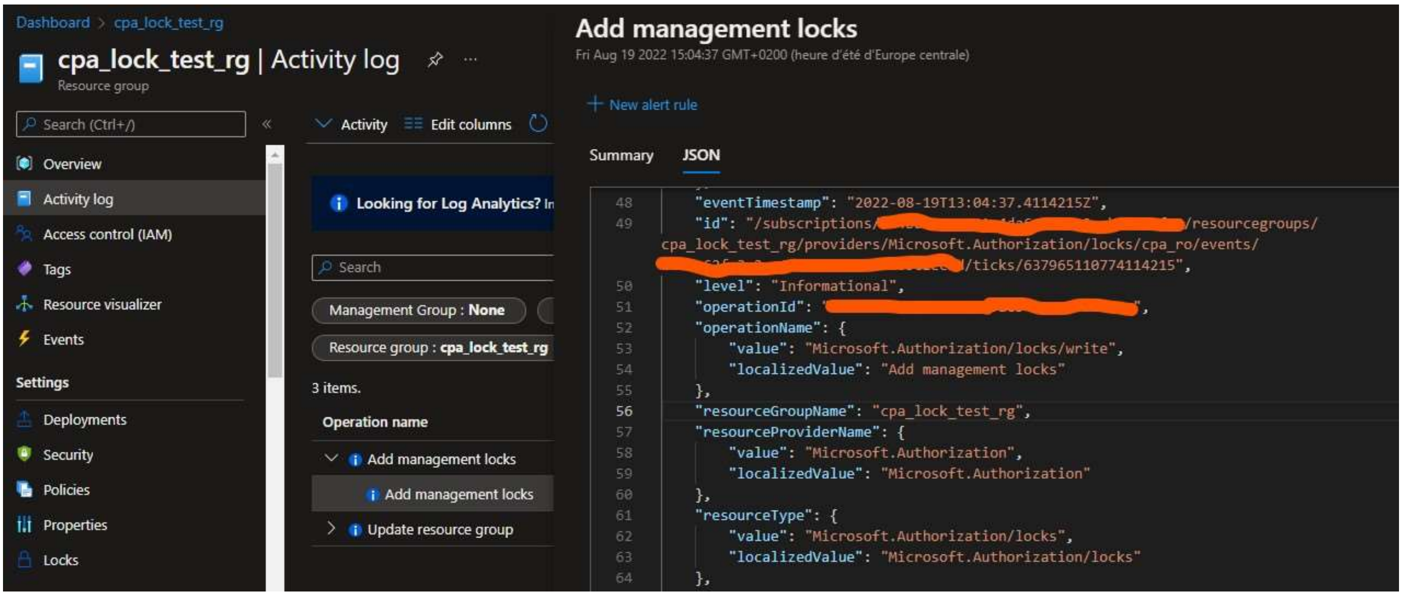
For handling time constraints, we resort to bitvectors logic.

Unlocked sensitive resources

The first part of the control addresses sensitive assets lock protection:

1. We create two Z3 asset sorts: “LOCKED” and “UNLOCKED”;
2. We add the following Z3 formula: LOCKED != UNLOCKED to ensure that LOCKED and UNLOCKED belong to separate equivalence classes;
3. Using ARG, we identify all sensitive assets (we may restrict the scope to a management group);
4. From the Azure resource ID of a sensitive asset, we infer its Resource Group ID (example in red: /subscriptions/abcdef/resourgeGroups/myResourceGroup/provider/Microsoft.Network/...)
5. We add all such Resource Group IDs to the “LOCKED” equivalence class;
6. Using Azure RBAC, we identify all locks (we may restrict the scope to the same management group as above);
7. For each lock, we determine its **level** (e.g.: ReadOnly, NotDelete, ...)
8. From the Azure resource ID of a lock, we infer its Resource Group ID as above;
9. If the lock is at level ‘ReadOnly’, we add its Resource Group ID to the “LOCKED” equivalence class. Otherwise, we add it to the “UNLOCKED” class.
10. If adding the RG ID of a not ReadOnly lock to the “UNLOCKED” equivalence class is NOT SATISFIABLE, it means that the RG ID has already been identified (in step 5) as LOCKED because it contains a sensitive asset. The lock should be ReadOnly but isn’t.
11. We poll Azure Activity logs to determine when the ReadOnly lock was deleted. We are then able to feed Z3’s BitVector logic to prove deletion happened since no longer than the grace

period. Exceeding this time span will produce a nontheorem and raise a **control anomaly** (**CONTAINED-GAMMA** is False).



