A Foundation for Azure RBAC Security

Edition 1.0 (May 24, 2023)

Author: Christophe PARISEL

Abstract

In Azure, RBAC provides coarse and fine-grained access management for Azure resources, enabling users to have different permissions for different resources. However, with the increasing complexity and scale of cloud deployments, managing and verifying the correctness of such access control systems has become challenging.

This whitepaper proposes an approach to the formalization and reasoning about Azure RBAC fine-grained control plane permissions, leveraging congruence theory. By framing Azure RBAC permissions as a structured set, we define congruence relations that identify equivalent permissions. The congruence relation forms the basis for a logical model of Azure RBAC and serves as a tool for reasoning about control plane permissions and their interactions.

Introduction

We introduce a formal approach to reasoning about Azure RBAC permissions, utilizing the mathematical foundation of congruences theory. The congruence theory offers robust tools for reasoning about equivalence relations and operations that are compatible with these relations.

Our approach aims to bring several key benefits for Azure customers:

1. Simplification of the management of RBAC permissions by identifying and grouping equivalent permissions.
2. Enhanced understanding of the interaction between different permissions, which aids in maintaining least privilege access.
3. Improved predictability and verification of the behavior of RBAC permissions.
4. Increased IT security by ensuring the correct assignment and management of permissions.
5. Reinforcement of the reliability of RBAC permissions, ensuring that they behave as expected in different contexts.

Pre-requisites

The concepts of equivalence relations and congruences form the foundation of many branches of mathematics. They are tools that allow us to classify and reason about mathematical objects. In this section, we will introduce these concepts briefly and illustrate them with an example from number theory.

Equivalence Relations

An equivalence relation is a particular type of relation between objects that partitions a set into equivalence classes. In formal terms, a relation "~" on a set S is an equivalence relation if it satisfies three properties:

- **Reflexivity**: For every element a in S, a ~ a.
- **Symmetry**: For every pair of elements a and b in S, if a ~ b, then b ~ a.
- **Transitivity**: For any three elements a, b, and c in S, if a ~ b and b ~ c, then a ~ c.

The set of all elements related to a particular element forms an equivalence class. These classes partition the set: every element belongs to exactly one equivalence class, and each class has at least one element.

Congruences

A congruence is a type of equivalence relation compatible with all operations on the set: hence it is a very useful property.

More formally, a relation "~" on a set S equipped with an operation ∘ is a congruence if it is an equivalence relation and, for any a, b, c, d in S, if a ~ b and c ~ d, then a ∘ c ~ b ∘ d.

An example of a congruence relation in number theory is congruence modulo n. For two integers a and b, we say a is congruent to b modulo n, written a ≡ b (mod n), if n divides the difference a -

b. This relation is an equivalence relation and is compatible with both addition and multiplication, making it a congruence. This compatibility is the property that for all integers a, b, c, d, if $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$, then $a + c \equiv b + d \pmod{n}$ and $a * c \equiv b * d \pmod{n}$.

These mathematical concepts provide a powerful framework to structure and understand complex systems, such as Azure RBAC fine-grained permissions. In the following sections, we apply these concepts to formalize and reason about Azure RBAC control plane permissions.

Azure RBAC and formalization

The set of Azure Permissions

In Azure Role-Based Access Control (RBAC), permissions are integral for defining access to the control plane of Azure resources. Each permission is a string that denotes a specific action or a set of actions on a particular resource type. The set of all Azure control plane permissions, denoted by P, is a large and complex set that captures all possible actions on all available resource types in Azure.

A permission in P is a string of the form 'Microsoft.provider/resourceType/op', where 'Microsoft.provider' is the resource provider, 'resourceType' is the type of the resource, and 'op' is the operation on the resource (read, write, action, delete).

Classification of Permissions

Classes

In order to make use of the equivalence relation, we first classify permissions into operation classes. This is achieved by examining the operation token in the permission string, i.e., the specific operation on the resource.

- **Read Class**: This class includes permissions where the operation is 'read' or ends with '/read'. For example, the permission 'Microsoft.Web/serverfarms/read' belongs to the read class. Furthermore, if the last segment of the permission string is a wildcard "*", but the segment before it indicates a 'read' op (for instance, 'Microsoft.Web/serverfarms//read'), it still belongs to the Read class as the wildcard does not change the nature of the permission.
- **Write/Delete Class**: This class is for permissions where the operation is 'write', 'delete', ends with '/write' or '/delete', or is ''. A wildcard "*" alone without a preceding 'read' segment would indicate a broader permission scope and is classified under write/delete.
- **Action Class**: This class includes permissions where the operation is 'action' or ends with '/action'. For example, the permission 'Microsoft.Web/serverfarms/join/action' belongs to the action class.

These operation classes provide us with a way to reduce the complexity of the set of permissions by grouping them into broader categories. This categorization is at the core of defining the equivalence relation and the subsequent congruence.

Please note that the classification is hierarchical with Write/Delete taking precedence over Action, which in turn takes precedence over Read. For instance, if a permission string contains both 'read' and 'delete', the permission would be classified as Write/Delete.

