The Abacus: A New Architecture for Policy-based Authorization

Jacob A. J. Siebach
Brigham Young University
jacob_siebach@byu.edu

Justin Scott Giboney
Brigham Young University
justin_giboney@byu.edu

Abstract

Modern authorization architectures using role-based, policy-based, and even custom solutions have numerous flaws and challenges. A new design for authorization architecture is presented called the Abacus. This paper discusses the architecture that the Abacus utilizes to overcome the issues inherent in other proprietary and open-source authorization solutions. Specifically, the Abacus respects domain boundaries, is less complex than existing systems, and does not require direct connections to domain data stores.

1. Introduction

Said Eric Evans, "Every software program relates to some activity or interest of its user. That subject area to which the user applies the program is the domain of the software" [1]. Domains are areas that control a specific set of data for an organization, e.g. HR, engineering, or customer support. Domains are at the heart of every computer system, storing data and enabling the business functions of the organization.

Every existing computer system has rules governing who is allowed to perform certain tasks or view specific data within that system, even if the rule is that anyone with access to the device is allowed to use it. These rules are called *authorization policies*. Domains use policies to safeguard the data within them. Numerous commercial and custom systems in the world today use roles and groups to control authorization, but these have proved to lack the fine-grained control needed, are prone to role explosion [2], and are often difficult to keep in sync with who should be allowed to have access [3]. In the past twenty years, several enterprise systems have been created to allow organizations to control authorization via authorization policies that rely on data attributes instead of roles or groups. While this reliance on data attributes allows for fine-grained authorization, one problem of many modern systems is the method of attribute gathering.

For a policy to grant authorization, the system using the policy needs access to the attributes of the user requesting authorization. Many current systems get these attributes by directly accessing the database tables where the attributes are stored. While this access method may allow the authorization system to get the current value of the attribute at run-time, it poses numerous security and domain-boundary issues, among which are tight coupling of the authorization system to the domains, the ability for a malicious actor to utilize the authorization system as a pivot into production databases, and increased authorization latency.

We propose that authorization gathering should not be a function of the authorization system, but that the attributes should be pushed to the authorization system from the source domains. In storing the attributes as they are pushed to the authorization system, checks for authorization never require external calls (which decreases latency), nor does the authorization system require direct pipes to domain data stores (increasing security and decreasing database load). We achieve this goal by reviewing current literature and commercial systems, identifying the strengths and weaknesses of current technology, and providing a case study of the implementation of the new system at a large U.S. university.

In the past two and a half years we have created and implemented a new authorization system that addresses the issues above and becomes a faster, more secure, and more architecturally-sound solution than the other options in the authorization space. We have found that it is possible to completely decouple the authorization system from domain databases, allowing the domains to truly own the attributes that they own. This paper introduces this new solution.

2. Background

2.1. Identity and Access Management

To comprehend the problem space of authorization, it is vital to recognize the distinction between the four components of Identity and Access Management (IAM): identity management, authentication, access management, and authorization. While many domain models conflate these components, decoupling the functions allows us to investigate authorization without focusing on the issues present in the other IAM pieces.

According to Recordon and Reed [4], *identity management* consists of the use of identifiers and

URI: https://hdl.handle.net/10125/71469
978-0-9981331-4-0

personally-identifiable information. An *identity* consists of a minimal amount of data, possibly including keys, certificates, or tokens [5], used to distinguish one user from another. Identity keys, such as name, address, membership number, exist in a system to allow humans to make sense of the identities stored within the system. Entities that may have an identity include users of the system, organizations, computer applications, and physical devices.

The process of confirming that a person or system is who they profess to be is called *authentication*. It is easy in a digital world to pretend to be someone or something else (as evidenced by a popular cartoon, On the Internet, nobody knows you're a dog) [6], but through verification of credentials and other methods [7] a system can be more certain that the entity being communicated with is indeed who they say that they are.

Certain attributes are maintained by the pillar of *access management*. This includes role and group management systems. While the aim of policy-based access control (PBAC) is to reduce the requirement for roles and groups, legacy systems often rely on roles and groups. Manual designations are most easily designated by adding them to a group or giving them a specific role.

With *authorization* a user has the permission to perform the actions desired. Depending on the system, authorization is a result of arbitrary designation by the business, inherent from a position, or granted by delegation. It is herein proposed that authorization should be granted through policies that rely on attributes, with a specific architecture to enable such.

