Policy Based Access Control (PBAC) for Diverse DoD Security Domains

Brad J. Cox, Ph.D.
Technica Corporation
brad.cox@technicacorp.com
March 2011

Abstract

This paper describes a reference implementation of policy enforcement and decision points designed to support source selection and evaluation for a fine-grained, centralized Policy-Based Access Control (PBAC) management process. This process yields executable policies to govern access to diverse, cloud-based Department of Defense (DoD) resources..

The first part of this paper describes the reference implementation as delivered in December 2010, which is based on Sun's XACML interpreter. The second part describes a subsequent implementation that features direct compilation of XACML to Java. The compiled implementation generates Java classes that are considerably easier to read, understand, and debug than XACML and that runs many times faster than the interpreted implementation.

Background

The Department of Defense is undergoing a transformation in information management to meet new operational requirements such as[1]:

- Faster mission tempo with users of diverse citizenship, organizational affiliation, operational roles, privacy needs, and security clearances. Seamless operation across sovereign boundaries while protecting sovereign assets.
- Unanticipated users; i.e. users with verifiable certificates but without entries in DoD attribute stores. For example, members of allied forces.
- Dynamically defined privileges. Risk-adaptive capabilities. Fine-grained authorization. Appropriate balance between need to know and need to share.

These requirements occur in the context of other trends such as cloud-based deployment and centralized access control. A policy management process generates eXtensible Access Control Markup Language (XACML) policies from formal policy documents. XACML is an Oasis standard for expressing access control policy. The execution engine (PDS) executes the XACML policies to determine who can access which resources. The management process harmonizes the often-conflicting policies of the diverse domains that may have an interest in which users can access which resources. The management process and execution engine work together to provide Policy Based Access Control (PBAC) in which access is defined and enforced centrally according to formal government policy, not by the local administrators of each application as today.

This paper presents early results of an effort to provide an open source reference implementation that our customer can use to explore policy management and enforcement[2]. The paper is in two parts. The first describes how the reference implementation addresses the requirement that it be open source and hosted on forge.mil. The second describes compilation of XACML to Java as an alternative to Sun's interpreted XACML engine, which the reference implementation uses now. The generated Java code is easier to read, understand, and debug, and many times faster than the interpreted implementation.

This document does not discuss security although that was a major part of our work. This part of the work involved choosing a SOA platform that implements the required standards and configuring it to meet the requirements. This also doesn't describe the portal Technica developed to demonstrate the reference implementation.

Reference Implementation

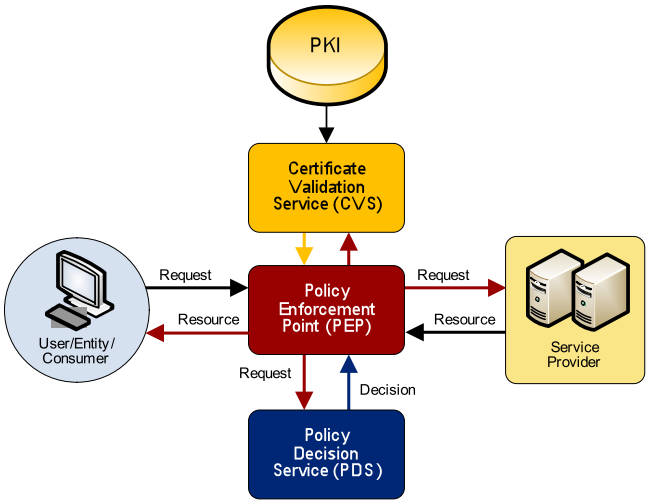

Figure 1. The PEP enforces decisions made by the PDS as to whether to permit or deny requests.

Figure 1 shows the core components of the reference implementation. The Policy Enforcement Point (PEP) and Policy Decision Service (PDS) were developed for this project and delivered to forge.mil in late 2010. The figure does not show the resources that the PEP and PDS are guarding or the policy

and attribute stores that they base decisions on. Nor does it show the infrastructure for maintaining and administering the attribute and policy stores. In time, these will have administration points (PAPs and PIPs) plus other tools to support the PBAC policy and attribute management processes.

