Incorporating Off-Line Attribute Delegation into Hierarchical Group and Attribute-Based Access Control

Daniel Servos and Michael Bauer

Department of Computer Science, Western University, London, Ontario, Canada
{dservos5,bauer}@uwo.ca

Abstract. Efforts towards incorporating user-to-user delegation into Attribute-Based Access Control (ABAC) is an emerging new direction in ABAC research. A number of potential strategies for integrating delegation have been proposed in recent literature but few have been realized as full ABAC delegation models. This work formalizes one such strategy, entitled User-To-User Attribute Delegation, into a working delegation model by extending the Hierarchical Group and Attribute-Based Access Control (HGABAC) model to support dynamic and *"off-line"* attribute delegation. A framework to support the proposed delegation model is also presented and gives implementation details including an updated Attribute Certificate format and service protocol based on the Hierarchical Group Attribute Architecture (HGAA).

Keywords: Delegation · Attribute-Based Access Control · ABAC · HGABAC

1 Introduction

Attribute-Based Access Control (ABAC) is an access control model in which users are granted access rights based on the attributes of users, objects, and the environment rather than a user's identity or predetermined roles. The increased flexibility offered by such attribute-based policies combined with the identityless nature of ABAC have made it an ideal candidate for the next generation of access control models. The beginnings of ABAC in academic literature date back as early as 2004 with Wang, et al's Logic-Based Framework for Attribute Based Access Control [9] and even earlier in industry with the creation of the eXtensible Access Control Markup Language (XACML)[1] in 2003. However, it is only in recent years that ABAC has seen significant attention[7]. This renewed interest has lead to the development of dozens of ABAC models, frameworks and implementations, but to date, few works have touched on the subject of delegation or how it might be supported in attribute-based models.

Delegation is a key component of comprehensive access control models, enabling users to temporarily and dynamically delegate their access control rights to another entity after policies have been set in place by an administrator. This ability allows users to adapt to the realities of everyday circumstances that are not possible to foresee during policy creation and is critical in domains such as healthcare[3]. ABAC brings new problems and complications when incorporating delegation that are not present in the traditional models (RBAC, DAC, MAC etc.) [6]. In the traditional models, delegation is relatively straightforward, a set of permissions or role memberships (as in RBAC) is delegated directly by a delegator to a delegatee under set conditions (e.g. an expiry date or "depth" of delegation). In ABAC, this is complicated by identityless nature of ABAC (i.e. access control decisions being made on the basis of attributes rather then the user's identity) and the flexibility of attribute-based policies that may include dynamic attributes such as the current time, physical location of the user, or other attributes of the system's environment.

In a previous work[6], we offered a preliminarily investigation into the possible strategies for incorporating delegation into ABAC and the benefits and drawbacks of each method. A number of these proposed strategies have been further developed into working models by others, such as the work by Sabathein, et al.[4] towards creating a model of delegation for the Hierarchical Group and Attribute-Based Access Control (HGABAC)[5] model using our User-to-Attribute Group Membership Delegation Strategy[6]. In this paper, we seek to explore and put forth a novel attribute-based delegation model based on an unutilized delegation strategy, User-to-User Attribute Delegation[6]. We offer both an extension to the HGABAC model to provide a theoretical blueprint for incorporating delegation as well as an extension to Hierarchical Group Attribute Architecture (HGAA)[8] to provide a practical means of implementing it. Unlike current efforts, a particular emphasis is placed on maintaining the identityless nature of ABAC as well as the ability to delegate attributes in an "off-line" manner (i.e. without the user having to connect to a third party to perform delegation).

The remainder of this paper is organized as follows; Sec. 2 introduces the potential delegation strategies developed in our previous work and gives background on the HGAA and HGABAC model. Sec. 3 details our model of User-to-User Attribute Delegation and how it is incorporated into the HGABAC model. Sec. 4 provides a framework for supporting our delegation model and details modifications to the HGAA architecture to support it, including new extensions to the Attribute Certificate format to include delegation concepts. Finally, Sec. 5 gives concluding remarks and directions for future work.

2 Background

2.1 The HGABAC Model

HGABAC[5] offered a novel model of ABAC that introduced the concept of hierarchical user and object groups. Attributes are assigned both directly to access control entities (e.g. users and objects) and indirectly assigned through groups. Users then have a *direct* set of attributes, directly assigned by an administrator, as well as an *inherited* set of attributes, indirectly assigned to them via their membership in one or more user groups. The set of attributes used for policy evaluations is the user's *effective* attribute set, that is, the set that is the result

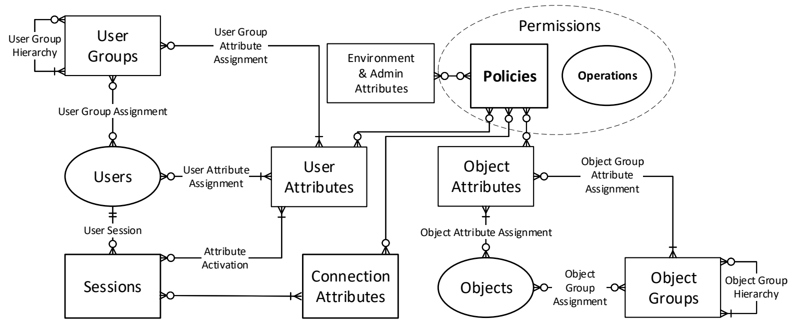


Fig. 1. HGABAC relations and components denoted in Crow's Foot Notation. Primitive components are shown in ovals.

of merging their *direct* and *inherited* attribute sets. This style of group membership and attribute inheritance is also used to assign attributes indirectly to objects via objects membership in object groups. These relations and the basic elements of HGABAC are shown in Fig. 1 and a brief description of the basic HGABAC entities, relations and functions are as follows:

- **Attributes:** attributes are defined as attribute name, type pairs, that is $att = (name, type)$, where *name* is a unique name for the attribute and *type* is a data type (e.g. string, integer, boolean, etc.). When assigned to entities via direct assignment (e.g. User Attribute Assignment) or groups (e.g. User Group Attribute Assignment) they are associated with a set of values being assigned for that attribute.
- **Users (U):** entities that may request access on system resources through sessions.
- **Objects (O):** resources (files, devices, etc.) for which access may be limited.
- **Operations (Op):** operations that may be applied to an object (read, write, etc.).
- **Policies (P):** policy strings following the HGPL policy language.
- **Groups (G):** a hierarchical collection of users or objects to which attributes may be assigned. Group members inherit all attributes assigned to the group and the groups parents in the hierarchy. Defined as $g = (name, m \subseteq M, p \subseteq G)$ where *name* is the name of the group, m is the set of members, M is either the set of all Users or all Objects, p is the groups parents and G is the set of all groups.
- **Sessions:** sessions allow users to activate a subset of their effective attributes. This subset is used as the user attributes for policy evaluations. Sessions are represented as a tuple s in the form $s = (u \in U, a \subseteq \mathit{effective}(u \in U), \mathit{con\_atts})$ where u is the user who owns the session, *con_atts* is the set of connection attributes for the session and a is the set of attributes the user is activating.
- **Permissions:** a pairing of a policy string and an operation in the form $perm = (p \in P, op \in Op)$. A user may perform an operation, *op*, on an object if there exists a permission that contains a policy, p, that is satisfied by the set of attributes in the user's session, the object being accessed and the current state of the environment.
- **inherited(x):** mapping of a group or user, x, to the set of all attributes inherited from their group memberships and the group hierarchy.
- **effective(x):** mapping of a group or user, x, to the attribute set resulting from merging the entities directly assigned and inherited attribute sets.

```
P1: /user/age >= 18 AND /object/title = "Adult Book"
P2: /user/id = /object/author
P3: /policy/P1 OR /policy/P2
P4: /user/role IN "doctor", "intern", "staff" AND /user/id != /object/patient
```

P1 describes a policy requiring the user to be 18 or older and the title of the object to be "Adult Book". P2 requires the user to be the author of the document. P3 combines policies P1 and P2, such that the policy is satisfied if either policy (P1 or P2) is satisfied. Finally, P4, requires a user's role to be one of "doctor", "intern", or "staff" and that they are not listed as the patient.

