Attribute Stream-Based Access Control (ASBAC) - Functional Architecture and Patterns

Dominic Heutelbeck
FTK e.V.
Forschungsinstitut für Telekommunikation und Kooperation
44149 Dortmund - Germany
Email: dheutelbeck@ftk.de

Abstract—Current architectures and data flow models for access control are based on a request-response model. In stateful or session-based applications with requirements to monitor access rights over time this results in polling of authorization services and for Attribute-Based Access Control (ABAC) in the polling of policy information points. This approach introduces latency or increased load due to polling. This paper introduces a functional architecture for access control based on a publish-subscribe model, to avoid the bottlenecks and latency implied by polling authorization services. The functional architecture supports attribute sources of streaming nature and is thus called Attribute-Stream-based Access Control (ASBAC). In addition, the paper describes key patterns for implementing ASBAC policy enforcement points in asynchronous event-driven applications based on patterns such as reactive programming.

Keywords—attribute-based access control, ABAC, data streams, reactive programming, information assurance

I. Introduction

Attribute-based access control (ABAC) [1] has been a successful model for establishing reliable and flexible authorization infrastructures across applications in complex domains. In ABAC systems, access to resources is granted based on policies, i.e., rules and relationships which determine which behavior of a user or process (the subject) is authorized with regards to different resources. The policies themselves are encoded in a machine-readable format and the rules contained in the policies refer to traits of relevant objects, such as name, date of birth, unique identifiers, location, or security clearance level. These traits are called attributes, which may require accessing different data sources during policy evaluation to determine the outcome of the authorization process. These data sources are called policy information points (PIPs). Other well-established access control schemes, such as discretionary access control (DAC) [2], mandatory access control (MAC) [2], and role-based access control (RBAC) [3] can be expressed by ABAC systems.

The two dominant ABAC control models currently in use are either based on the eXtensible Access Control Markup Language (XACML) standard [4] or on Next Generation Access Control (NGAC) [5]. The functional architectures of both XACML and NGAC share the concept of a policy enforcement point (PEP), as a functional entity which enforces access decisions made by a policy decision point (PDP). These access control models follow a request-response pattern and usually establish the access rights once prior to access and then allow the operation in question to be executed. This approach works well for stateless services, such as RESTful APIs. The only way to continuously monitor access rights to a resource is to poll the PDP repeatedly. This introduces latency dependent on the poll frequency, and additional load on the infrastructure for communication and computation. Polling the PDP will also introduce load on the PIPs required for decision-making.

If the access control model is applied to the user interface tier to determine if content is to be displayed or UI elements are to be active during initial rendering (e.g., loading a web page) this may lead to a bad user experience when access rights change. E.g., the user expects to be able to perform an action as a button is active, but the domain logic returns an access denied error as permissions have changed in the meantime. Other problems include expected data not becoming visible immediately when expected, or data still being available after permissions have been revoked.

Another domain in which the request response scheme of current access control models may introduce bottlenecks are internet of things (IoT) applications based on managing and processing continuous...
II. Requirements

The goal of the architecture is to remove the need for polling the PDP and underlying PIPs and to switch from a request-response authorization flow to an event-driven, asynchronous model.

An authorization system that implements the basic ABAC model benefits from the expressiveness and acceptance of this paradigm. Thus, it should be applied in the authorization system.

As an ABAC systems, the decisions made by a PDP for a given authorization subscription depend on the following parameters:

- PDP configuration
- Policies published and persisted in the policy retrieval point (PRP).
- Return values of PIPs at the time of policy evaluation.

All these parameters are potentially subject to change while a subscription is active and may result in a different decision made by the policy decision point being sent to the subscribing PEP. The data sources should be accessed by the PDP in an event driven publish-subscribe fashion where supported.

The system should support domain-specific PIPs for integration with the application domain.

PIPs may return individual attributes, or deliver a stream of attribute values over time, e.g., location tracking information of objects to support quasi real-time geographical access control policies. Thus the name "Attribute Stream-Based Access Control".

Due to the increased relevance of JSON as a data format in web applications, the system should be able to easily integrate with JSON-based services.

III. Functional Architecture

The ASBAC functional architecture, as illustrated in Figure 1, is based on the architectures of both XACML and NGAC, with a policy decision point in the center, responsible for interpreting policies and making authorization decisions. The primary difference to the previously mentioned architectures is not the structure of components, but the way they communicate. In XACML and NGAC a single authorization request made by the PEP to the PDP results in a number of requests and responses to and from the connected components for accessing policies, configuration, and external attributes.