Guarding service provider resources is the role of the PEPs. These authenticate each request to ensure that the sender has valid credentials. If so, it sends the authenticated subject identifier, the requested URL, the requested action (get, post, etc.), and environmental attributes (time, date, etc.) to the PDS, which returns one of the four possible decisions defined by the XACML standard (Permit, Deny, Not Applicable, or Indeterminate).

In general, there are many PEPs and service providers. The goal is to support them with a single, centrally administered, generic PDS (replicated as required) that makes access control decisions for diverse security domains. The policy management process supplies the algorithms the PDS uses to make decisions as XACML files. The XACML engine bases its decisions on attribute values that the PDS supplies as a context record whose format is defined by the XACML standard.

The XACML standard defines the context format but not the attribute names that it contains. PEPs might send the validated subject ID, the resource URL, and the requested action, leaving the PDS to retrieve other needed attributes. Or they might include other attributes, leaving the PDS to map those provided to those the policy needs. And the standard does not specify the attributes a policy uses, while privacy and other concerns prevent simplistic "solutions" such as sending everything "just in case". Finally, diverse domains are often unable to agree on a uniform set of attributes to base their policies on.

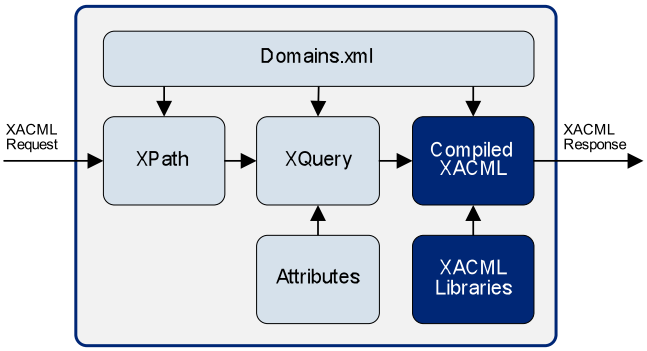

Figure 2. The PDS uses a three-step process for each decision. Each step is specified by entries in the Domains.xml configuration file. This is automatically reloaded when it changes.

These observations suggest that the PDS needs a way for each domain to specify how to build the context record from the available attributes. This is specified by a Domains.xml file which specifies an XQUERY script. The script specifies the attribute stores each domain trusts, retrieves attribute values, and builds the context record[1] before sending the result to the XACML engine for execution. The three-step process is shown in Figure 2.

The PDS provides open source implementations of the three standards in the middle row. The PDS loads the Domains.xml file from the class path at initialization time and reloads it when this file, or the files it references, changes. Domains.xml contains a list of entries, one for each of the domains that the PDS manages plus a default entry that handles requests that are not matched by the other entries. Each entry specifies the three elements in the middle row. The Domains.xml file is owned and managed by the PDS owner while the XQUERY and XACML files it references are owned and managed by the security domains.

The PDS determines the domain that holds the requested resource by applying each entry's XPATH statement to the request. It is an error if more than one domain matches and a default entry is supported for any requests that are not matched. The matching XQUERY script then transforms the request to form the XACML context. This generally involves retrieving attribute values via Java extension functions developed for this project.

The PDS loads and compiles the XQUERY script and XACML policies when they change so that their load and compile times do not affect each request. Attribute store connections are established at this time and cached for reuse.

The need to harmonize conflicting policies across diverse security domains is what distinguishes PBAC from ordinary access control systems. Diversity implies a need to draw attributes from heterogeneous attribute stores and to harmonize conflicting policies. How and when this harmonization occurs is still being decided. One plausible outcome is that policies are developed, tested, harmonized, certified, and signed entirely inside the policy management process, and the online PDS simply reports them as faults in the management process.

Compiling XACML to Java Source

The usual reason for replacing an interpreter with a compiler is performance, but this was not true in this case. The compiler arose from concerns over XACML's verboseness and complexity, and concern about how individuals engaged in the policy management process would build confidence that complex policies behave as they expect.