Fig. 2. Example HGABAC HGPLv2[8] policies.

The largest advantage of attribute groups is simplifying administration of ABAC systems, allowing administrators to create user or object groups whose membership indirectly assigns sets of attribute/value pairs to its members. The hierarchical nature of the groups, in which child groups inherit all attributes from their parent groups, allow for more flexible policies, administration and even emulation of traditional models such as RBAC, MAC and DAC when combined with the HGABAC policy language.

Permissions in HGABAC take the form of operation/policy pairs, in which an operation is allowed to be performed if the policy is satisfied by the requesting user's active attribute set, attributes of the object being affected, the current state of the environment and a number of other attribute sources. Policies are defined in the HGABAC Policy Language (HGPL), a C style language using ternary logic to define statements that result in *TRUE*, *FALSE* or *UNDEF*. Example HGPLv2 policies are given in Fig. 2 and the full language is defined in [8]. These policies would be combined with an operation to form a permission. For example the permission $Perm_1 = (P1, read)$ would allow any user who is at least 18 years of age to read the object titled "Adult Book" based on the permission $P1$ from Fig. 2. Similarly, the permission $Perm_2 = (P2, write)$ would allow any author of an object to write to that object.

2.2 Hierarchical Group Attribute Architecture (HGAA)

While HGABAC provides an underlying model for ABAC, a supporting architecture is still required to provide a complete system and facilitate use in real-world distributed environments. HGAA[8] provides a system architecture and implementation details for HGABAC that answer questions such as *"who assigns the attributes?"*, *"how are attributes shared with each party?"*, *"how does the user provide proof of attribute ownership?"*. and *"where and how are policies evaluated?"* that are often left unanswered by ABAC models alone. HGAA accomplishes this by adding five key components:

- **Attribute Authority (AA):** A service responsible for managing, storing and providing user attributes by issuing attribute certificates.
- **Attribute Certificate (AC):** A cryptographically secured certificate listing a user's active attributes for an HGABAC session as well as revocation information.
- **HGABAC Namespace:** A URI-based namespace for uniquely identifying attributes and HGABAC elements across multiple federated domains and authorities.
- **Policy Authority:** A service which manages and evaluates HGABAC policies on behalf of a user service provider.
- **User Service Provider:** A provider of a service to end users that has restricted access on the basis of one or more HGABAC policies (i.e. the service on which access control is being provided).

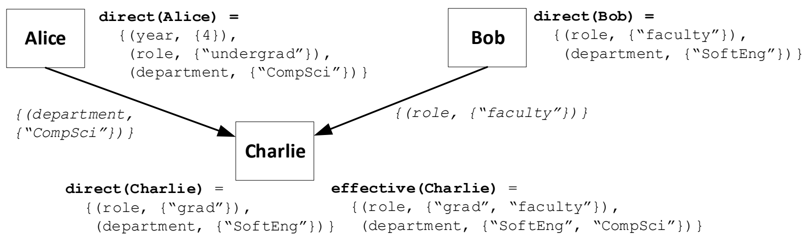

Alice
direct(Alice) = {(year, {4}), (role, {"undergrad"}), (department, {"CompSci"})}
Bob
direct(Bob) = {(role, {"faculty"}), (department, {"SoftEng"})}
{(department, {"CompSci"})}
{(role, {"faculty"})}
Charlie
direct(Charlie) = {(role, {"grad"}), (department, {"SoftEng"})}
effective(Charlie) = {(role, {"grad", "faculty"}), (department, {"SoftEng", "CompSci"})}

Fig. 3. Example of User-to-User Attribute Delegation. Arrows denote direction of delegation (arrow points to delegatee), boxes represent users of the system.

In a system following HGAA, users request Attribute Certificates (AC) from their home domain's Attribute Authority (AA) containing a list of their attributes for a session. Users may then use this AC to make requests on protected user services (both in their home domain or run by external organizations). These protected services verify the user's AC and check that the user has permission to access the service using their local domains Policy Authority. A key feature of the architecture is the separation of the AA from the other services. Once users are issued an AC, there is no longer a need for the user or other parties to contact the AA for the duration of the session. Separating services in this way simplifies the problem of user's using their attribute based credentials across independent organizational boundaries. Services run by external organizations need not communicate with the user's home organization to verify their attributes beyond trusting the home organization's public key used to sign ACs.

The framework presented in Sec. 4 extends the AC format to add delegation related features. ACs are ideal for this purpose as room has been left for future extensions including space for delegation extensions that were not part of the original work. Sec. 4 also presents a modification to the HGAA protocol to add an optional delegation step in which users may delegate part of their certificate to another user.

2.3 Potential Delegation Strategies

In our previous work[6], we explored possible strategies for integrating delegation into ABAC and propose several potential methods primarily based on the access control element being delegated (e.g. attributes, group memberships, permissions, etc.). Each family of strategies results in unique proprieties and complications to overcome. The delegation model and architecture presented in the subsequent sections (Sec. 3 and 4) of this paper are based on the User-to-User Attribute Delegation strategy in which users acting as a delegator, delegate a subset of their user attributes to another user acting as the delegatee. The delegated attributes are merged with the delegatee's directly assigned attributes (i.e. assigned through any means but delegation) to form the delegatee's set of user attributes used in policy evaluations.