2.2. Individual Authorization and Access Control Lists

When computers were first invented, a user could type the command for a program and it would run. As systems began to allow multiple users to access them, authorization was required to prevent unauthorized access to specific data. Oft times a table with a list of users authorized for a specific program was kept, and if the user was in the table, then they could run the program. Sometimes an Access Control List (ACL) only allowed communication from specific IP addresses to protect access.

While ACLs and individual authorization were good as an initial step, they had their problems. To begin with, every person or system address had to be individually added to the access table. This required manual effort on the part of the administrator, and if they were not in the office when a new user wanted access, then the new user may have to wait for hours or days before receiving access. This also presented a problem in the opposite direction: it was also common for a user to still have authorization when they left an organization because the administrator forgot to remove them from the list, either by oversight or from being uniformed about the departure.

2.3. Roles and Groups

The next type of authorization came in the form of roles and groups. A role is like a permission for a specific task or function, and a group was simply a group of people in a list. Functionally equivalent, a system would check if the user had a specific role or a was in a certain group to grant authorization to the user. This meant that program code could specify a role or group instead of looking for a certain user in a table, but the manual challenges of adding or removing roles/groups from the user remained.

Roles and groups have been the de-facto standard for decades. Almost every enterprise resource planning (ERP) system in existence uses roles and groups to administer access and grant authorization to program functions and data. Numerous commercial [8] and open-source solutions [9] have been developed to manage roles and groups.

2.4. Attribute and Policy-based Systems

While the idea for authorization systems relying on policies and attributes has been around for decades, the real effort in this area did not begin until the turn of the millennium. Attribute-based Access Control (ABAC) and PBAC serve to provide an authorization decision by utilizing an authorization engine that is separate from the system that the user is attempting to access. This engine is commonly called the Policy Decision Point (PDP). Other common components of authorization systems include the Policy Administration Point (PAP) which allows domain owners to create policies, and Policy Information Points (PIPs) that are responsible for gathering attributes [10].

There are many advantages to using attributes and policies over roles and groups, to the point that research has been conducted to see if ABAC policies can be used within a Role-Based Access Control (RBAC) framework [11]. The benefits of PBAC including the ability to know which systems use which attributes, ease of auditing, enabling systems to use attributes from other domains in their policies, and separation of authorization logic from business logic. It is unknown to the authors of any commercial systems that utilize this methodology inherently, yet there are several companies that offer ABAC/PBAC services to organizations [12]–[14]. While these systems offer a simpler way of checking authoring and evaluating authorization logic, most modern architectures do not respect domain boundaries and suffer from inherent latency issues.

2.5. Domain Authorization Through Authorization Policies

A well-implemented domain consists of several parts: a central data store (CDS) that holds the relevant domain data, events that are raised as certain business processes occur, application programmable interfaces (APIs) that enable other systems to interact with the domain, data retrieval integration protocols (DRIP) for data lakes that enable metrics, and other such features. In an idea promoted as "Hexagonal Architecture", Cockburn says that there should be no "...infiltration of business logic into the user interface code" [15] and that the APIs should make available the business functions of the domain. Vernon states that domain models should be "technology-free" [16] and not contain implementation-specific details. In this way "the data model should be subordinate" [17], meaning that the domain model should care about the business of the domain and not the data model that becomes the implementation of the business model.

We support Cockburn's assertion that domain business logic should be controlled from inside of the domain and Vernon's statements regarding domain models focused on the business processes. We seek to expand upon these ideas with what we call General Moore's Medallion, named after Brent Moore, Chief Solutions Architect at Brigham Young University. In Figure 1 we see that the core of a domain is surrounded by authorization policies, and these policies protect access to the domain components. For example, when an API is queried, the authorization policies for that API should be evaluated to see if the calling user/system is authorized for the data. Similarly, if an event is raised by the domain, policies should govern what subscribers are authorized by the domain to receive the event. This pattern should persist in all accesses to the domain.

Using policies to govern authorization provide several advantages over previous authorization methodologies [18]. Fine-grained control is possible with policies, and access can be based on dynamic properties such as time of day, calling client system, user employment status, or other volatile factors—things that are not necessarily available to ACLs or role-based systems.

This paper will enumerate the advantages of attributes and policies in authorization. It will then evaluate the difficulties of current implementations and provide solutions using a new methodology contained in a technology that we call the Abacus.