In ASBAC the authorization request is replaced by an authorization subscription to the PDP (2). The PDP in turn subscribes to the PDP configuration (3). The configuration contains fundamental parameters for the decision making (e.g., the default policy combining algorithm selected for the deployment) or parameters for PIPs connected to the PDP (e.g., access tokens for message brokers or RESTful APIs).

In addition, the PDP makes a subscription to the policy retrieval point, parameterized by the authorization subscription. The PRP will return all policies (policy

Streams of sensor data and control information. This data is typically distributed using a message-broker (MB) allowing for asynchronous publish-subscribe communication. With regards to expressiveness ABAC is considered a model well suited for access control in these systems [6]. However, in access control models tailored towards message brokers, such as discussed in [6], only the act of subscribing to a data stream or publishing to it are covered by the operational model. The forwarding of individual messages is in this case hard coded and verifies if the matching topic has previously been subscribed to by any given accessing system.

Since the introduction of XACML the industry has seen wide adoption of structured data serialization formats other than XML. Especially the JSON [7] format has found wide adoption for web services and IoT messages. The development of access control systems has been picking up on this and several approaches exist which use JSON for representing data or policies. E.g., the XACML JSON Profile [8] specifies a JSON representation for the authorization requests and responses in communication with a PDP. JACPoL [9], [10] presents a JSON-based ABAC policy language, which proves to be less resource consuming than XACML policies. JACPoL omits the possibility to access external data sources for attributes and only uses attributes encoded in the authorization request. Both XACML and JACPoL make use of the hierarchical document structure of XML or JSON to represent rules and policies, resulting in optimization for machine processing, but are difficult to manually author by administrators since the semantics of a rule are often difficult to understand between the verbose notation in XML, and to a lesser extent, JSON. This results in an error-prone authoring process. This has been identified by commercial vendors of XACML systems, and Abbreviated Language for Authorization (ALFA) [11] has been developed as a domain-specific language for policy authoring, which can be translated back and forth to XACML, improving the readability of policies. Other work [12] proposes to apply JSON Schema for modeling an ABAC rule language.

This paper introduces the Attribute Stream-Based Access Control (ASBAC) functional architecture for resolving problems introduced by polling. In addition, key patterns for implementing policy enforcement in applications are described.
sets, rules if applicable) to be evaluated for the authorization subscription, and sends a new set of matching policies if they change due to policy management activities of the policy administration point. Each time the PDP has a new set of configuration and matching policies, it starts evaluating the policies.

During the evaluation, when the PDP encounters references to policy information points, the PDP subscribes to them (5) as encoded in the policies and aggregates incoming attributes from the PIPs to calculate the most recent decision (6), which is forwarded to the PEP if it differs from the most recent previous decision (7).

The PEP then enforces the decision (8) and, if applicable, performs the requested action. Then it provides the client application with access to the return value, if present, of the action (9) which may be altered before being delivered to the client application (10), if this is indicated by the decision.

The architecture should at every step be implemented using non-blocking asynchronous code. The next sections will discuss an implementation strategy for doing so, while providing a more precise picture of the ASBAC architecture than the high-level perspective provided in this section.

IV. Reactive Programming

In the following, this paper presents a number of asynchronous algorithms for handling authorization subscriptions where asynchronous information can change the PDP’s decision over time. A publish-subscribe design is applied, similar to the observer pattern [13]. More precisely, the principles of reactive programming as proposed by the Reactive Manifesto [14] are applied, which also addresses requirements regarding responsiveness, resilience, elasticity and message-driven systems.

A notable difference to the original observer pattern is the introduction of functionality for handling back-pressure, i.e., for informing upstream data sources about bottlenecks in downstream processing and allowing for elastic adjustment of data transfer.

A number of runtime environments provide APIs for implementing reactive applications. Implementations include cross platform efforts like ReactiveX [15], or platform-specific implementations like Project Reactor [16] or domain-specific languages like ELM [17].

To be able to express algorithms following this paradigm, this paper uses a simplified notation for reactive algorithms, loosely based on Project Reactor, which is also used in the ASBAC reference implementation. This notation mostly ignores error handling and only introduces a few required asynchronous operations on data streams.

Definition 1. A value is a chunk of serialized data. For simplicity, we assume that all values are expressed as JSON [7] values. Then let JSON be the set of all JSON values.

Definition 2. Data streams or short streams are composable, asynchronous sequences of values. Let S be the set of all possible streams.

