A Secure Network Node Approach to the Policy Decision Point in Distributed Access Control

Geoffrey Stowe
Advisers: Ed Feustel, Sean Smith, David Kotz
Dartmouth College Department of Computer Science

Tech Report TR2004-502

Winter, 2004

Abstract

To date, the vast majority of access control research and development has been on gathering, managing, and exchanging information about users. But an equally important component which has yet to be fully developed is the Policy Decision Point - the system that decides whether an access request should be granted given certain attributes of the requestor. This paper describes the research and implementation of a new PDP system for an undergraduate honors project. This PDP system employs three unique features which differentiate it from existing technology: collaboration capabilities, trusted management, and interoperability with other access control systems. Security considerations and future research areas are also discussed.

Contents

1 Introduction **1**

1.1 Background 1

2 Survey of Current Access Control Technology **4**

2.1 .htaccess 5

2.2 Shibboleth 6

2.3 PERMIS 8

2.4 CORBA 8

2.5 Home Grown System 9

2.6 SAML and XACML 10

2.7 Cardea 12

2.8 Our Entire Project 13

3 The Core Software **14**

3.1 Sun's XACML Software 15

3.2 Integrating Network Sockets 16

3.3 Simple Policy Management 17

3.4 Multiplexing PDPs 18

3.5 Original Protocol 19

3.6 Binary Requests 20

3.7 Adding Threads 21

3.8 Integrating SSL 22

4 PDP Collaboration **23**

4.1 PDP Collaboration in Our Software 25

4.2 Future Developments 28

5 SAML **28**

5.1 Translating SAML into XACML 30

6 PDP Management **33**

7 Security Considerations **35**

7.1 Core Software 36

7.2 PDP Collaboration 36

7.3 PDP Management 37

8 Conclusion **39**

9 Acknowledgements **40**

10 Glossary **42**

A Some Terminology **50**

A.1 XML 50

A.2 Public Key Cryptography and Certificates 51

A.3 SSL 53

A.4 Denial of Service Attacks 54

B Choosing a Development Environment **55**

C Other Attempts at Creating Object Streams **56**

C.1 ObjectOutputStream 57

C.2 Streaming XML 57

C.3 SOAP 59

D Implementing in a Secure Environment **60**

D.1 Secure Hardware 61

D.2 Using the IBM 4758 Secure Coprocessor 62

D.3 Other Secure Hardware/Java Card 63

D.4 Compiling the Java Code 64

List of Figures

1 A model of obtaining medical records from a doctor's office . . 3

2 Our access control model 13

3 A sample PDP collaboration configuration 26

4 A SAML request and the corresponding XACML request . . . 31

1 Introduction

This Honors Thesis was begun in the Fall of 2003 by Geoffrey Stowe under the guidance of Dr. Edward Feustel. While it is a computer science thesis, and hence inherently technical, I went to lengths to make this paper approachable and interesting to all audiences. People without a technical background may wish to read the appendix first for an explanation of some basic terms and concepts.

1.1 Background

The explosive growth of the internet over the last ten years has changed the way people live their lives in many different ways. Finding and sharing information from different parts of the world has gone from being a difficult problem to something everyone takes for granted. The entire paradigm of human communication has shifted to embrace the benefits of network technology. This shift has brought tremendous benefits, but with them come new dangers from people exploiting weaknesses in electronic systems. Words like 'hacker,' 'virus,' and 'encryption' have entered the cultural vocabulary. As government and industry continue to move their vital operations onto the internet, the need for secure, reliable systems becomes more imperative. This paper will discuss my research into a particular topic of network security that has promising potential to secure one particular part of internet communication.

Let's use the analogy of medical records to take us through an electronic

transaction. Say an insurance company enrolls a new client, and they want to retrieve the patient's records from his doctor across town. This is illustrated in Figure 1. The company sends a courier to the other office, and the courier asks the receptionist there for the records. The receptionist calls a doctor who must approve the release of the records. The receptionist will probably ask for a signed note from the patient and might even call the insurance company to make sure the courier is supposed to be picking up this information. The doctor will take all this information, decide the records should be released, and will tell an assistant to retrieve the records from their filing cabinet. The assistant then puts them in a sealed envelope and gives them to the courier who brings them back to the insurance company.
This analogy transfers naturally to an electronic setting. Every time you visit a webpage, a similar process will occur - your computer sends a request over the internet to a webserver running on a remote system, and the webserver sends back the webpage that you asked for. So for internet transactions, the insurance office in our example is some application on your computer. TCP/IP and other lower-level protocols play the role of the courier. The doctor's office is some remote server, the receptionist an application running on that server, and the filing cabinet a database. Sealing the record in an envelop is analogous to sending it over an encrypted channel, such as one provided by SSL. [1] A certificate exchange [2] could simulate calling the insur-

[1] See Appendix A.3 for a description of SSL.
[2] See Appendix A.2 for a description of certificates.