An example of this style of delegation is shown in Fig. 3, in which Alice (the delegator) delegates a subset of their directly assigned attributes to the user Charlie (the delegatee) such that Charlie may satisfy the policy requiring the user to be in the Computer Science department (e.g. `user.department = "CompSci"`). At the same time, the user Bob (a second delegator) also delegates a subset of their attributes to Charlie such that Charlie may satisfy a policy requiring him to be a faculty member in the Software Engineering department (e.g. `user.role = "faculty" AND user.department = "SoftEng"`). In this example, both the attributes delegated by Alice (a *department*

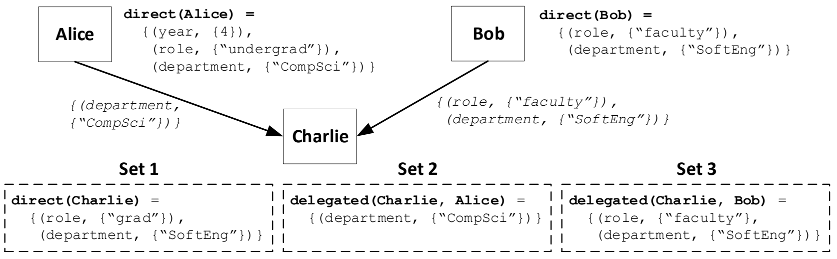

Alice
direct(Alice) = {(year, {4}), (role, {"undergrad"}), (department, {"CompSci"})}
Bob
direct(Bob) = {(role, {"faculty"}), (department, {"SoftEng"})}
{(department, {"CompSci"})}
{(role, {"faculty"}), (department, {"SoftEng"})}
Charlie
Set 1
Set 2
Set 3
direct(Charlie) = {(role, {"grad"}), (department, {"SoftEng"})}
delegated(Charlie, Alice) = {(department, {"CompSci"})}
delegated(Charlie, Bob) = {(role, {"faculty"}, (department, {"SoftEng"})}

Fig. 4. Example of Isolated Attribute Delegation. Arrows denote delegation direction and solid boxes users. Charlie may activate one set shown in dashed boxes at a time.

attribute with the value *"CompSci"*) and the attributes delegated by Bob (a *role* attribute with the value *"faculty"*) are combined with Charlie's directly assigned attributes to form their effective attribute set. Note that Bob only needed to delegate the *role* attribute as Charlie already had a *department* attribute with the value *"SoftEng"*.

While this style of attribute delegation may seem straightforward, our previous investigation[6] identified a number of potential issues regarding Attribute Delegation:

Conflicting Policy Evaluations: Merging attribute sets can lead to multiple values for an attribute. While this is an intended feature of HGABAC, it can lead to unintended conflicts when the values are a result of delegation as opposed to careful design. For example, the policy `user.department` $\neq$ `"SoftEng"` results in two different results for Charlie in Fig. 3 depending on the value of *department* used.

User Collusion: Merging attribute sets allows users to combine their attributes such that they may satisfy policies they could not individually. In Fig. 3, Alice and Charlie may collude to satisfy the policy `user.department` = `"CompSci" and user.role` = `"grad"`, if Alice delegates her *department* attribute such that Charlie satisfies the policy.

Loss of Attribute Meaning: A key ABAC feature is that attributes are descriptive of their subjects and policies are created with this in mind. Merging attribute sets leads to an effective set that is no longer descriptive of the user. In Fig. 3, it is not clear what Charlie's role or department is based solely on their effective attribute set. While this makes delegation possible, it increases the difficulty of policy creation as all allowable delegations and combinations of attributes would need to be taken into account.

To resolve these issues, the delegation model and architecture described in this paper takes a modified approach to User-To-User Attribute Delegation in which the attributes delegated to a delegatee from delegators are isolated and not merged with the delegatee's directly assigned attribute set. A delegatee may then choose to activate either their own directly assigned attributes or their delegated attributes in a given session but never both at the same time. An example of this approach is shown in Fig. 4. In this example, Alice still wishes to delegate their attributes such that Charlie may satisfy the policy `user.department = "CompSci"` and Bob still wishes Charlie to satisfy the policy `user.role = "faculty" AND user.department = "SoftEng"`. In Alice's case, they still only delegate their *department* attribute, however, now Charlie must choose between activating the directly assigned attribute set, *Set 1*, or the set delegated by Alice, *Set 2*. Charlie is unable to combine their own directly assigned attributes with those delegated by Alice and must activate the delegated attributes *(Set 2)* to satisfy the policy `user.department = "CompSci"`. Note that in this case Alice was only required to delegate a subset of their attributes to satisfy this policy.

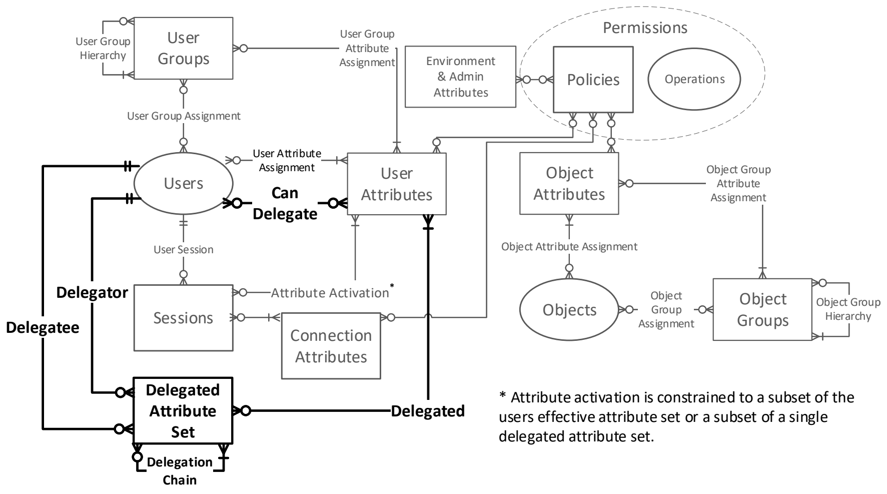


Fig. 5. User-to-User Attribute Delegation extension to the HGABAC model. Added components are bold and black, original components are greyed out.

In the case of Bob, it is now required that both Bob's *role* and *department* attributes are delegated to Charlie. In the previous example (Fig. 3), Bob only needed to delegate their *role* attribute as Charlie was already assigned a *department* attribute with the value *"SoftEng"* and it was merged with the attributes delegated by Bob. With the attributes sets isolated, both attributes are required as Charlie may not merge them with his own. The policy `user.role = "faculty" AND user.department = "SoftEng"` will only be satisfied when the set delegated by Bob *(Set 3)* is activated.

Isolation of delegated attributes avoids conflicting policy evaluations (at least those caused by delegation) and user collusion as attributes sets are not merged. Attribute meaning is maintained to the extent that the active attribute set will be descriptive of either the delegatee or delegator. Issues with user comprehension, while still an open ABAC problem[7], are abated as the delegator can be ensured that regardless of the attributes delegated, the delegatee will not be able to satisfy any extra policy that they themselves are not able to. It is important to note that negative policies such as `role` $\neq$ `"undergrad"` are still problematic as a user could simply not delegate an attribute with a restricted value. This is however a problem with negative policies in all ABAC models that allow users to activate a subset of their attributes (such as HGABAC) and not simply one limited to attribute delegation.