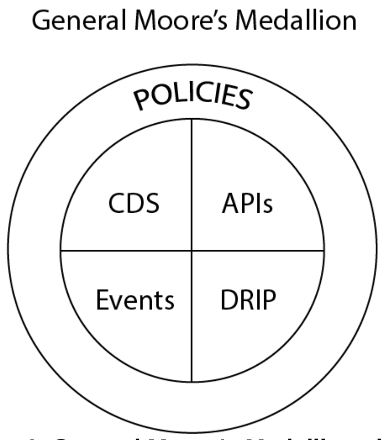

Figure 1. General Moore's Medallion shows that authorization policies should protect every aspect of a domain.

3. Issues with Existing Technologies

While developing code to give access to a user with a specific role was a large step forward, decades of this method of authorization has revealed massive issues. The problems range from maintenance to authorization granularity to data leakage will now be enumerated and explained.

3.1. Role Explosion

Over time it was discovered that role-based systems suffered from role explosion [2], a phenomenon resulting from authorization requiring a granularity that is not available with roles. With attributes it is easy to create a policy that requires specific attribute values, but a system that can only check for a single role suffers from an issue of combinatorics. If there are three attributes, each with three possible values, then a total of sixty-three roles would need to be created to express every combination of these three attributes together: nine individual roles (one for each attribute value), plus another twenty-seven roles (for each combination of only two attribute values), plus another twenty-seven roles (for the possibilities of all three attributes).

For example, a system might allow a user to access it if the user is a faculty member. The role of "Faculty" may be given to the user, and this will enable them to access the said system. The problem begins when such a wide role is not enough; additional roles of “Tenured Faculty” and “Associate Faculty” might be created for more specific system functionality. When the distinction of “Research Faculty” or “Teaching Faculty” arises, the number of roles increases. The number of roles continues to grow over time, requiring increasingly precise niches, such as “Research Faculty with Federal Grant” or “Research Faculty without Funding”.

Eventually, systems with roles required hundreds or thousands of roles to express what was going on for individual functions in systems. At this point, when a new system would be developed, instead of just finding a role that matched the need, the developer would just create a new role, to avoid spending exorbitant amounts of time looking through the existing roles. Similarly, role explosion makes maintenance completely untenable for an administrator of a role management system. With thousands of roles, it is extremely time-consuming to find roles that match the required functionality of a new system, or to find which roles are no longer in use.

3.2. Data Leakage

Another problem with roles and groups is the issue of data leakage. If a person has the permission to view the roles and groups of an individual user, then they can learn confidential things about that user. For instance, if the user has a role called “Six-figure salary”, then it can be determined that the user makes a lot of money. If the group “Sexual harassment victim” is present, then someone can know information that should not be made available. Such data leakage can have extremely detrimental effects on individuals and on an organization, not to mention legal ramifications.

3.3. Usage Invisibility

The problem with a role/group governance system is the lack of knowing what other systems utilize which roles or groups. If Program A requires Role B, then Program A will ask the role manager if the current user has Role B in order to authorize the use for their current operation. While this is useful to Program A, the role manager has no inherent way to know that Program A uses Role B. This presents a massive issue when transitioning from old systems to new ones, as system administrators do not know which applications need to be updated.

To illustrate this fact, one organization had the following experience. Years ago, the Federal Government of the United States changed the definition of part-time employee. The organization had certain groups that signified the description for the old part-time status, and they were forced to create new groups based on the new government definitions. There was no way to tell from the group database who was using which groups, nor was there any way to know from the LDAP directories that housed copies of these groups. The only way to know was to look at the code for every system in the organization.

The engineer tasked with this change had an idea: every Monday morning he would remove the old part-time groups from the database and LDAP directories. He would then wait until a couple of departments had called to complain that their systems were no longer working. He would then restore the groups and spend the remainder of the week moving those departments to the new part-time groups. Come Monday morning he would repeat the process and work with the new callers to fix the groups used by their departments’ code. Sometimes a department would call multiple times, because their authorization logic was in multiple segments of code. It was six months before all the organization had been moved to the new part-time groups.

From such a painful example it is easy to see the benefit that comes from being able to quickly query what systems utilize which attributes.

4. Problems with Modern ABAC/PBAC Architectures

There are numerous architectural considerations with modern commercial authorization system that arise out of the architectural model of said systems. The central design of these systems is to host a decision engine, the PDP, that is responsible for calculating an authorization decision for an identity. When the PDP is queried, the common method is for the PDP to call a PIP to gather the attributes in real-time from the domains that own the attributes. The PIPs are often “connectors” that go straight to the central data store (CDS) of the domain and directly retrieve the attribute value from a database table. It is this method of attribute gathering that causes significant issues with security, latency, and maintainability. We will address several issues with the current methodologies before proposing our solutions in this section.