The goal was also not to replace XACML with Java. DoD chose XACML as its policy interchange language. The goal was to help individuals understand XACML within a complex policy management process. A bridge from XACML to Java allows individual participants to use Java IDEs to support change notification and management, development, testing,

[1] This description reflects our understanding of the standard while developing the reference implementation. Subsequent work on the compiler revealed that XACML allows implementations to retrieve "implicit" values directly from attribute stores. These are values that are referenced in a policy but not defined in the context record.

deconfliction, and certification. Java is to XACML as machine language is to C, a machine-oriented representation that in this case is easier to comprehend than the original XACML.

For example, the XACML standard[3] provides an example policy, "Only allow logins between 9am and 5pm". This one line of text expands into a whole page of XACML text. Our first thought was to use a more approachable language such as Attempto Controlled English (ACE), but this was unable to support the deeply nested conditionals[2] that dominate the rules in Figure 3.

```
public boolean rawRuleConditionDecision()
{
    boolean isRuleMatched = (request.subject.CountryOfCitizenship.oneAndOnly().equa
    &&newXStringBag(new XString("USN"), new
XString("USMC")).contains(request.subject.OrganizationID.oneAndOnly())
    &&newXStringBag(new XString("2201"), new
XString("2207")).contains(request.subject.DutyOccupationalCode.oneAndOnly())
    &&!(newXStringBag(new XString("E1"), new XString("E2"), new XString("E3"), new
XString("E5"), new XString("E6")).contains(request.subject.PayGrade.oneAndOnly()))
    &&newXStringBag(new XString("A"), new
XString("C")).contains(request.subject.PersonnelCategoryCode.oneAndOnly())
    &&(request.subject.GeographicSubRegion.contains(new XString("Country Yellow"))
    ||request.subject.GeographicSubRegion.contains(new XString("Fifth Fleet")))
    &&(request.subject.ExtendedGroup.contains(new XString("CTF 153"))
    ||request.subject.ExtendedGroup.contains(new XString("Fifth Fleet"))
    ||request.subject.ExtendedGroup.contains(new XString("Second Fleet"))
    ||request.subject.ExtendedGroup.contains(new XString("Third Fleet"))
    ||request.subject.ExtendedGroup.contains(new XString("Fourth Fleet"))
    ||request.subject.ExtendedGroup.contains(new XString("Sixth Fleet"))
    ||request.subject.ExtendedGroup.contains(new XString("Seventh Fleet"))
    ||request.subject.ExtendedGroup.contains(new XString("CSG1"))
    ||request.subject.ExtendedGroup.contains(new XString("CSG2"))));
    return isRuleMatched;
}
```

Figure 3. Java code generated by the XACML compiler for one of the three reference implementation test cases.

Figure 3 shows the Java source that the compiler produces for one of four rules of the reference implementation test case. Notice the deeply nested and/or conditions. Space does not allow direct comparison (the XACML is 12 pages long), but the generated code is considerably more concise and readable.

The compiler was developed to this stage in about a month. Rapid progress was possible because XACML's policy schema could be converted into a serviceable expression tree by JAX-P[3]. Emitting the Java source then becomes a matter of walking this tree, emitting code at each step to call the appropriate function from the library defined in the XACML standard.

The compiler emits a Java class for each XACML Policy plus internal classes for each Rule. PolicySets are handled analogously, with internal classes for each Policy plus Rules within these. The policy class constructors are receive the context record in the form of a DOM tree and immediately use it to populate subject, resource, action and environment variables as instances of the appropriate XACML library Bag types. The tree is also retained in an instance variable to support any XPATH functions that the policy might use.