3 Delegation Model

3.1 Delegated Attribute Set

Several extensions to HGABAC model are required to support User-to-User attribute style delegation. The most critical is the addition of the *Delegated Attribute Set* component (shown in Fig. 5 with the other extensions), which contains the set of attributes delegated to a user in addition to the rules under which the delegation is permitted. This component is defined as follows where *DAS* is the set of all *Delegated Attribute Sets* in the system:

$$\begin{aligned}&\forall das \in DAS :\\ &\quad das = (delegatee \in U, delegator \in U, att\_set, depth \in \mathbb{N}_{\leq 0}, rule\_set \subseteq P, parent \in DAS)\end{aligned} \quad (1)$$

where *delegatee* is the unique ID of the user who is the recipient of the delegation, and *delegator* is the unique ID of the user who initiated the delegation. *att_set* is the set of attributes being delegated and their corresponding values (defined the same as attribute sets in the HGABAC model). The *att_set* is constrained to only containing attributes listed in *delegatable(delegator)* function (as defined in Sec. 3.2). *depth* is a positive integer selected by the delegator that describes how many more levels of delegation are permitted (e.g. if the user can further delegate these attributes on to another user). *rule_set* is the set of policies in the HGPL policy language selected by the delegator that must all evaluate to `TRUE` for the delegation to be maintained. Finally, *parent* is a reference to another *Delegated Attribute Set* in the case of subsequent delegations (see Sec. 3.3) or ∅ if this is the root of the delegation chain.

For example, the following *Delegated Attribute Set* would be created by the user Bob to delegate their role and department attribute to Charlie as shown in Fig. 4.

$$\begin{aligned} del\_att\_set = (\quad & Charlie, \\ & Bob, \\ & \{ \quad (role, \{\text{“}faculty\text{”}\}), \\ & \quad (department, \{\text{“}SoftEng\text{”}\}) \quad \}, \\ & 0, \\ & \{ \quad \text{“}/environment/date < 2020-04-12\text{”}, \\ & \quad \text{“}/connection/ip = 129.100.16.66\text{”} \quad \} \\ & \emptyset \\ ) & \end{aligned} \tag{2}$$

In this case, the delegation is constrained with two policies; `/environment/date < 2020-04-12` revokes the delegation if the current date is past April 12th, 2020 and `/connection/ip = 129.100.16.66` only makes this delegation valid if the delegatee is connecting from the IP address 129.100.18.66. A depth value of 0 limits the delegatee from further delegating these attributes onto other users. A null parent (∅) indicates that this is the first level of delegation and that Bob is first delegator in the chain.

3.2 Constraints on Delegatable Attributes

The set of attributes that may be assigned via delegation is not unlimited. There are two major constraints placed on the attributes and values a delegator may pass on to a delegatee. The first constraint is that delegator must have the delegated attribute and corresponding values in their effective attribute set (i.e. the set of attributes directly assigned to the delegator combined with those the delegator inherited from group membership). The second constraint placed on delegatable attributes comes from the new *Can Delegate (CD)* relation added to the HGABAC model (as shown in Fig. 5). The *CD* relation allows a system administrator to directly constrain the set of attributes a user may delegate to a finite list and is defined as follows:

$$\begin{aligned} &\forall cd \in CD : \\ &\qquad cd = (delegator \in U, att\_name \subseteq \{name | (name, type) \in UA\}, max\_depth \in \mathbb{N}_{\leq 0}) \end{aligned} \tag{3}$$

where *delegator* is the unique ID of the user who is permitted to delegate the set of user attributes listed in *att_name*, a list of unique user attribute names from the set of all user attributes (UA). *max_depth* is a positive integer value that limits the value of *deph* used in the *Delegated Attribute Set* such that $depth \leq max\_depth$. A delegator may not select a *depth* larger than *max_depth* when delegating an attribute in the set *att_name*. A *max_depth* of 0 would limit the attributes from being further delegated.

The attributes a delegator may delegate is defined by the *delegatable* function which combines both constraints and maps a user to the set of attributes they may delegate:

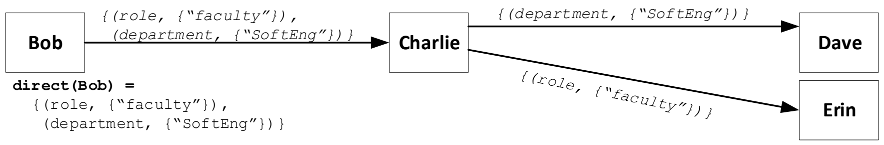

Bob
{(role, {"faculty"}), (department, {"SoftEng"})}
Charlie
{(department, {"SoftEng"})}
Dave
{(role, {"faculty"})}
Erin
direct(Bob) =
{(role, {"faculty"}), (department, {"SoftEng"})}

Fig. 6. Example delegation chain. Bob delegates a *role* and *department* attribute to Charlie, who delegates the *department* attribute on to Dave and *role* attribute to Erin.

$$\begin{aligned} delegatable(u) = \{att\_name | &(att\_name, values) \in effective(u) \\ &\wedge att\_name \in att\_set \\ &\wedge (u, att\_set, depth) \in CD\} \end{aligned} \tag{4}$$

where u is the ID of the delegator and *effective(u)* is the delegator's effective attributes set as defined by HGABAC. The result is an attribute only being delegatable if it is both assigned to the delegator normally (through User *Attribute Assignment* or *User Group Attribute Assignment*) and explicitly permitted via the *Can Delegate* relation.

3.3 Subsequent Delegations & Delegation Chains

In addition to the attributes listed in *delegatable(u)*, users may also further delegate attribute sets they have been delegated so long as the maximum depth has not been reached. If a user, u, wishes to delegate a set of attributes they have been delegated, $das_{old} \in DAS$, they create a new *Delegated Attribute Set*, das_{new}, such that:

$$\begin{aligned} das_{new} = (\quad & delegatee \in U, \\ & u, \\ & att\_set_{new} \subseteq das_{old}.att\_set, \\ & depth < das_{old}.depth, \\ & rule\_set_{new} \supseteq das_{old}.rule\_set, \\ & das_{old} \\ ) & \end{aligned} \tag{5}$$

That is das_{new} must contain the same or a subset of the attributes of das_{old}, must have a depth less than the depth listed in das_{old}, must have a rule set that is more restrictive than das_{old} (i.e. must contain the same rules plus optionally any additional rules) and must list das_{old} as the parent. These conditions ensure that subsequent delegations in a delegation chain are always more restrictive than their parents, the maximum depth is maintained and that attribute sets remain isolated.

An example delegation chain is shown in Fig. 6. In this case, the user Bob is delegating his *role* attribute with the value *"faculty"* and *department* attribute with the value *"SoftEng"* to the user Charlie. This is the same delegation as discussed in Sec. 3.1 and Bob creates the same *DAS* as shown in Equation 2 but with a *depth* of at least 1. This *DAS* is referred to as das_C. The key difference is that Charlie now further delegates a subset of these attributes on to Dave and Erin. In the case of Dave, Charlie delegates just the *department* attribute and in the case of Erin, only the *role* attribute. To accomplish this, the following *DAS*s are created, das_D for Dave and das_E for Erin:

$$\begin{aligned} das_D = (& \\ & Dave, \\ & Charlie, \\ & \{(department, \{\text{“}SoftEng\text{”}\})\}, \\ & 0, \\ & \{\text{“}/environment/date < 2020-04-12\text{”}, \\ & \ \text{“}/connection/ip = 129.100.16.66\text{”} \\ & \ \text{“}/user/age >= 18\text{”}\}, \\ & das_C \\ ) & \end{aligned} \tag{6}$$

$$\begin{aligned} das_E = (& \\ & Erin, \\ & Charlie, \\ & \{(role, \{\text{“}Faculty\text{”}\})\}, \\ & 0, \\ & \{\text{“}/environment/date < 2020-04-12\text{”}, \\ & \ \text{“}/connection/ip = 129.100.16.66\text{”} \\ & \ \text{“}/environment/date < 2020-04-01\text{”}\}, \\ & das_C \\ ) & \end{aligned} \tag{7}$$

das_D delegates the *department* attribute to Dave, but also adds in a new constraint on the delegation, `/user/age >= 18`, which requires that the user of this attribute set must have an *age* attribute in their effective attribute set (*effective(u)*) with a value equal to or greater than 18 for this delegation to be valid. All other constrains from das_C are present in das_D as required by Equation 5. If the *depth* in das_C was greater than 0, and Dave further delegated this attribute set on to another user, the `/user/age >= 18` constraint would have to be maintained, requiring all future delegatees in the chain to also be 18 years or older to use the delegated attributes.