4.1. Current Authorization Flow

For many instances, the process to get data from a domain begins with a user or other system that makes a request to the domain (see Figure 2). The domain verifies that the caller is approved for such data, then returns it. Several modern authorization systems modify this by placing a Policy Enforcement Point (PEP) before the domain. The PEP is responsible for calling the Policy Decision Point (PDP) which calculates the authorization decision, and if approved, the PEP passes the calling request to the domain. The domain gets the data and returns it to the PEP. The PEP may then filter the data, based on the authorization policies, before returning it to the caller. Here is the normal flow of modern systems:

1. A user or system requests access to a resource.
2. The PEP takes the request, determines who/what is making the call, and sends a request to the PDP for authorization.
3. PIPs request data from other domain stores.
4. Attribute data is returned to the PDP.
5. The PDP calculates the authorization and returns a response to the PEP.
6. a. If the result is “Deny”, then the PEP is directed to return a “Not Authorized” message to the caller.
   b. If the result is “Permit”, then the request is forwarded to the domain.
7. The domain checks the business rules to see if it should send an error or the requested resource.
8. If the business rules check out, the domain queries its CDS to get the data.
9. The CDS returns the relevant data to the domain.
10. The domain returns the data to the PEP.
11. The PEP may filter the data based on various authorization configurations.
12. The PEP returns the authorized data to the caller.

This model requiring a PEP has several disadvantages: increased cost, increased latency, conflated authorization and business logic, connectivity configuration complexity, and endpoint configuration complexity.

First, the greater the number of components required for authorization, the greater the cost. Both the PEP and the PDP have a cost to install, configure, and run. Both must be operative for this model to work, requiring additional server allocation and running expense.

Second, if the domain were calculating authorization on its own, it would only require the network hops to get the attributes needed for its decision. With a PEP in place, the number of network hops is reduced for authorization, only to replace it with four more hops: going to the PEP, going to the PDP, returning from the PDP to the PEP, and from the PEP back to the caller. Additionally, the PEP can become a network bottleneck if there are a significant number of requests going to various domains that must all be sent via the PEP.

Third, since all traffic must pass through the PEP, the PEP must be configured to know the location of every system that it may stand as the guardian for. This requires significant operational resources to make sure that any change in domain location is accurately updated within the PEP. This places increased demand on DevOps teams to ensure that nothing in a domain change has broken the ability for extant entities to contact it.

Finally, being a gatekeeper, the PEP must know every endpoint, protocol type, and available contact methods for the domains that it is protecting. Setting this configuration is well beyond the realm of the domain’s business owners and falls squarely into the hands of IT. By placing this burden on IT, the business owner is further removed from the ability to easily change things, should they require it.

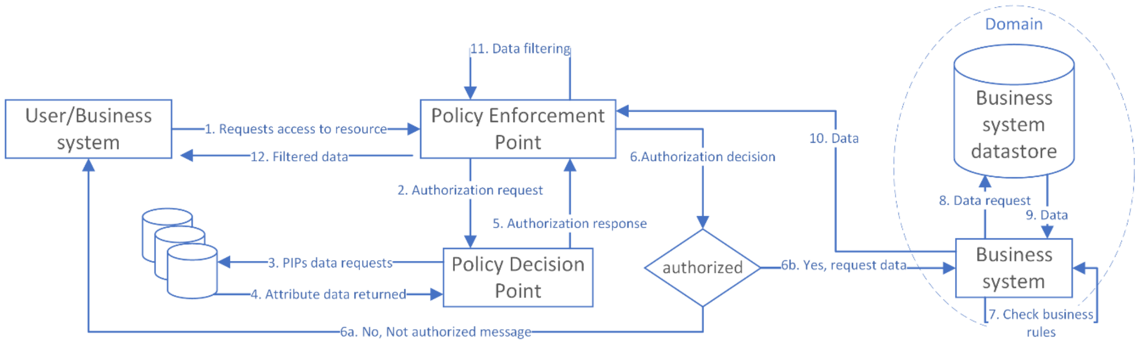

Figure 2. Existing authorization architecture

4.2. Breaking Domain Boundaries

The core tenants of Domain-Driven Design (DDD) espouse that the business processes of the domain govern the access and use of the data within the domain. When any external entity connects to the CDS of a domain, all data governance is lost, for the domain has no way to mitigate access to the data. The domain itself should maintain the protection of data from external sources, making that data only available via APIs, events, claims, and so forth, each validating requests through domain-controlled policies.