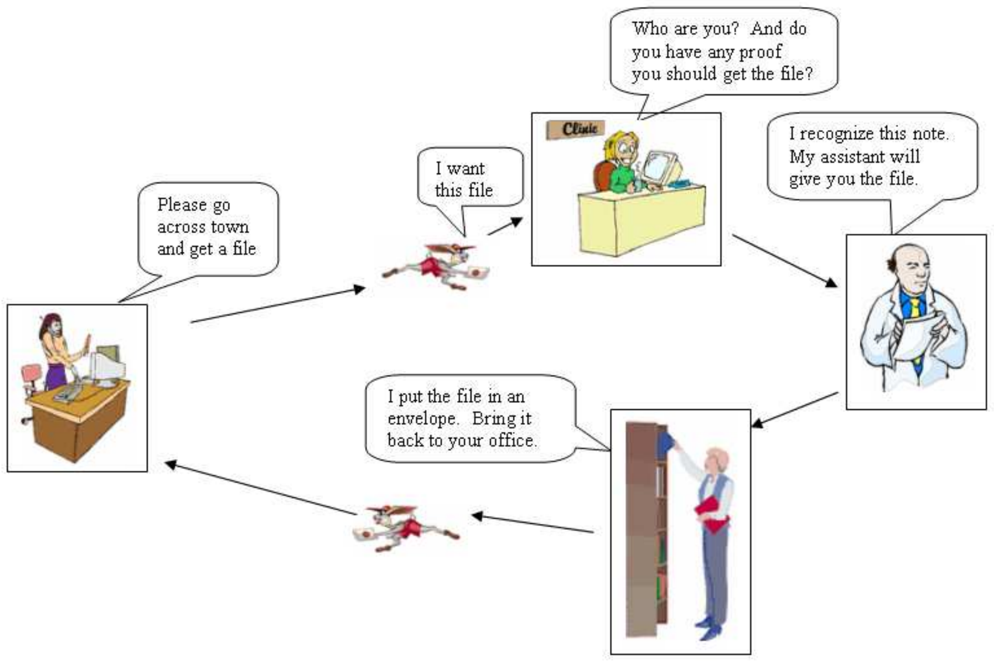
Please go across town and get a file
I want this file
Clinic
Who are you? And do you have any proof you should get the file?
I recognize this note. My assistant will give you the file.
I put the file in an envelope. Bring it back to your office.

Figure 1: A model of obtaining medical records from a doctor's office

ance company to make sure they authorized the courier. All of these steps have been, and continue to be, thoroughly studied.

There is, however, another key component to this process which is only beginning to be researched - the doctor. At the doctor's office, he is the one who decides whether to release the files or not. In an electronic setting this is called the Policy Decision Point (PDP), and it is essentially an application that decides whether to permit or deny a request. This is the topic of my research and software development. Using common internet standards and open source software, I developed a robust PDP system with features that, to our knowledge, do not exist in any other software.

In this paper I will first survey the current range of access control software. Then, I will describe the components of my software implementation focusing particularly on the original features. Finally, I will make suggestions for future research and extensions of this project. In my appendices I also continue discussion of various attempts and failures during the development process.

2 Survey of Current Access Control Technology

Access control and general network security are synonymous in many cases. A firewall, such as iptables [Wel04], is the primary means of controlling access to many online resources such as websites and email systems; firewalls are also used to protect personal computers from hackers and viruses. This is a program to moderate who is allowed to make any connection to a system. For example, a company's firewall might allow anyone in the world to access the company's website, but only allow certain people (or IP addresses, more specifically) to access the company e-mail server. Firewalls and other similar software can also be used to keep out viruses and protect against denial of service attacks.[3] However, as a means of access control they are limited to making yes or no decisions regarding whether an IP address can make any connection to a service. So, they can prevent unknown people from accessing any part of a website, but they cannot figure out whether a certain person is

[3]See Appendix A.4 for a description of denial of service attacks.

allowed to access a certain part of a website.