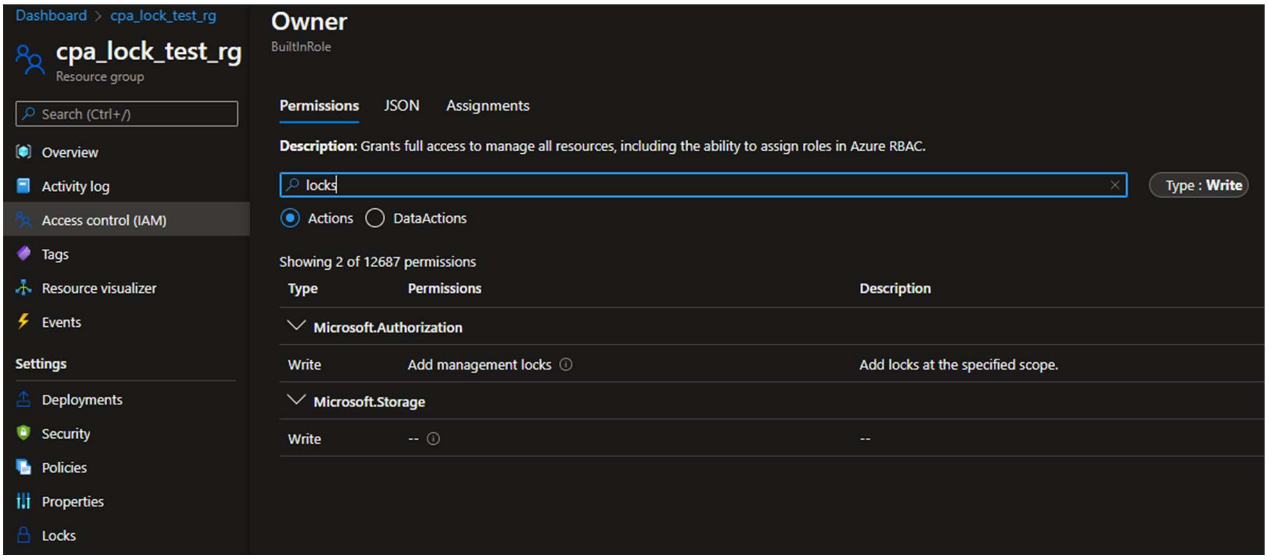
```
    "eventTimestamp": "2022-08-19T13:04:37.4114215Z",
    "id": "/subscriptions/[redacted]/resourcegroups/
cpa_lock_test_rg/providers/Microsoft.Authorization/locks/cpa_ro/events/
[redacted]/ticks/637965110774114215",
    "level": "Informational",
    "operationId": "[redacted]",
    "operationName": {
        "value": "Microsoft.Authorization/locks/write",
        "localizedValue": "Add management locks"
    },
    "resourceGroupName": "cpa_lock_test_rg",
    "resourceProviderName": {
        "value": "Microsoft.Authorization",
        "localizedValue": "Microsoft.Authorization"
    },
    "resourceType": {
        "value": "Microsoft.Authorization/locks",
        "localizedValue": "Microsoft.Authorization/locks"
    },
```

Figure 4: manually picking the timestamp of an "Add Azure lock event" in Azure Activity Logs.

Toxic combinations

The second part of the control addresses toxic combinations:

1. We create two Z3 asset sorts: "LOCKMASTER" and "LOCKSLAVE"
2. We add the following Z3 formula: LOCKMASTER != LOCKSLAVE to ensure that LOCKMASTER and LOCKSLAVE belong to separate equivalence classes;
3. We query AAD RBAC to collect all role assignments. We add the ID of each assignment able to modify the *Microsoft.Authorizations* Resource Provider to the LOCKMASTER class. The other role assignments IDs are added to the LOCKSLAVE class;
4. We query AAD RBAC to collect the principals (including apps and Managed Identities) which have role assignments at management group scope or below (i.e. at subscription and resource group scopes). If an AAD group is returned, we expend it recursively to make sure we collect all principals and their corresponding role assignments;
5. We iterate over the list of sensitive assets collected in the previous subproblem , and for each of them, we map the principals and role assignments (at management group, subscription, and RG scopes);
6. We add the role assignment IDs of such principals to the LOCKSLAVE class. If it is NOT SATISFIABLE, it means that these principals may not only modify sensitive assets, but also unexpectedly perform lock operations, so **NO-BLUE-LOCKS** is False. We raise a **control anomaly**.
7. We iterate over the list of ReadOnly locks collected in the previous subproblem, and for each of them, we map the principals and role assignments (at management group, subscription, and RG scopes);
8. We add the role assignment IDs of such principals to the LOCKMASTER class. If it is NOT SATISFIABLE, it means that these principals may not only perform lock operations, but also unexpectedly modify sensitive assets, so **NO-GREEN-OPS** is False. We raise a **control anomaly**.