Definition 3. For a stream, \( s \in S \), \( s.first() \) returns a stream only containing the first element of the stream \( s \).
Definition 4. For a stream, \( s \in S \), \( s\text{.distinctUntilChanged}() \) returns a stream which is \( s \) with subsequent repetitions filtered out.

Definition 5. Let \( \text{combineLatest} : [S_1, \ldots, S_n] \to S, n \in \mathbb{N} \) be a function returning a stream which, starting from the time all input streams have emitted at least one value, emits an array of the most recent values for each input stream whenever a new value is emitted by any input stream.

Definition 6. For a stream, \( s \in S \), \( s\text{.map}(\text{fun}|\text{fun} : \text{JSON} \to \text{JSON}) \) returns a stream, where for each value \( v \in s \), \( s\text{.map}(\text{fun}) \) contains \( \text{fun}(v) \).

Definition 7. For a stream, \( s \in S \), \( s\text{.flatMap}(\text{fun}|\text{fun} : \text{JSON} \to S) \) returns a new stream merging the results of \( \text{fun}(v) \) for each value \( v \in s \) into a new stream asynchronously, i.e., the inner streams are flattened into one output stream.

V. Reactive ASBAC Policy Decision Point Realization

This section discusses how the PDP composes the individual data streams during decision making for an authorization subscription based on the reactive programming model. The following terms formalize components of the functional architecture. At this point the model is formulated mostly agnostic to the specific syntax and semantics of the policy language used. A few assumptions about the policy language will be made though. A full implementation of a police engine and domain specific language for ASBAC has been developed alongside the functional model. The policy language and engine are out of scope of the paper and will be covered in further publications. The engine, language and demos are published as open source on GitHub [18], [19], [20].

Definition 8. A protected resource, \( r \), is an element of \( R^* := S \cup R \). Resources can be streams or in the set \( R \) of any other object depending on the domain logic or runtime environment.

Definition 9. Let \( Subjects \subset \text{JSON} \) be the set of values describing all potential users and processes which may attempt to access any given \( resource \in R^* \). Let \( Resources \subset \text{JSON} \) be the set of values identifying and describing any given \( resource \in R^* \). Let \( Actions \subset \text{JSON} \) be the set of values describing all potentials actions to be performed by any \( subject \in Subjects \) on any \( resource \in R^* \).

Definition 10. Let \( sub \in ASub \subset \text{JSON} \) be an authorization subscription. \( ASub \) is the set of all objects containing the keys \( subject, action, \) and \( resource \) with matching values from \( Subjects, Actions, Resources \). A \( sub \) is used to indicate that a PEP requests a decision stream from the PDP about the authorization of the subject to perform the action with the given resource. The three values have to unambiguously describe these objects. For JSON objects, values of keys are considered attributes of the object. The resource object may be a direct JSON representation of the resource, if applicable.

Definition 11. An authorization decision, \( d \), is an object value containing a decision \( \in \{"PERMIT", "DENY", "INDETERMINATE", "NOT_APPLICABLE"\} \), an optional array of values for obligations, an optional array of values for advices, and an optional resource value. The values of decision follow the same semantics as XACML [4]. Obligations encode actions that must be performed when the given subject accesses the resource. If a PEP cannot follow all obligations, the decision must be handled as if the result was "DENY". For actions encoded in advices only a best effort to perform them should be made. Advices also may contain any supplementary information. No further assumptions are made about the semantics of obligations, advices and their encoding as this may be domain specific. If a resource value is present, the decision indicates, that the originally provided resource in the request should be replaced by the value returned by the PDP. Let \( ADec \subset \text{JSON} \) be the set of all possible authorization decisions.

Definition 12. A policy, \( p \in Policies \), is a document containing rules expressed in a policy language, where \( Policies \) is the set of all policies expressible in some policy language, e.g. SAPL [18]. The policy language should include semantics similar to XACML target blocks, allowing for defining conditions under which the policy should be fully evaluated, i.e., the policy \( p \) matches \( sub \in ASub \). The function \( eval : Policies \times ASub \to SDec \), returns a stream of policy decisions for a subscription as indicated by the input policy. \( SDec \) is the set of streams only containing authorization decisions.