Figure 4 is an example of the code that the compiler emits to support the policy and policy set constructor methods. The policy constructor receives the context as a DOM tree produced by JAX-P. The constructor initializes policy instance methods by constructing instances of the generated classes shown in the figure. Decisions are rendered by calling the instance's evaluateQuery method. The compiler generates this method to call the combining algorithm specified in the XACML source. Policies inherit combining algorithm definitions to minimize the size of the generated code. The combining algorithms invoke a PolicyTarget method in each Policy class and RuleTarget and RuleCondition methods in each internal Rule class. The compiler generates these methods from the XACML source. These methods return booleans to represent Permit and Deny or throw runtime exceptions to represent Indeterminate and NotApplicable responses. The combining algorithms catch these exceptions to support the four XACML decision types.

```
public static class Request extends AbstractRequest
{
    final Subject subject;
    final Resource resource;
    final Action action;
    final Environment environment;

    public Request(OMElement request, AttributeStore[] attributeStores)
    {
        super(request);
        this.subject = new Subject(request.getChildrenWithLocalName("Subject"), attributeStores);
        this.resource = new Resource(request.getChildrenWithLocalName("Resource"), attributeStores);
        this.action = new Action(request.getChildrenWithLocalName("Action"), attributeStores);
        this.environment = new Environment(request.getChildrenWithLocalName("Environment"),
attributeStores);
    }
    private static class Subject
    {
        XString.Bag PersonnelCategoryCode = new XString.Bag("PersonnelCategoryCode");
        XString.Bag ExtendedGroup = new XString.Bag("ExtendedGroup");
        XString.Bag CountryOfCitizenship = new XString.Bag("CountryOfCitizenship");
        XString.Bag PayGrade = new XString.Bag("PayGrade");
        XString.Bag GeographicSubRegion = new XString.Bag("GeographicSubRegion");
        XString.Bag OrganizationID = new XString.Bag("OrganizationID");
        XString.Bag Clearance = new XString.Bag("Clearance");
        XString.Bag DutyOccupationalCode = new XString.Bag("DutyOccupationalCode");

        Subject(Iterator subjectElmts, AttributeStore[] attributeStores)
        {
            populateSubject(subjectElmts, new AbstractBag[]
{PersonnelCategoryCode,ExtendedGroup,CountryOfCitizenship,PayGrade,GeographicSubRegion,OrganizationI
rance,DutyOccupationalCode}, attributeStores);
        }
    }
    // Similarly for Resource, Action and Environment request types
```

Figure 4. Representative classes that show how attributes from the request are represented as Bag types from the XACML library.

XACML interpreters are black boxes. They are loaded with a policy, passed a request, and return a decision with no explanation of how that response was derived. Finding policy errors is hard when the only options are reading the logs and stepping through an interpreter's internal logic with a debugger. Compiled Java, by contrast, is easily handled by Java debuggers. The user is then working directly with the policy itself, not the internal workings of an interpreter. Breakpoints can easily stop execution and values inspected at any point.

Another month and a half was then spent building an XACML run-time library. This library contains approximately 20

---

[2] ACE does allow commas to be used to invert the precedence of and-or expressions. This helps with correctness but not with the readability of long nested expressions.

[3] Java Architecture for XML Binding (JAX-B) compiles XML schema into Java classes. It is based on Java Architecture for XML Parsing (JAX-P) for XML parsing. The compiler uses JAX-P for context records so that the DOM tree will be available to XPATH expressions. Otherwise, the compiler uses JAX-B types for its expression tree. Mappings from JAX-B to Java expressions is held in a spreadsheet that can be easily extended to support other languages. For example, the Attempto Controlled English experiment used another column in this table, since deleted.

XACML attribute types plus Bag types for each. The compiler and library were then tested for compliance with the Oasis Conformance Test Suite. This suite consists of 400 triplets, each consisting of a sample request, a policy or policy set, and the expected response. The compiler and library currently passes all but 10 of the 400 tests. The outstanding issues concern features that are not clearly defined in the standard or where the tests seem to diverge from the standard.

Compiled XACML Performance

Performance was never a goal of this project. But to get some indication of performance for this paper, the compiled and interpreted implementations were packaged as a pair of offline applications that could be compared side by side. Each application accepts a list of request, policy, and expected response triplets and produces a spreadsheet of computed responses and execution times as shown in Figure 5. Since expected and observed responses are the same in all cases, these columns were omitted.