Description: Grants full access to manage all resources, including the ability to assign roles in Azure RBAC.

locks

Type : **Write**

● Actions ○ DataActions

Showing 2 of 12687 permissions

Type	Permissions	Description
Microsoft.Authorization		
Write	Add management locks ⓘ	Add locks at the specified scope.
Microsoft.Storage		
Write	-- ⓘ	--

Figure 5: manually checking whether a role has "Add locks" permission on a resource group.

Pseudocode

Initialize LOCKMASTER, LOCKSLAVE, LOCKED and UNLOCKED equivalence classes

```
from z3 import *


def addAsset(asset):
  global assets
  global AssetSort
  rasset=asset.replace("/","___")
  globals()[rasset]=None
  rasset=Const(rasset,AssetSort)
  assets[asset]=rasset
  return rasset



def addLockMaster(lmaster):
  global lockMasters
  global LockMasterSort
  rlmaster="___"+lmaster
  globals()[rlmaster]=None
  rlmaster=Const(rlmaster,LockMasterSort)
  lockMasters[lmaster]=rlmaster
  return rlmaster

LA=addAsset('LA')  # lockedRessources (LOCKED class)
UA=addAsset('UA')  # unlocked resources (UNLOCKED class)
LO=addLockMaster('LO')  # lock operator (LOCKMASTER class)
NLO=addLockMaster('NLO') # not lock operator (LOCKSLAVE class)

sol=Solver()
sol.add(LA!=UA)
sol.add(LO!=NLO)
```

Check whether a principal ID as a toxic combination of roles (simplified)

```
        if aSensitiveRG in scope:

          principals=resourceContainerPrincipals[aSensitiveRG]

          for aPrincipal in principals:

             if (aSensitiveRG in principals[aPrincipal]) and (aP in usersAndSpns):

               roleDefinitions=principals[aPrincipal][aSensitiveRG]

               for aRD in roleDefinitions:

                 rdID=aRD['roleDefinitionId']

                 addLockMaster(rdID)

                 sol.push()

                 sol.add(lockMasters[rdID]==NLO)

                 c=sol.check()

                 sol.pop()

                 if c==unsat:

                   print("incompatible principal ",aPrincipal,"has role",rdID,"which is a lock
master on RG",aSensitiveRG)
```

Using BitVectors to solve timing formula for unlocked sensitive resources

```
def solveTiming(unlockedTime):

  t=BitVec('t',32)

  expiry=14400  # 4 hours

  grace=1800  # 30 minutes

  now=int(time.time())

  if unlockedTime<0:

    return False

  if (unlockedTime>now):

    return False

  upperBound=expiry+grace+unlockedTime

  formula=And(ULE(t,upperBound),UGE(t,unlockedTime))

  formula=And(formula,UGE(t,now))

  sol=Solver()

  sol.add(formula)

  return sol.check()==sat:
```

References

[1]: *A new approach to Cloud security for native Cloud customers,* https://www.linkedin.com/pulse/new-approach-cloud-security-native-customers-christophe-parisel/

[2]: *Proving the Hub & Spoke pattern at scale,* https://www.linkedin.com/posts/parisel_proving-the-hub-spoke-pattern-in-aws-and-activity-6929324025382883328-kvZr

[3]: *A primer of automated reasoning for Cloud engineers,* https://www.linkedin.com/pulse/primer-automated-reasoning-cloud-engineers-christophe-parisel

[4]: Werner Vogels, *curious about automated reasoning,* https://www.allthingsdistributed.com/2022/03/curious-about-automated-reasoning.htm

[5]: Microsoft Research, *Programming Z3,* https://theory.stanford.edu/~nikolaj/programmingz3.html