The rules within a policy may want to access attributes of the subject or resource, which are not contained in the subscription. These have to be unambiguously identified. During evaluation, \( eval \) will subscribe to the policy information points able to provide the attributes in question. For the purpose of this paper, we assume, that these attributes are aggregated using the \( \text{combineLatest} \) and \( \text{distinctUntilChanged} \) operations. Then the policy will be evaluated to return a decision whenever a new aggregated set of attributes is available. This is a naive way to implement attribute streaming for the policy engine. In practice, given a...
concrete policy language, the subscription to PIPs and evaluation of policies can be more complex. The core idea is to transform the policy into an abstract syntax tree (AST) and to compose the `eval` function through composition of reactive stream operations based on the AST of the policy. This allows for dynamic conditional subscription based on lazy evaluation of expressions, limiting subscriptions to PIPs which are currently relevant for the decision-making process. These algorithms will be covered in future publications.

**Definition 13.** A policy retrieval point is a component containing a mutable set of policies. The function \( prp : ASub \rightarrow S \) returns a stream of arrays containing the subset of policies in the PRP which match \( ASub \).

Policy retrieval points may use the target block information for indexing policies, eliminating the need to evaluate the target blocks of all policies for each subscription, as implemented in SAPL [18].

**Definition 14.** Let \( pCombine : [d_1, \ldots, d_n] \mapsto d, \ d_i, d \in ADec, \ i \in \{1, \ldots, n\} \), be a policy combining-algorithm, mapping the array of decisions to a single decision.

A number of combining algorithms can be found in XACML and SAPL [18]. E.g., a "deny unless permit" algorithm will always return a deny unless at least one decision indicating a permit is present. In case the decisions contain obligations, advices, or resources, different combining algorithms may implement individual strategies for aggregating this information. E.g., include the obligations and advices from all decisions, and return an indeterminate decision, if decisions disagree about the resource value.

Based on these components, a reactive ASBAC policy decision point can be implemented as follows. The modeling of, and subscription to the PDP configuration have been omitted for simplicity.

**Algorithm 1 Policy Decision Point**

Given: \( sub \in ASub \)

\[
\text{combineLatest(}
\begin{align*}
\text{prp(sub)} \\
\text{.map([&p_1, \ldots, p_n] \mapsto \text{[\text{eval}(p_1), \ldots, \text{eval}(p_n)])} \\
\text{).map(pCombine).distinctUntilChanged()} &
\end{align*}
\]

The policy decision point exactly follows the rules outlined by the functional architecture. A stream of matching policies is mapped to a set of decision streams generated by evaluating the individual policies. The decisions made by the individual policies are aggregated using `combineLatest` and then are combined into a final decision by applying a policy-combining algorithm. Then subsequent duplicate decisions are filtered out of the resulting stream.

**VI. Reactive Policy Enforcement Point Patterns**

The practical implementation of a PEP depends on the application domain and the type of action and resource in question at any given PEP.

In the following, a number of typical use cases are identified, and the general patterns of matching PEP implementations are outlined. The first use case illustrates how to apply ASBAC to data streams. The two remaining use cases determine one-time access decisions for the execution of an action on a resource, as a reactive equivalent to other ABAC systems, e.g., based on XACML.

**Definition 15.** \( RAP_{r \in R^*, a \in Actions} \) is the Resource Access Point (RAP) component that provides the only method of executing the action \( a \) on resource \( r \). \( RAP_{r \in R^*, a \in Actions, access()} \) performs the given action and returns a stream. The returned stream depends on the application domains. For example, actions only performing a side effect or state change may return an empty stream.

**Definition 16.** A Policy Enforcement Point \( PEP \) is the only means of accessing any RAP and performing actions on the underlying resource for other components in the software system. In practice, a PEP uses knowledge about the application domain and system context to determine an appropriate authorization subscription \( ASub \), subscribes to \( PDP_{ASub} \) and enforces the returned policy decisions during access, including the execution of obligations and advices provided by the decisions, or resource replacement when indicated.

**A. Enforcing Obligations and Advices**

The requirements on the PEP with regards to obligations is stricter than in XACML 3.0 [4], where an obligation only \( should \) be performed, while here it is defined as a hard requirement. This also means, that obligations should be reversible operations, or at least provide a means of documenting that the operation happened an attempted access which was later denied because its obligations were no longer met following the successful execution of another obligation. As the authorization system is to be implemented asynchronously, global blocking transactions should be avoided as reversible or event-based approaches are better suited in such a runtime environment.

The following algorithm outlines the processing of enforcing obligations and advices. Modifications may be necessary due to application domain requirements.
Algorithm 2 Obligation and advice handling

function enforceConstraints(d ∈ ADec)
for all o ∈ d.obligations ∪ d.advices do
    attempt to perform o
end for
if Any o ∈ d.obligations not performed then
    roll back all previously successful operations
return error "access denied"
end if
return d

B. Streaming Enforcement

Typical use cases for ASBAC are applications where a resource is in use over a longer time in conjunction with a user session. Two examples are interactive (web)applications where certain resources are made available, or processes accessing data streams over a longer period of time.