Letting users and systems connect directly to data stores without their domain creates a major security risk. The more connections that come into a database, the more likely it is that one of those users will be compromised at some point, allowing access.

4.3. Attribute Gathering

All retrieval of attributes in any authorization methodology must be fast. While there are numerous ways to accomplish this, each has limitations. Possible options include the run-time retrieval of attributes, caching attributes within the memory space of the decision engine itself, utilizing an external cache with complex logic to determine when to expire attributes and when to refresh them from the central data stores, or employing a persistent cache that always contains the full set of existing attributes. We will now evaluate each option to provide its drawbacks and advantages.

It is possible to keep cached data within the memory space of the engine, but this requires the implementation of the cache code within the engine, adding complexity. It also requires that the system running the engine use larger and larger amounts of memory as the number of attributes in the cache grows.

Run-time retrieval of attributes is the best way to ensure that the attribute is accurate at the time of the request. The issues that arise in this situation are those of latency and domain resiliency. For the engine to make a request out to the domain with attributes, PIP must either go directly against the data source (which has been addressed before), or they must query an intermediary system. To query an API or data system requires additional time and configuration. If a domain goes down, then all authorization dependent upon the attributes in that domain will no longer work until the domain is back online.

The problems of complexity, latency, and possible outages can be avoided if the system is correctly designed. We will present the proper architecture next.

5. The Abacus: A New Architecture for Authorization

The Abacus is a policy-based authorization management system (see Figure 3). It sits inside of the same network as the business systems and the only systems that are inside this network can invoke the Abacus. The data flow of Abacus consists of 7 steps:

1. A user or system requests access to a resource.
2. The domain takes the request, determines who/what is making the call, and sends a request to the Abacus for authorization.
3. The Abacus calculates the authorization and returns a response to the domain.
4.
   a. If the result is “Deny”, then the domain returns a “Not Authorized” message to the caller.
   b. If the result is “Permit”, then the domain checks the business rules to see if it should send an error or the requested resource.
5. If the business rules check out, the domain queries its CDS, with whatever business rules it requires, to get the data.
6. The CDS returns the relevant data to the domain.
7. The domain returns the data to the caller.

Compared to existing models, this architecture provides several advantages: reduced cost, decreased latency, separation of authorization and business logic, simplified connectivity configuration, simplified endpoint configuration. We will expound upon these further in the next major section.

As mentioned previously in this paper, existing authorization systems require either 1) a connection to domain data for run-time retrieval, or 2) that all information required to calculate a decision is passed in the request to the PDP. The Abacus solves the challenges presented through these methodologies by utilizing a persistent cache of all attributes that the policies need. The cache is kept current through the updates of the domains that own the attributes: when an attribute that the domain controls changes for a user, that change is then pushed to the Abacus via a simple API.

5.1. Defining Policies

All PBAC systems define policies, as does the Abacus, but the method at which those policies are combined to define authorization for resources differs. While other systems require an administrator to define a resource, then define the actions available to a resource, the Abacus simplifies this process into one step: a policy set is defined as an action on a resource. By so doing, the policy set becomes technology-agnostic. Many systems today promote REST methods for web contexts. Should a new protocol come around, all of the existing systems will need to change, but the Abacus can continue unhindered.

For example, an authorization system may define resource A as an HTTPS endpoint representing certain records for the domain, and then it configures GET, POST, and DELETE methods. If a user invokes the GET method to view records, the PDP will evaluate the policy set for that method on that resource. But if in the future the technology moves away from REST, then the whole of this configuration must be redone.

In contrast, the Abacus would define a policy set of “ViewResourceA”, and then the calling API can interpret the technology that it uses (which may be a GET method) to request “ViewResourceA”. In the future, if the domain changes to a new technology (which does not use “GET”), the *action on the resource does not change*, and the domain still calls for “ViewResourceA”, no matter how the technology of the API is administered. This forward-compatible architecture further decouples authorization from the domains.

5.2. Persistent Cache

The most efficient way to return information from a data store is to keep that data in memory. Thus, the most effective model for attribute retrieval involves the use of a persistent cache, one that never expires rows and contains a complete copy of all attributes needed for authorization policies. This Attribute Cache is located as close to the decision engine as possible to reduce the latency between the two. It has the advantage in providing the engine with all available attributes as needed—even if the domain that owns a set of required attributes goes off-line. In this way, authorization can continue even as domains are serviced for maintenance or become inaccessible in unforeseen instances.