The union-find data structure

To utilize operation classes efficiently, we employ the Union-Find data structure. The Union-Find data structure is a fundamental tool used in computer science to keep track of a partition of a set into disjoint subsets. It provides two primary methods:

- **Find**: Determine which subset a particular element is in. This method can be used for determining if two elements are in the same subset.
- **Union**: Join two subsets into a single subset.

In our context, the 'Find' method helps us identify the operation class of a permission. The 'Union' method allows us to merge permissions into the same class if they are equivalent (belong to the same class).

The efficiency of the Union-Find algorithm is enhanced by using a technique called **Union by Rank**. The rank of a subset represents an upper bound for the height of the trees representing the subsets. When two subsets are unioned, the one with the lower rank is attached to the root of the one with a higher rank. This technique helps keep the tree balanced, thus improving the efficiency of the 'Find' method.

By combining the classification of permissions, the equivalence relation, the Union-Find algorithm, and Union by Rank, we are able to efficiently partition the set of permissions into equivalent classes of actions.

With this classification at hand, we can now define the equivalence relation based on the class of the operation.

Equivalence Relation

We define an equivalence relation "~" on the set P based on a particular property – the class of the operation. That is, two permissions are equivalent if they belong to the same class (read, write/delete, or action).

We can show that "~" is an equivalence relation by proving that it satisfies reflexivity, symmetry, and transitivity:

- **Reflexivity**: Every permission is equivalent to itself because it belongs to the action class as itself.
- **Symmetry**: If permission p1 is equivalent to p2 because they belong to the same operation class, then p2 is also equivalent to p1 because the class of p2 is the same as that of p1.
- **Transitivity**: If permission p1 is equivalent to p2 and p2 is equivalent to p3 (because they all belong to the same class), then p1 is also equivalent to p3 because it belongs to the same class as p3.

Congruence

As we have partitioned the set of Azure RBAC permissions into classes using the Union-Find data structure, we now introduce a specific property to facilitate easier management and analysis of permissions: the **congruence property**.

The congruence property is based on the **combine** operation, a binary operation that takes two permissions and merges them. The result is a permission which has the 'highest' privilege of the two input permissions.

In terms of our defined classes, 'Write/Delete' is considered the highest privilege, followed by 'Action', and lastly 'Read'.

Here is a sample implementation of *combine* in Python:

```python
def combine(permission1, permission2):
    # Order of precedence: write/delete > action > read > unknown
    precedence = {'unknown': 0, 'read': 1, 'action': 2, 'write/delete': 3}

    class1 = classify_permission(permission1)
    class2 = classify_permission(permission2)

    # Return the permission with the highest precedence
    if precedence[class1] > precedence[class2]:
        return permission1
    else:
        return permission2
```

The *classify_permission* function called within *combine* is defined as such:

```python
def classify_permission(permission):
    if permission == "*" or permission.endswith("/*"):
        return "write/delete"

    segments = permission.lower().split("/")

    if any("write" in segment or "delete" in segment for segment in segments):
        return "write/delete"

    if any("action" in segment for segment in segments):
        return "action"

    if any("read" in segment for segment in segments):
        return "read"

    return "unknown"
```

We claim that the *combine* property is congruent with respect to our equivalence relation. In mathematical terms, this means:

For any permissions p1, p2, q1, q2,

if p1 is equivalent to p2 and q1 is equivalent to q2 (based on our defined equivalence relation),

then the results of combining p1 with q1 and p2 with q2 are equivalent.

This congruence property has substantial benefits. It allows us to 'merge' permissions while respecting the classes we have defined. For instance, if we combine a 'Read' permission with a 'Write/Delete' permission, we know the result belongs to the 'Write/Delete' class regardless of which specific permissions we started with, as long as they were in the appropriate classes. This knowledge greatly simplifies the analysis and manipulation of permissions.

Proving Congruence of the Combine Operation

To demonstrate that the **combine** operation holds the congruence property, we need to show that for any permissions p1, p2, q1, q2, if p1 is equivalent to p2 and q1 is equivalent to q2 (based on our defined equivalence relation), then the results of combining p1 with q1 and p2 with q2 are equivalent.

Claim

Let p1, p2, q1, q2 be Azure RBAC permissions such that p1 is equivalent to p2 and q1 is equivalent to q2. If we combine p1 with q1 and p2 with q2, the results are equivalent.

Proof

By the definition of our equivalence relation, permissions p1 and p2 belong to the same class. Similarly, q1 and q2 belong to the same class.

Let's denote the classes of p1, p2 as [p] and the classes of q1, q2 as [q]. Then, due to the rules of the **combine** operation:

combine(p1, q1) = [p] if [p] > [q]

combine(p1, q1) = [q] if [q] >= [p]

And similarly,

combine(p2, q2) = [p] if [p] > [q]

combine(p2, q2) = [q] if [q] >= [p]

Since [p] and [q] are the same classes for both operations, it follows that combine(p1, q1) and combine(p2, q2) belong to the same class and hence are equivalent.