			Interpreted				Compiled				I/C		
Request			Rsp	Req	Pol	Run	Rsp	Req	Pol	Run	Run	P+R	R+P+R
Bogus	Fleet		0.3	0.4	66.9	19.0	0.6	0.5	74.9	1.0	19.0	1.1	1.1
Bogus	Fleet	DutyRoster	0.5	0.5	94.3	16.5	0.7	0.6	63.8	0.2	105.2	1.7	1.7
Bogus	Fleet	Plans	0.4	0.5	52.3	11.6	0.8	30.4	38.5	0.0	320.9	1.7	0.9
BrianWilson	Fleet		1.3	0.6	43.8	17.6	8.2	1.4	53.8	0.1	247.8	1.1	1.1
BrianWilson	Fleet	DutyRoster	0.4	0.5	59.8	17.6	0.6	0.5	31.6	0.4	49.0	2.4	2.4
BrianWilson	Fleet	Plans	0.4	0.6	13.3	7.6	0.7	0.6	69.0	0.0	321.4	0.3	0.3
DavidBurns	Fleet		0.4	0.7	28.8	27.5	15.2	1.2	68.0	0.2	168.7	0.8	0.8
DavidBurns	Fleet	DutyRoster	0.5	4.8	9.1	19.9	0.7	0.7	14.3	0.1	280.0	2.0	2.2
DavidBurns	Fleet	Plans	0.4	6.1	8.4	12.4	0.5	0.8	8.8	0.0	541.3	2.3	2.8
FrankLincoln	Fleet		0.4	0.7	7.8	19.6	0.6	0.7	16.2	0.0	463.7	1.7	1.7
FrankLincoln	Fleet	DutyRoster	0.9	1.0	19.4	21.9	0.5	0.8	7.6	0.0	662.9	5.4	5.0
FrankLincoln	Fleet	Plans	0.4	113.2	13.5	4.7	0.5	2.7	22.9	0.1	42.1	0.8	5.1
JohnWalker	Fleet		0.5	13.8	7.2	10.6	0.5	3.9	44.2	0.0	385.5	0.4	0.7
JohnWalker	Fleet	DutyRoster	0.4	2.8	30.4	11.6	0.5	0.8	11.9	0.4	25.7	3.4	3.4
JohnWalker	Fleet	Plans	0.4	0.7	6.2	5.5	0.6	7.5	7.8	0.0	402.7	1.5	0.8
PaulJones	Fleet		9.7	33.6	7.3	18.0	0.4	6.3	20.4	0.5	35.8	1.2	2.2
PaulJones	Fleet	DutyRoster	0.4	25.2	6.5	18.4	45.3	18.1	22.6	0.2	79.8	1.1	1.2
PaulJones	Fleet	Plans	0.4	13.8	14.3	5.2	0.4	0.8	30.7	0.0	406.6	0.6	1.1
PhillipDaniels	Fleet		0.5	6.0	6.5	11.1	0.5	5.8	14.5	0.0	420.2	1.2	1.2
PhillipDaniels	Fleet	DutyRoster	0.4	5.9	6.6	12.9	0.5	3.6	149.8	0.7	17.3	0.1	0.2
PhillipDaniels	Fleet	Plans	5.5	42.2	6.9	11.8	0.4	0.7	3.5	0.0	674.2	5.3	14.4
RickFoster	Fleet		5.9	13.2	7.0	15.3	0.5	7.6	3.4	0.0	576.6	6.5	3.2
RickFoster	Fleet	DutyRoster	0.4	0.8	6.9	14.2	0.4	1.0	46.6	0.1	96.0	0.5	0.5
RickFoster	Fleet	Plans	0.4	0.7	6.4	6.2	0.5	1.6	3.6	0.0	562.8	3.5	2.6
ScottStevens	Fleet		0.4	0.7	47.6	17.4	0.4	0.8	8.1	0.2	115.1	7.9	7.2
ScottStevens	Fleet	DutyRoster	0.4	4.7	8.0	16.5	0.4	1.7	2.8	0.0	597.9	8.5	6.4
ScottStevens	Fleet	Plans	0.5	11.0	7.2	5.4	0.4	31.9	50.1	0.0	475.3	0.3	0.3
SteveFinn	Fleet		0.4	11.0	10.4	12.4	0.3	0.8	2.7	0.0	491.3	8.2	9.5
SteveFinn	Fleet	DutyRoster	0.3	0.7	6.5	11.5	0.4	4.9	67.0	0.3	43.3	0.3	0.3
SteveFinn	Fleet	Plans	0.4	0.8	6.5	6.7	0.4	0.8	2.9	0.0	439.0	4.6	3.8
Averages			5.8	16.7	38.0	15.7	2.2	10.9	30.4	0.1	409.5	5.8	4.9

Figure 5. Interpreted vs. Compiled run times for one of the three policies from the reference implementation test cases.