Only attributes that exist should be placed in the cache, to wit, only attributes that will cause a policy to evaluate to “true”. For instance, if a policy requires that the user be an employee to access a specific resource, then the system should only store the employee attribute for those that work for the organization. There may be thousands, or millions, of other IDs within the system that the Attribute Cache contains attributes for who are not employees: clients, customers, devices, etc. Restricting the cache to contain only attributes that exist for an ID allows for data reduction in the cache size by orders of magnitude. Another added benefit is simplification of the decision engine logic: if an attribute is not present in the cache for an ID, then that entity does not have the attribute and processing will respond appropriately.

5.3. Attribute Database

While a cache is excellent for performant data retrieval during decision requests, it does not provide a permanent store for the collection of attributes known to the system. It is possible to replicate the cache database, but there are not many tools (if any) for this. By design, the cache is kept with as little information within it to make it lightweight and fast. No data about who or what added the attribute, when it occurred, or what the definition of the attribute even is, exists in the cache. There must be Attribute Database to maintain the master record of the attributes stored in the engine which keeps these points of data.

The Attribute Database should contain the expected items for attributes, such as the attribute type, value, and the ID that has the attribute, but it should also maintain the ID of the user or system that added the attribute,

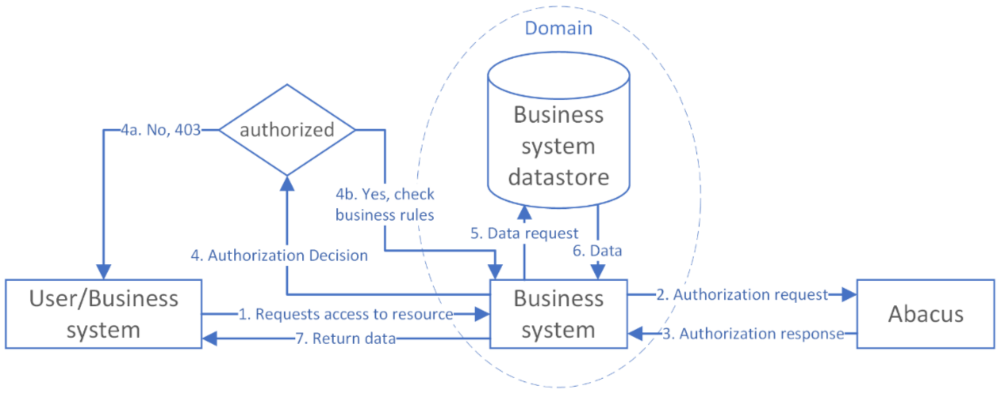
Domain
Business system datastore
authorized
4a. No, 403
4b. Yes, check business rules
5. Data request
6. Data
4. Authorization Decision
User/Business system
Business system
Abacus
1. Requests access to resource
7. Return data
2. Authorization request
3. Authorization response

Figure 3. The Abacus

timestamp of when the attribute was added, and other such things that are too bulky to be stored in the Attribute Cache.

A persistent store of all known attributes provides several advantages to the decision engine, though it never directly interacts with the engine. When a new Attribute Cache is initialized, it can pull the list of attributes from the Attribute Database and be operative in a matter of seconds. This also enables the administrators to remove all data from the cache and quickly restore the full record if there are ever concerns about data integrity. Backup and recovery of the Attribute Database is easy with tools that exist for whatever platform the organization chooses to work with.

5.4. Attribute Updates

When using a cache that does not expire rows, the natural question of data integrity arises. What happens when an attribute changes in a domain? There must be a system that allows attributes in the cache to be modified. For the Abacus, we take the complexity of cache ejections, refreshes, and so forth away from the cache and put it in the hands of the domain, where it rightly belongs. The system has an API that allows domains to push attributes changes (that they control) into the Abacus. Since the domain alone knows when an attribute within it changes, and the domain knows the components of data that make up said attribute, then the domain can easily call the Attribute Update API in the Abacus to add or delete an attribute for a user.

The method of updating attribute in the Abacus can be accomplished in several ways. We suggest that the best solution is for the domain to raise events when the governed attributes change, and to also write event consumers that translate those events into attributes that are then pushed to the authorization engine. By so doing, these attribute-updating consumers can be kept in one place and easily reviewed. Additionally, all the implementation details of the domain are abstracted from the consumers and code can be focused on only pushing attributes as they are defined by the business logic. Should any of the underlying infrastructure of the domain change, be it database, API code, or otherwise, the event consumer will not need to be rewritten and authorization updates can continue without hindrance.