The das_E set delegates the *role* attribute to Erin, but also adds an additional constraint of `/environment/date < 2020-04-01` which invalidates the delegation after April 1st, 2020. It is important to note that this does **not** conflict with the existing rule, `/environment/date < 2020-04-12`, from the parent set, das_C, but further constrains it as all policy rules must evaluate to `TRUE` for the delegation to be valid. In this way, subsequent delegators may tighten constrains on delegations but not loosen them.

Cycles in the delegation chain are permitted but not useful as each child in the chain must have the same or stricter constraints. The impact of such cycles is negligible as delegated attributes are isolated from each other and the user's effective attribute set as only one such set may be activated in a given session as discussed in Sec. 3.4. Cycles are prevented from being infinite in length as the *depth* of each set in the chain must be less than that of the parent and eventually reach 0, preventing further delegation.

3.4 Sessions & Attribute Activation

An important feature of the proposed User-to-User Attribute Delegation model is the isolation of delegated attributes from the user's effective set of attributes as well from other delegated attribute sets. This is accomplished through a modification of HGABAC's session definition. In the original HGABAC model, sessions are defined as a tuple of the form $s = (u \in U, att\_set \subseteq effective(u), con\_atts)$ where u is the user the session belongs to, att_set is the subset of the user's effective attributes being activated for this session and con_atts is the set of connection attributes that describe this session (e.g. IP address, time the session was started, etc.). To support delegation, we update the definition of a session to the following:

$$\begin{aligned} s = (\quad & u \in U, \\ & att\_set_{effective} \subseteq effective(u) \vee att\_set_{delegated} \in \{\ del\_att\_set \mid \\ & \quad das \in DAS \wedge das = (u,\ delegator,\ del\_att\_set,\ depth, rule\_set,\ parent)\}, \\ & con\_atts\ ) \end{aligned} \tag{8}$$

Or more simply put, the activated attribute set in a session may now be one of $att\_set_{effective}$ or $att\_set_{delegated}$ where $att\_set_{effective}$ is any subset of the user's effective attribute set (as per the original HGABAC session definition) and $att\_set_{delegated}$ is one of the delegated attribute sets delegated to the user via the new *Delegated Attribute Set* component. This limits users to either using their normally assigned attributes or **one** of their delegated attributes sets at a time, eliminating or vastly reducing the issues discussed in Sec. 2.2 related to merging delegated attribute sets.

3.5 Revocation

An important feature of the HGAA architecture is maintaining a separation between the Attribute Authorities (AA) which grant attributes to users (via Attribute Certificates (AC)), and the Policy Enforcement Points and Policy Decision Points. This separation provides an important advantage in distributed and federated systems as no communication is required between the AAs and the services their attributes grant access to beyond a user passing on their AC. This, however, raises a number of issues when it comes to revocation. As direct communication between the AAs and other services is optional, an AA's (and by extension it's user's) ability to revoke delegated attributes is limited to the predefined delegation rules in the *rule_set* component of the *DAS*. As is shown in the example *DAS*s (in Equations 2, 6 and 7) these rules may be any valid HGPL policy. If all policies evaluate to *TRUE* the delegation is valid, if the result is *FALSE* or *UNDEF* it is considered to be revoked.

In cases of delegation chains (as shown in Fig. 6), if any policy in a *rule_set* is invalidated all subsequent delegations in the chain are also revoked. This is in part a consequence of all *rule_set*s in subsequent delegations being required to be a superset of the parent *rule_set* as is stated in Equation 5 (i.e. they must contain at least all policies in the parent rule set), but it is further required that each user in the chain satisfies the policies in their own *rule_set*. For example, if the policy */user/age >= 18* is made a condition of a delegation from Bob to Charlie and Charlie subsequently delegates the attributes on to Dave, both Charlie and Dave must have their own `age` attribute that has a value of 18 or greater. The value of environment and administrator attributes are determined based on their current value at the Policy Decision Point. As the value of connection attributes for parents in the delegation chain may be unknown or undefined, how they are evaluated is left as an implementation decision (i.e. conditions involving connection attributes of parent users can be assumed to be *TRUE*, *UNDEF*, based on the last known values, or based on their values at the time of delegation).

Formally, we define the recursive function *active* which takes a Delegatable Attribute Set, *das*, and returns *TRUE* if the delegation is active (not revoked) and *FALSE* if the delegation is considered to be revoked.

$$active(das) = \begin{cases} active(das_{parent}) & if\ das.parent \neq \emptyset \\ \quad \bigwedge\ das.depth < das.parent.depth & \\ \quad \bigwedge\ das.depth \geq 0 & \\ \quad \bigwedge\ das.att\_set \subseteq delegatable(das.delegator) & \\ \quad \bigwedge\ das.rule\_set\ \supseteq das.parent.rule\_set & \\ \quad \bigwedge\ \forall rule \in das.rule\_set : valid(rule,\ das.delegatee) = TRUE & \\ & \\ das_{att\_set} \subseteq delegatable(das.delegator) & if\ das.parent = \emptyset \\ \quad \bigwedge\ \forall rule \in das.rule\_set : valid(rule,\ das.delegatee) = TRUE & \end{cases} \tag{9}$$

where *valid* is a HGABAC function which takes an HGPL policy and a user and returns *TRUE* if the user satisfies that policy for the current value of that user's attributes (including connection attributes) and the current state of the system (environment attributes and administrator attributes), *FALSE* if the policy is violated and *UNDEF* if the policy cannot currently be evaluated.

A secondary means of revocation is possible through HGAA's optional AC revocation lists. In HGAA, each AA may publish a revocation list that includes the serial number of any revoked AC issued by the authority. Policy Decision Points may optionally request this list either on demand or periodically depending on the nature of their service and if communication with the AA is possible. In the delegation framework detailed in the next section (Sec. 4), DAS are represented as special Delegated Attribute Certificates (DAC). These DACs may be revoked by the same mechanism. If a revoked DAC is part of a delegation chain, all subsequent delegations are also revoked.