Therefore, our **combine** operation holds the congruence property with respect to our defined equivalence relation on Azure RBAC permissions.

Practical use cases

Proving custom IAM statements in AAD tenants

The prime use case for our equivalence relation is **automated reasoning**: we can formally verify whether a principal (group, user, app)'s permission belongs to the expected equivalence class for any given scope, in an unsupervised way.

The partitioning of the set of Azure control plane permissions into only three equivalences classes may look very coarse, yet it enables to verify whether powerful theorems holds in a customer tenant.

For instance, we may define an **uninterpreted function** called isWriteable(principal,subscription) and verify the satisfiability of the following statement:

Implies(isWritable(AliceId,sub1),And(Not(isWritable(AliceId,sub2),sub1 != sub2))

The consistency and completeness of isWritable is a direct consequence of the fact that combine is an equivalence relation.

Reasoning across multiple roles or groups

In the context of Azure RBAC and permission systems in general, two operations reflecting common tasks in managing permissions in RBAC systems compatible with a congruence relation could be:

1. **Union:** This operation could represent the combining of permissions from multiple roles or groups. For example, if a user is part of two roles, each with a set of permissions, the total set of permissions for the user would be the union of these two sets. Since combine() is a congruence, the union operation preserves the equivalence classes.
2. **Intersection:** This operation could represent finding the common permissions between two roles or groups. For example, to find permissions that are common in two different roles, you could use the intersection of the two sets of permissions. Again, since combine() is a congruence, the intersection operation preserves the equivalence classes.

As an example, let's assume we have four permissions: p1, p2, q1, q2.

If combine(p1, p2) and combine(q1, q2) are in the same equivalence class, then intersect(p1, p2) and intersect(q1, q2) should also be in the same equivalence class for our congruence to hold.

To be more specific, the following statement should be true for all p1, p2, q1, q2:

If combine(p1, p2) ~ combine(q1, q2) then intersect(p1, p2) ~ intersect(q1, q2)

Here '~' denotes that the two permissions are in the same equivalence class.

It's crucial to ensure that the semantics of the operations (union, intersection) align with the intended semantics of the permission system. For example, unioning permissions should generally increase access, while intersecting should reduce access to the common elements only.

Opportunities

Three other operations could bring interesting long-term opportunities, depending how and what Microsoft will make of its RBAC permissions system:

1. **Composition:** This operation could represent the chaining or sequencing of permissions. This would be especially relevant in systems where the order of operations matters, such as in a workflow where certain actions must be performed before others.
2. **Expansion:** This operation could represent the interpretation of wildcard or other shorthand notation in permissions. For example, if you have a permission that grants read access to all resources (**'*/read'**), the expansion operation would translate this into a set of read permissions for each individual resource.
3. **Contraction:** The inverse of expansion, this operation would replace a set of permissions with a shorthand notation if possible. For example, if a user has read permissions on every individual resource, the contraction operation could replace this with a single permission granting read access to all resources (**'*/read'**).

Conclusion

This document has presented an analytical approach to understanding and classifying Azure RBAC permissions using concepts from set theory, namely equivalence relations and congruences. The principle driving this analysis is to improve understanding of Azure RBAC permissions and potentially provide a systematic way to classify and reason about them.

Through our analysis, we identified a property, *combine*, which was found to be an equivalence relation over the set of Azure control plane permissions. Although combined partitioned this set into just three equivalence classes (Write, Read and Action), it offered enough expressiveness to formally verify powerful statements holding within a customer Tenant.

What's more, combine was found to be a congruence with respect to common IAM operations. Specifically, we showed that operations like composition and intersection preserve the combine property, leading to consistency and predictability when these operations are applied.

This is of significance because, if combine were merely an equivalence relation without being a congruence, we wouldn't be able to guarantee that these operations preserve the equivalence classes defined by combine. Consequently, applying these operations could result in unpredictable and inconsistent outcomes. This means that reasoning about permissions and predicting the effects of these operations becomes more reliable.

While our approach focuses on Azure RBAC permissions, the concepts applied here are not specific to Azure or RBAC systems. Therefore, it is plausible that similar approaches could be adopted in other settings to improve understanding and management of permissions or other types of classified data.

Future work may also consider exploring other congruences or properties that may be relevant to classifying permissions.

Customer IAM security in Azure

It's worth noting that while the congruence relation offers a mathematical foundation for equivalence and consistency, it doesn't replace the need for a well-thought-out permission system design that suits your specific needs and requirements. Mathematical relations should serve to simplify and ensure consistency, but shouldn't overly simplify permissions to the point of reducing security or functionality.

Azure RBAC holds thousands of permissions. What's more, they are subjected to change without notice: this is especially true since customers are not expected to manipulate permissions and reason about them.

The semantics of permissions used by Microsoft might be not fully consistent across the whole scope of Azure RBAC. If this was the case, it could have an impact on the way we classify permissions.