6. Advantages of the Abacus

Having explained the architecture of the Abacus, we will now enumerate upon the advantages that this new design gives over the existing systems, and we will show how it solves the problems of previous authorization methodologies.

6.1 Domains Define Attributes

All business logic for a domain should be contained and maintained within the domain. Because the Abacus specifically requires the domains to push attributes into it, the business definition of the attribute is maintained by the domain itself. We suggest that all domains record the definitions of their attributes in a central tool so that any other domain within the organization can understand the attributes as well.

By allowing domains to truly control their attributes, the authorization system no longer needs to completely understand the domain business in order to do its job. This allows the authorization administrators to focus on other tasks, plus it allows those from other domains to learn the ubiquitous language of the domain in question (and vice versa).

The domain also knows best about how and what it should filter before returning data to a user. Removing the PEP allows the domain to fully perform its primary functions (including data filtration). Instead of requiring the business logic to be placed inside of the authorization component, this architecture allows for good microservice design, letting the authorization be a complete package in its sphere while the domain handles the business filtering that it understands inherently. This both gives the domain control over data and simplifies the authorization process as well.

6.2 Respected Domain Boundaries

The web of interconnected, tightly-coupled domains goes away with the design of the Abacus. DDD principles are respected when *only* the domain has access to its data stores. No longer are other systems reaching into the domain database, and the domain is free to change the underlying structure as it sees fit, per its business needs, without the threat of breaking other systems.

This massive decoupling allows each domain to operate effectively as a microservice. The business owners can define the domain logic while the developers can implement each component completely independent of other systems, and authorization can be provided as an external service that places no load on the domain itself.

As domains push their attributes into the Abacus, both systems are utilizing a well-defined contract, and the authorization system itself becomes a domain that also has *its* boundaries respected.

6.3 Efficiency

Using the domain as the PEP itself provides financial and chronotical advantages. If the domain enforces the decision of the authorization engine, it removes the cost of running a PEP server. Besides the hardware and electricity cost, there is also the reduced work of the professionals that would have had to configure the PEP.

From a latency perspective, removing the PEP and PIP calls to domain stores eliminates at least six extra network hops. This in addition to the time spent in the PEP filtering data, allows the Abacus to provide a more efficient response.

Additionally, because the Abacus is constructed with a policy grammar specifically designed for its architecture, it is more efficient than other systems (we will be writing another paper in the future to discuss this in depth).

6.4 Simplified Configuration

Modern authorization architectures often require specific configuration of domain endpoints, including endpoint address, method types, authentication tokens, etc. Such technical specificity means that the IT staff must be the ones to define the interactions with the PDP. By design, the Abacus only defines *actions on resources*, which removes the need for specific connections to other systems. This simplifies configuration, as the authorization engine does not need to have any explicit connections to other domains defined within it. A domain may change its endpoints, but since the Abacus defines its policy sets as *actions on a resource*, no reconfiguration is needed.

6.5 Data Security

The ability to ask the Abacus for an authorization decision *without* passing lots of data back and forth is a huge win for data security and privacy. As with the example of the bouncer at a club, instead of handing over all your information on your ID, just to get an authorization decision about your age, the bouncer can now ask the Abacus, which checks the necessary attributes and returns an authorization to the bouncer, who then acts based on the response. No longer do systems need the information from other domains, just to find out if the caller is authorized. A domain can make available sensitive attributes to the authorization domain and have no worries about those attributes ever being leaked to other systems, yet that data can still be used for decisions.

As DDD is respected, a domain will have few connections to its CDS. This results in easier administration of the domain, and there are less concerns about access from compromised credentials as there are less credentials available. Security is more easily moderated when there are only a few people with manually-assigned roles or groups while policies take care of the majority of the cases.

6.6 Reverse Query Functionality

The most powerful advantage of the Attribute Database is the ability to run queries against it. Domain owners often want to know, for auditing purposes or otherwise, “Who has access to this resource?” Because all the policies authorizing the resource are known to the engine, and thus the attributes needed, and since the Attribute Database contains the list of current resources, SQL statements can be constructed that query the Attribute Database for the IDs that have the requisite attributes.

Some modern PBAC systems contain reverse query functionality, but some do not. Of those that do, some require queries to be executed directly against domain production databases. This has the disadvantages of increased load on production systems. Also, if the data store is not a relational database, multiple types of queries must be constructed and then combined to give a response. With the Abacus, these reverse query statements are executed against the Attribute Database which neither 1) impacts domain servers, nor 2) impacts the performance of the decision engine itself. Production domains can use any type of storage model that they want, and the Abacus can still quickly generate a list of authorized entities because the attribute storage is decoupled from the domains.