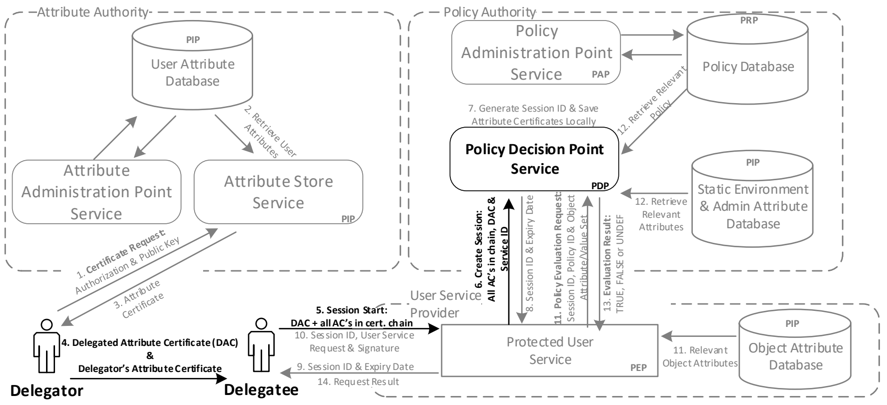

Attribute Authority
PIP
User Attribute Database
2. Retrieve User Attributes
Attribute Administration Point Service
Attribute Store Service
PIP
1. Certificate Request: Authorization & Public Key
3. Attribute Certificate
4. Delegated Attribute Certificate (DAC) & Delegator's Attribute Certificate
Delegator
Delegatee
5. Session Start: DAC + all AC's in cert. chain
10. Session ID, User Service Request & Signature
9. Session ID & Expiry Date
14. Request Result
Policy Authority
Policy Administration Point Service
PAP
PRP
Policy Database
12. Retrieve Relevant Policy
7. Generate Session ID & Save Attribute Certificates Locally
Policy Decision Point Service
PDP
PIP
Static Environment & Admin Attribute Database
12. Retrieve Relevant Attributes
6. Create Session: All AC's in chain, DAC & Service ID
8. Session ID & Expiry Date
11. Policy Evaluation Request: Session ID, Policy ID & Object Attribute/Value Set
13. Evaluation Result: TRUE, FALSE or UNDEF
User Service Provider
Protected User Service
PEP
11. Relevant Object Attributes
PIP
Object Attribute Database

Fig. 7. Updated HGAA protocol to support delegation step. New and modified components shown in bold dark black, pre-existing HGAA components shown in light grey.

4 Delegation Framework

The proceeding section (Sec. 3) laid out the theoretical delegation model and extensions to HGABAC to incorporate User-To-User Attribute Delegation. This section seeks to provide more practical details for how this delegation model may be implemented by extending HGAA to create a supporting delegation framework. Two key aspects of HGAA need to be expanded; the Attribute Certificate (AC) format to include delegation extensions and rules (detailed in Sec. 4.2), and the communication steps between users and services to provide a full certificate chain (detailed in Sec. 4.1).

4.1 Protocol Additions

In HGAA, users are issued an AC from an AA's Attribute Store Service. This document provides proof of a user's attributes in a cryptographically signed document as well as providing a mechanism for single sign-on and authentication with remote services. The AC format includes a listing of the user's attributes, details of the issuing authority, a public key assigned to the user, a range of dates for which the certificate is valid and a number of areas reserved for future extensions. User's prove ownership of an AC via a private key corresponding to the public key embedded in the AC. As the certificate is signed and contains all information about the user to base policy decision on, direct communication between the service being accessed and the AA is not required.

In the original architecture, after being issued an AC, users use the certificate to make requests on services. To support delegation, an additional delegation step is needed (as shown in Fig. 7 as step 4). Rather than directly querying services, a user may now delegate all or a subset of the attributes in their AC to a third party by issuing a new AC called a Delegated Attribute Certificate (DAC). The DAC is identical to an AC issued by an AA but lists the delegator as the issuer and signer (rather than an AA), and the delegatee as the holder. The extensions to the AC format to support DACs and delegation, detailed in Sec. 4.2, enable the delegator to include delegation rules to trigger revocation (as discussed in Sec. 3.5), set a maximum delegation depth for subsequent delegations and select what subset of the attributes in their AC will be contained in the DAC (and delegated to the delegatee).

To complete the delegation, the delegator sends their AC and the new DAC to the delegatee. The delegatee validates both by checking the following:

1. The original components of the AC are valid as described in [5] (i.e. correctly signed by the AA, that the AC has not expired, etc.)
2. The *ACHolder* from the AC is the *ACIssuer* in the DAC (same UID, key, etc.).
3. The *ACHolder* given in the DAC is the delegatee (correct UID, public key, etc.).
4. All attributes listed in the DAC are also found in the AC and have a *maxDepth* greater than zero in the AC.
5. All attributes in the DAC have the delegator listed as the delegator and a *maxDepth* less than or equal to the *maxDepth* in the DAC for that attribute.
6. The *ACRevocationRules* in the DAC are the same or stricter than in the AC.
7. The *ACDelegationRules* in the DAC are the same or stricter than in the AC.
8. The overall delegation *depth* in the DAC is less than the delegation *depth* in the AC and greater than or equal to 0.
9. That the delegation has not been revoked (i.e. all delegation rules return *TRUE*).
10. The DAC is signed by the delegator with the public key listed in the *ACHolder* sequence of the AC and the *ACIssuer* sequence of the DAC.

These checks enforce the rules on DASs described in Sec. 3.1 and ensure the delegation has not been revoked (as per Sec. 3.5). If the AC and DAC are valid, the delegatee may make requests upon services by sending both the AC and DAC with their request. The remainder of the HGAA protocol remains the same, but with the DAC being sent with the AC in steps 5 and 6 (Fig. 7). The Policy Decision Point also makes the same checks (as listed above) on the DAC when validating the deletagee's attributes.

Subsequent delegations by the delegatee, to further delegatees, are supported. In such cases, the delegatee becomes the delegator and issues a new DAC using the processes previously described (their existing DAC becoming the AC and they become the issuer of the new DAC). This creates a chain of certificates leading back to the AA, each certificate being signed by the parent delegator. This process is shown in the Low Level Certificate Chain Diagram found in Appx. A. To allow services and the Policy Decision Point to verify subsequent delegations, each certificate in the chain is included with the first request upon a service and each certificate is validated.

4.2 Attribute Certificate Delegation Extensions

To incorporate our delegation model and updated HGAA protocol, several extensions to the AC format are required (described in Listing 1.1). The *Attribute* sequence is extended to include a *maxDepth* and *delegatorUniqueIdentifier* value for each attribute in the certificate. *delegatorUniqueIdentifier* states the ID of the original delegator (first in the chain) of the attribute or no value if not delegated. *maxDepth* corresponds to the *Can Delegate* relation (defined in Sec. 3.2) and has a value equal to 0 if this attribute cannot be delegated, 255 if there is no limit on the delegation depth or some value between 1 and 254 equal to the maximum depth allowed for this specific attribute.

Delegation rules from the DAS (defined in Sec. 3.1) are encoded in a new *DACDelegationRule* sequence which contains a HGPLv2 policy for each rule. The *depth* value from the DAS is included in a new instance of the *ACExtension* sequence in addition to a record of the original AA and the serial number of each certificate in the chain.