Test files are from the reference implementation test suite. This models three security domains; a carrier fleet and a pair of strike forces within it. Each domain has access control policies, but only those for the fleet are shown in Figure 5. The columns show the time in milliseconds to process requests for three resources (Home Page, Duty Roster and Plans) by subjects with various roles in the three domains plus one ("bogus") that is not in any domain. Times for Sun's interpreter are highlighted in blue and the compiled implementation in green. The Req, Rsp, and Pol columns are time in milliseconds to parse each request (Req) and response (Rsp) and to instantiate a policy (Pol) for that request. The Run column shows the time to execute the policy's authzDecisionQuery method. The randomness results from the garbage collector intervening unpredictably. To minimize the randomness, each decision was recomputed 100 times and the average time reported in the Run column.

The interpreted and compiled runners both use JAX-P to load responses and JAX-B to load requests, so these columns should be the same. The Pol and Run columns are the times to instantiate a policy (Pol) for that request and then invoke it to render a decision (Run). The P+R column is the ratio of interpreted to compiled Pol+Run times. The average of this column, at the bottom, shows that the compiled implementation is 5.8 times faster. The Run column is the ratio of interpreted to compile Run times. It shows more radical differences, with the compiled implementation averaging 409.5 times faster.

These results are for an artificial offline environment with several differences from the online PDS environment. For example, the offline environment does not use XPATH and XQUERY stages. More fundamentally, the offline runners both load a policy for each request while the online versions reload policies only when they change and reuses them for many requests. The safest conclusion that can be drawn from these numbers is that compiled decision time is negligible compared to other overheads in the SOA processing chain. Compilation moves any lingering concerns over XACML performance away from the XACML core and out to the periphery, particularly to the transmission, parsing, and security costs of a large secure SOA system such as this.

Related Work

In view of how quickly we got this far, it is surprising that direct compilation of XACML to Java seems not to have been explored elsewhere.

Attempto Controlled English is often mentioned[456] in connection with XACML, OWL, and RuleML. However this does not seem to have led to working implementations. This is not surprising in view of Attempto's difficulties with nested expressions.

The MyABDAC project[7] uses compilation to enforce access control for MySQL database records. Rather than compiling XACML to executable code, it generates access control lists (ACLs) that MySQL interprets at run time. This might lead to a hybrid approach in which traditional PEPs guard access to applications while MyABDAC provides a SQL PEP that guards specific fields within applications.

The XEngine project[8] is similar to our approach. It differs by reducing the XACML expression tree to an intermediate numerical form to move string comparison overhead to compile time. It then converts the normalized policy to a tree data structure for run-time. That makes it an interpreter like Sun's that transforms the XACML expression tree into one adapted to the machine's needs at runtime. Our approach originated from the opposite goal; providing a format (Java source code) suited to the *user's* needs.

[5]

[1] Department of Defense Privilege Management Roadmap by The Office of the Assistant Secretary of Defense for Networks and Information Integration / DoD Chief Information Officer; 6 January 2010.

[2] IdAM Development and Sustainment Support's PBAC Proof of Concept (POC) Design

[3] Oasis extensible Access Control Markup Language (XACML) Version 2.0, OASIS Standard, 1 Feb 2005

[4] Analysis of Existing Policy Languages by Christopher Alm, HITeC / University of Hamburg, Michael Drouineaud, TZI / University of Bremen for the German Ministry of Education and Research (BMBF).

[5] XACML Policy Analysis Using Description Logics, VLADIMIR KOLOVSKI and JAMES HENDLER. Proceedings of the 15th International World Wide Web Conference (WWW 2007).

[6] NISTIR 7657; A report on the Privilege (Access) Management Workshop, NIST/NSA Privilege Management Conference Collaboration Team.

[7] Enhancing Database Access Control with XACML Policy by Sonia Jahid, Imranul Hoque, Hamed Okhravi, Carl A. Gunter University of Illinois at Urbana-Champaign

[8] XEngine: A Fast and Scalable XACML Policy Evaluation Engine; Alex X. Liu, Fei Chen, JeeHyun Hwang and Tao Xie; SIGMETRICS'08, June 2–6, 2008.