7. Disadvantages of the Abacus

The Abacus provides significant advantages over existing architectures, as previously presented. While powerful, flexible, and novel, there are some considerations that must be evaluated before it can be implemented.

First, there is initial setup of the attributes takes time and effort. Domain owners must agree on the definition of an attribute. The technical integration work must then be done to push new attribute values to the

Abacus whenever the values change for identities in the system. Additionally, if domain A requires attributes from domain B, then the same work must be done in domain B to enable the policies from A to work properly, and this effort needs to be budgeted within B's schedule.

While not a technical challenge, the process of making business owners more directly responsible for their data governance does invoke push-back from some people. Traditionally, businesses will hand the policy requirements to IT teams and expect the work to be done. We have found that asking business owners to take ownership of the policies occasionally produces feelings of resentment and stubbornness where some feel that "that's IT's job".

The most difficult concern is data synchronization issues. When a message changing an attribute is dropped somewhere, then the Abacus may be permitting or denying inaccurately. One possible mitigation technique is to use database ETL (extract-transform-load) processes to verify accuracy with the source domains, but this breaches domain boundaries. Alternatively, an API could be created to allow domains to view the attributes that they own within the Abacus. The domains could then compare what the Abacus has with what they contain and (re)push needed changes. Future research should investigate this problem.

8. Conclusion

In this paper we presented a new architecture for authorization that completely respects DDD principles, simplifies the architecture of the authorization domain, more effectively secures data within domains, and gives more control over data access to domain owners. The Abacus ensures that domains may change technology without needing to rewrite their authorization logic, and domains can use attributes that are governed and maintained by other systems *without* needing to know the implementation and/or business logic of those systems. Configuration becomes much easier and simpler than utilizing roles or groups, or even than systems which require implementation details of the domain itself. We affirm that the Abacus is breaking new ground in authorization.

9. References

[1] E. Evans, Domain-Driven Design. Addison-Wesley, Boston, MA. 2004.

[2] A. Elliott and S. Knight, "Role Explosion: Acknowledging the Problem," Software Engineering Research and Practice, 2010, pp. 349-355.

[3] "A Third of Ex-Employees Accessing Company Data," IS Decisions blog. https://www.isdecisions.com/blog/it-security/a-third-of-ex-employees-accessing-company-data/. Accessed: June 2020.

[4] D. Recordon and D. Reed, "OpenID 2.0: A Platform for User-Centric Identity Management," In Proceedings of the second ACM workshop on Digital identity management, 2006, pp. 11-16.

[5] D. B. Cross, P. J. Hallin, M. W. Thomlinson, and T. C. Jones, "Digital Identity Management," US 7,703,128 B2, 2010.

[6] P. Steiner, On the Internet, nobody knows you're a dog. The New Yorker, 5 July 1993.

[7] P. A. Grassia, M. E. Garcia, and J. L. Fenton, "Digital Identity Guidelines." DRAFT NIST Special Publication 800-63-3 Digital Identity Guidelines. National Institute of Standards and Technology, Los Altos, CA, 2017.

[8] SolarWinds Access Rights Manager. SolarWinds Worldwide, LLC. https://www.solarwinds.com/access-rights-manager

[9] Grouper.InCommon. https://incommon.org/software/grouper/

[10] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone. "Guide to Attribute Based Access Control (ABAC) Definition and Considerations." NIST special publication 800.162 (2013).

[11] G. Batra, V. Atluri, J. Vaidya, and S. Sural, "Deploying ABAC Policies using RBAC Systems," Journal of Computer Security, vol. 27, no. 4, 2019, pp. 483–506.

[12] PingIdentity. Ping Identity. https://www.pingidentity.com/en.html

[13] NextLabs. Next Labs Inc. https://www.nextlabs.com/

[14] PlainID. PlainID. https://www.plainid.com/

[15] A. Cockburn, "Hexagonal architecture," 2012. http://alistair.cockburn.us/Hexagonal+architecture.

[16] V. Vernon, Domain-Driven Design Distilled. Addison-Wesley, 2016.

[17] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley, 2013.

[18] G. Helemski, "PBAC vs RBAC: Why Role Based Access Control Is Not Enough," Feb. 23, 2020. https://blog.plainid.com/why-role-based-access-control-is-not-enough.

[19] G. Orwell, Animal Farm. Secker and Warburg, London, England. 1945.