In both cases, it may be reasonable to assume, that access rights may change during the session. Algorithm 3 realizes a PEP stream access scenario. For the web application, the PEP should instead be integrated into the underlying UI implementation pattern, such as the model-view-controller or model-view-presenter pattern.

Algorithm 3 Stream RAP Enforcement

Given: sub ∈ ASub, a RAP pdp(sub).first()
combineLatest([RAP.access(), pdp(sub)])
.map((value, d) →
    if (d.decision ≠ "PERMIT")
        return error "access_denied";
    else
        return d
).map(enforceConstraints)
.map(d →
    result = RAP.access();
    if (d.resource is present)) {
        return d.resource;
    } else {
        return result;
}).

C. Pre RAP Enforcement

A typical use case is the enforcement of policy decisions before invoking the RAP. This pattern is applicable in situations, where the authorization subscription sub ∈ ASub can be formulated without knowledge of return values or side effects of performing the action.

In Algorithm 4 the authorization subscription ASub is constructed based on a-priori knowledge. Then only the first decision returned by the PDP PDPASub.first() is consumed, the subscription is canceled. Then, it checks if the decision was "PERMIT", and all obligations were performed. Then, the RAP is accessed, and the result is replaced if indicated by the decision. For replacement, typing limitations may apply depending on the implementation language and runtime environment. Otherwise a "permission denied" error occurs.

D. Post RAP Enforcement

The final use case is the enforcement of policy decisions after the RAP access. It should be avoided in cases where using the RAP performs a state change in the application domain model. This pattern is to be applied to RAP actions where the returning streams only contains one element, else the streaming pattern is to be used. It is intended for reading access, where the authorization subscription ASub cannot be formulated without knowledge of return values or side effects of performing the action, or where the application domain implies that the PDP is likely to demand a replacement of the return value with a transformed version.

In Algorithm 5 the authorization subscription sub(value) ∈ ADec is constructed based on the return value of a read action accessing a resource. Then only the first decision returned by the PDP is consumed. It checks if the decision was "PERMIT" and all obligations were performed. The result is replaced if indicated by the decision. For replacement, typing limitations may apply depending on the implementation language and runtime environment. Else a "permission denied" error occurs.

The post enforcement algorithm can be combined with the pre enforcement algorithm.
Algorithm 5 Post RAP Enforcement

Given: RAP

\[
\text{RAP}.\text{access}().\text{first()}
\]
\[
 .\text{map}(\text{value} \mapsto [\text{value, sub(value)}])
\]
\[
 .\text{flatMap}([\text{value, s}] \mapsto
\]
\[
 \text{pdp}(s).\text{first()}
\]
\[
 .\text{map}(d \mapsto
\]
\[
 \text{if} (d.\text{decision} \neq \text{"PERMIT"})
\]
\[
 \text{return} \text{error} \text{ "access_denied"};
\]
\[
 \text{else}
\]
\[
 \text{return} \ d.\text{resource};
\]
\[
 \text{if} (d.\text{resource} \text{ is present}) \{ \text{return} \ d.\text{resource}; \}
\]
\[
 \text{else} \{ \text{return} \ \text{value}; \}
\]

VII. Conclusions

This paper presented a new functional architecture for attribute stream-based access control (ASBAC) solving a number of issues occurring with existing ABAC systems in scenarios where access rights may change during access and polling the authorization infrastructure is not an acceptable solution. The proposed reactive implementation strategy illustrates how to implement ASBAC. An authorization infrastructure following the proposed patterns is a natural fit for applications following the reactive programming model and allows for quasi-real time publish-subscribe authorization in stateful applications.

A complete reference implementation of the ASBAC architecture has been implemented and is publicly available as open source under the Apache 2.0 license at [18], including a full domain specific language (SAPL) for expressing ASBAC policies. A number of case studies based on medical scenarios and geographic access control are available [20].

This paper presented a primary conceptual view on ASBAC which is the foundation for presenting further results, such as the design and implementation of the policy language, policy evaluation algorithms, and data structures for policy indexing.

Future work will examine how to apply ASBAC in IoT applications. Also, ASBAC is missing an administrative model, which should be investigated.

Acknowledgment

The author would like to thank Felix Siegrist and Colin de Klerk for their help.

References