Listing 1.1. Updates to the AC format to support Atrtibute Delegation writen in ASN.1 notation. Bold text indicate addtions. Only updated sequences are shown.

```
Attribute ::= SEQUENCE {
        attributeID       OBJECT IDENTIFIER,
        attributeType     OBJECT IDENTIFIER,
        attributeValue    ANY DEFINED BY attributeType OPTIONAL,
        attributeName     VisibleString OPTIONAL,
        maxDepth INTEGER(0..255),
        delegatorUniqueIdentifier OBJECT IDENTIFIER OPTIONAL,
}

ACDelegationRules ::= SEQUENCE {
        SEQUENCE OF DACDelegationRule
}

DACDelegationRule ::= SEQUENCE {
        HGPLv2Policy VisibleString
}

– One instance of ACExtension with the following values
UToUAttDelv1 ACExtension ::= SEQUENCE {
        extensionID ”ext:UToUAttDelv1”,
        depth INTEGER(0..254),
        rootAuthorityUniqueIdentifier OBJECT IDENTIFIER,
        SEQUENCE OF DACCertificateSerial
}

DACCertificateSerial ::= SEQUENCE {
        certificateSerial INTEGER
}
```

The extended AC is kept backwards compatible with the original AC format by only updating sections marked for future extension. The changes have a minimal impact on the certificate size, adding at worst $3+U$ bytes per attribute (where U is the size of the largest delegator ID), $2*P$ bytes per delegation rule (where P is the maximum length of a HGPL policy), and $1+S$ bytes per certificate in the chain (where S is the serial number size in bytes). A byte level representation of the new AC is found in Appx. B.

5 Conclusions & Future Work

We have introduced the first model of User-to-User Attribute Delegation as well as a supporting architecture to aid implementation. Extensions to the HGABAC model (Sec. 3) add relations for authorizing what attributes can be delegated (*Can Delegate*) and to what depth. A new access control element, the *Delegated Attribute Set*, is added for representing current delegations in the system and the restrictions placed on them. Delegated attributes are kept isolated to prevent issues with Attribute Delegation, including user collusion and unexpected side effects on policy evaluations.

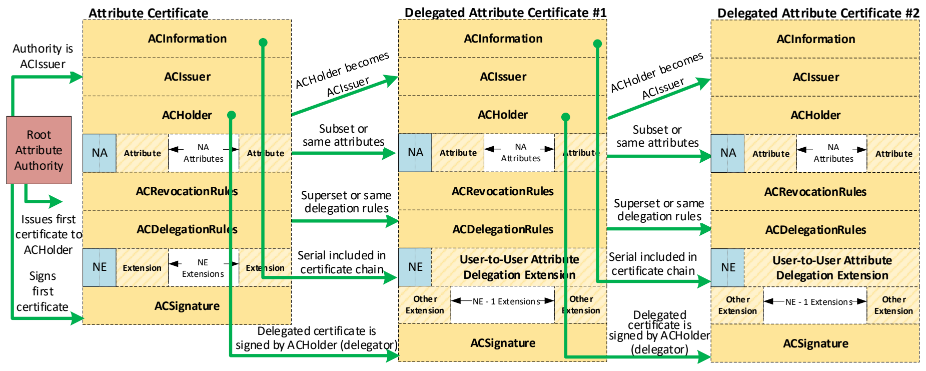
Updates to the HGAA protocol and AC format have been made (Sec. 4) to support the extended HGABAC model. These changes to the AC format are minimal in size, scaling with the number of attributes, delegation rules, and certificates in the chain. As changes have only been made to sequences marked for future expansion, the extended AC format remains backwards compatible with the original HGAA AC. Care has been given to ensure that delegation is preformed in an *“off-line”* manner, without the need to contact a third party, to maintain the distributed nature of HGABAC and HGAA. However, support for *“off-line”* delegation comes at the cost of revocation flexibility and limits the possibility for real time revocation invoked by the delegator. To combat this, HGPL policies are used to embed delegation rules that trigger revocation.

This work is part of an ongoing effort towards introducing delegation to ABAC and directions for future work will follow this path. To date, models for User-To-User Attribute and User-to-Attribute Group Membership [4] Delegation have been completed. The next steps will involve creating models for and implementing the remaining strategies so they can be fully explored, validated, evaluated and compared. Directions for the User-To-User Attribute Delegation model include exploring the use of a *"Can Receive"* relation for users in place of *"Can Delegate"* for attributes and experimenting with adding constraints that prevent specified users form being delegated a restricted attribute (e.g. to prevent certain users from stratifying a policy via delegation). Such *"Can Receive"* relations have been used in RBAC models[2] and were shown to add flexibility. Work is needed to see if the same will hold true for ABAC. Finally, a more thorough evaluation of our delegation model is planned that will involve both formal validation (safety analysis) and experimental evaluation (reference implementation).

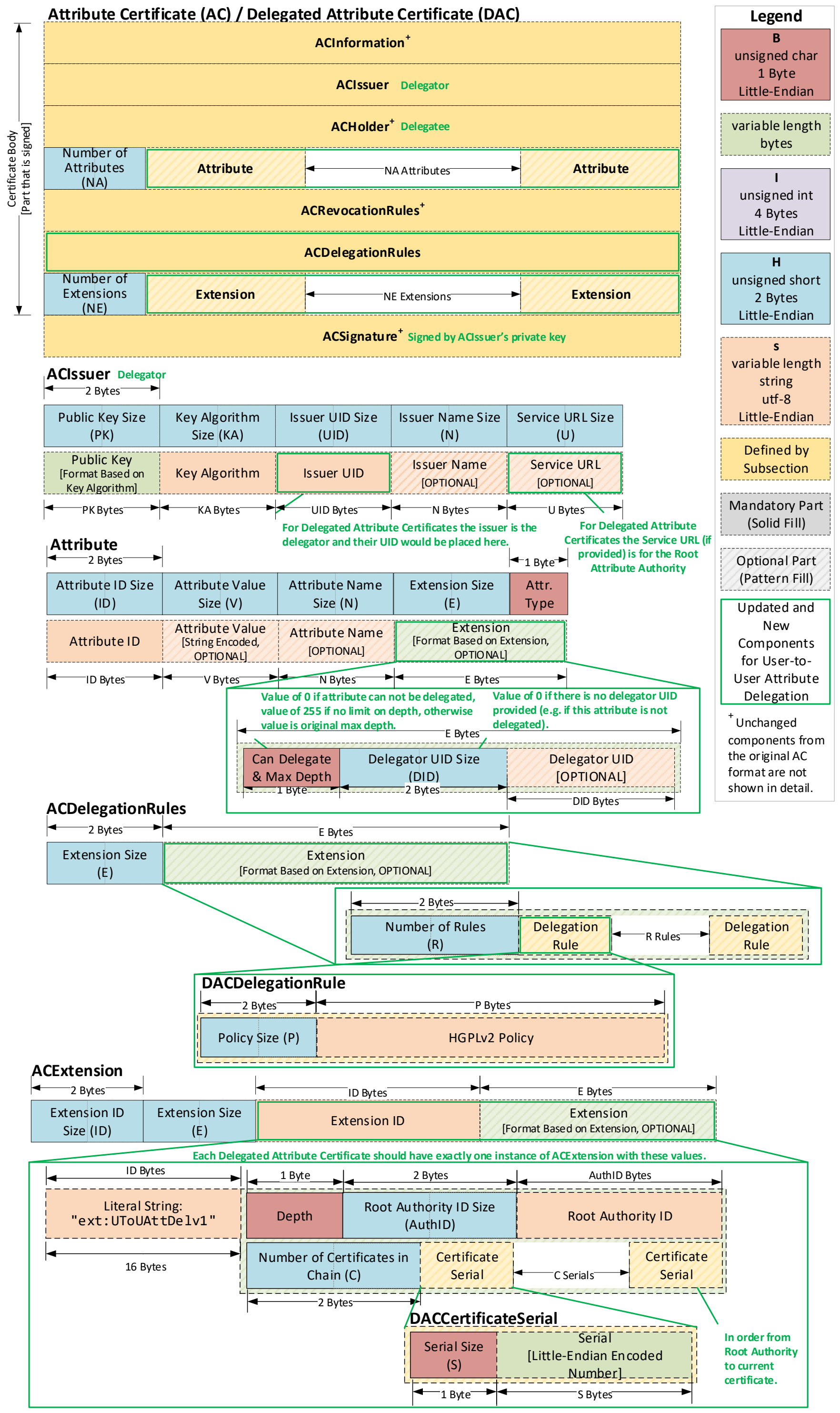
References

1. Anderson, A., Nadalin, A., Parducci, B., et al.: eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS (2003)
2. Crampton, J., Khambhammettu, H.: Delegation in role-based access control. International Journal of Information Security **7**(2), 123–136 (2008)
3. Rostad, L., Edsberg, O.: A study of access control requirements for healthcare systems based on audit trails from access logs. In: 22nd Annual Computer Security Applications Conference (ACSAC'06). pp. 175–186. IEEE (2006)
4. Sabahein, K., Reithel, B., Wang, F.: Incorporating delegation into ABAC: Healthcare information system use case. In: Proceedings of the International Conference on Security and Management (SAM). pp. 291–297 (2018)
5. Servos, D., Osborn, S.L.: HGABAC: Towards a formal model of hierarchical attribute-based access control. In: International Symposium on Foundations and Practice of Security. pp. 187–204. Springer (2014)
6. Servos, D., Osborn, S.L.: Strategies for incorporating delegation into attribute-based access control (ABAC). In: International Symposium on Foundations and Practice of Security. pp. 320–328. Springer (2016)
7. Servos, D., Osborn, S.L.: Current research and open problems in attribute-based access control. ACM Computing Surveys (CSUR) **49**(4), 65 (2017)
8. Servos, D., Osborn, S.L.: HGAA: An architecture to support hierarchical group and attribute-based access control. In: Proceedings of the Third ACM Workshop on Attribute-Based Access Control. pp. 1–12 (2018)
9. Wang, L., Wijesekera, D., Jajodia, S.: A logic-based framework for attribute based access control. In: Proceedings of the 2004 ACM workshop on Formal methods in security engineering. pp. 45–55 (2004)

A Low Level Certificate Chain Diagram


Attribute Certificate
Delegated Attribute Certificate #1
Delegated Attribute Certificate #2
ACInformation
ACIssuer
ACHolder
NA
Attribute
NA Attributes
ACRevocationRules
ACDelegationRules
NE
Extension
NE Extensions
ACSignature
User-to-User Attribute Delegation Extension
Other Extension
NE - 1 Extensions
Authority is ACIssuer
Root Attribute Authority
Issues first certificate to ACHolder
Signs first certificate
ACHolder becomes ACIssuer
Subset or same attributes
Superset or same delegation rules
Serial included in certificate chain
Delegated certificate is signed by ACHolder (delegator)

B Low Level Extended Attribute Certificate Diagram


Attribute Certificate (AC) / Delegated Attribute Certificate (DAC)
Certificate Body [Part that is signed]
ACInformation+
ACIssuer Delegator
ACHolder+ Delegatee
Number of Attributes (NA)
Attribute
NA Attributes
Attribute
ACRevocationRules+
ACDelegationRules
Number of Extensions (NE)
Extension
NE Extensions
Extension
ACSignature+ Signed by ACIssuer's private key
Legend
B unsigned char 1 Byte Little-Endian
variable length bytes
I unsigned int 4 Bytes Little-Endian
H unsigned short 2 Bytes Little-Endian
s variable length string utf-8 Little-Endian
Defined by Subsection
Mandatory Part (Solid Fill)
Optional Part (Pattern Fill)
Updated and New Components for User-to-User Attribute Delegation
+ Unchanged components from the original AC format are not shown in detail.
ACIssuer Delegator
2 Bytes
Public Key Size (PK)
Key Algorithm Size (KA)
Issuer UID Size (UID)
Issuer Name Size (N)
Service URL Size (U)
Public Key [Format Based on Key Algorithm]
Key Algorithm
Issuer UID
Issuer Name [OPTIONAL]
Service URL [OPTIONAL]
PK Bytes
KA Bytes
UID Bytes
N Bytes
U Bytes
For Delegated Attribute Certificates the issuer is the delegator and their UID would be placed here.
For Delegated Attribute Certificates the Service URL (if provided) is for the Root Attribute Authority
Attribute
2 Bytes
1 Byte
Attribute ID Size (ID)
Attribute Value Size (V)
Attribute Name Size (N)
Extension Size (E)
Attr. Type
Attribute ID
Attribute Value [String Encoded, OPTIONAL]
Attribute Name [OPTIONAL]
Extension [Format Based on Extension, OPTIONAL]
ID Bytes
V Bytes
N Bytes
E Bytes
Value of 0 if attribute can not be delegated, value of 255 if no limit on depth, otherwise value is original max depth.
Value of 0 if there is no delegator UID provided (e.g. if this attribute is not delegated).
E Bytes
Can Delegate & Max Depth
Delegator UID Size (DID)
Delegator UID [OPTIONAL]
1 Byte
2 Bytes
DID Bytes
ACDelegationRules
2 Bytes
E Bytes
Extension Size (E)
Extension [Format Based on Extension, OPTIONAL]
2 Bytes
Number of Rules (R)
Delegation Rule
R Rules
Delegation Rule
DACDelegationRule
2 Bytes
P Bytes
Policy Size (P)
HGPLv2 Policy
ACExtension
2 Bytes
ID Bytes
E Bytes
Extension ID Size (ID)
Extension Size (E)
Extension ID
Extension [Format Based on Extension, OPTIONAL]
Each Delegated Attribute Certificate should have exactly one instance of ACExtension with these values.
ID Bytes
1 Byte
2 Bytes
AuthID Bytes
Literal String: "ext:UToUAttDelv1"
Depth
Root Authority ID Size (AuthID)
Root Authority ID
16 Bytes
Number of Certificates in Chain (C)
Certificate Serial
C Serials
Certificate Serial
2 Bytes
DACCertificateSerial
Serial Size (S)
Serial [Little-Endian Encoded Number]
1 Byte
S Bytes
In order from Root Authority to current certificate.