2.1 .htaccess

.htaccess is probably by far the most common access control scheme currently used on the internet. When a website asks for a username and password before displaying a page, it is most likely using a .htaccess file. This essentially password-protects a directory on a file system, so a web browser must present a valid username/password pair when requesting a document from the server. The content of a .htaccess file provides the simple rules for what usernames or IP addresses can access a directory.

The .htaccess scheme is simple, so it can be secure in most circumstances, but it is difficult to manage in a large system, and it does not hold up to serious attacks. Over regular channels, .htaccess is vulnerable to packet sniffing and replay attacks, although these problems can be eliminated if it is run over SSL. [Han01] Its decision-making capacity is limited to checking if a username and password is correct. This means that access to a resource can only be controlled by limiting who knows a valid password. In a large-scale system, managing usernames and passwords for every user on every resource is nearly impossible. Passwords might be written down or shared with other users, defeating the objective of access control. The username/password paradigm suffers other drawbacks such as the necessity to transmit passwords through some secure side channel, intruders cracking passwords, users forgetting passwords, and other weaknesses that can be mitigated with Public Key Infrastructure (PKI).[4] So, while .htaccess is a useful tool in many circumstances, it cannot provide sufficient power and flexibility to the next generation of distributed applications.

2.2 Shibboleth

Shibboleth, an Internet 2 project, is more sophisticated means of providing access control decisions. The Shibboleth model is relatively straightforward: a user contacts his or her institution and follows that institutions means of being authenticated. This part of the process could involve sharing personal information such as a person's name, their credit card number, or a certificate. When the user requests a service from a third party, that party can request and obtain credentials from the user's institution. If the credentials match the service's access control policy, it returns the requested service. [NS04] Privacy is very important in the Shibboleth handshake, so Shibboleth includes a mechanism to allow the user to choose which attributes it wants sent to the requested service. [Car01] In other words, a student trying to access a course web page could allow his name to be sent to the web server for verification, but the same student trying to access a political website could forbid his name from being released. Also, since the service requests credentials from the institution, many attributes will be generic instead of uniquely identifying individuals. Also, Shibboleth includes the ability to use an LDAP server to find more attributes about a person. This work is all

[4]See Appendix A.2 for a description of PKI.

done at the origin site, so the Policy Decision Point itself does not contact external attribute sources but rather uses only the information provided in a request. These two features of privacy and LDAP support would probably be useful in any robust access control system. Shibboleth is also embracing SAML (which will be introduced later) and uses it for much of its query and response traffic. [EC02] This could allow Shibboleth to communicate with other access control systems, including ours.

Shibboleth, however, is lacking in several regards. Shibboleth's version of the Policy Decision Point is called the Resource Manager, and it is very similar in concept to our system. However, Shibboleth leaves this part of the model as essentially a black box without defining standard policy language, combining algorithms, etc. Thus, it does not allow for collaboration between multiple decision points on the service side. Dartmouth researchers have developed a SDSI/SPKI system called SPADE that will allow users to control what attributes they release to their institution. This allows for decision point collaboration, but it applies only to the client authentication part of a Shibboleth exchange. [NS04] So, Shibboleth's RM model allows for only one institution to control a given database. Also, it does not define any uniform policy language, so it cannot utilize the flexibility found in XACML. Our system includes both these features. So, our access control system utilizing XACML could conceivable be used with Shibboleth through the common language of SAML and also improve upon Shibboleth by including additional features.

2.3 PERMIS

The PrivilEge and Role Management Infrastructure Standards validation (Permis) is a European project designed to determine user authorization when accessing a computer system. One of their large-scale implementations is in Bologna, Italy and it allows architects and other construction professionals to view, modify, and comment on building plans online. Permis takes the approach of dividing certificates into two categories: permanent identity (PKI certificates) and changeable attributes (PMI certificates). PKI certificates can be issued by standard CAs just as they are under other schemes, but PMI certificates are only issued by the responsible organization. In other words, if an individual wants to certify that he is an architect for Company A, he will get a certificate issued by Company A.

The Permis infrastructure is similar to that of Shibboleth in that it allows for individual sites to issue certificates attesting to attributes they can verify. Also, like Shibboleth, it leaves definitions of specific Policy Decision Point operation to whoever implements the system. Currently, Permis is being integrated into the Internet 2 project NMI, so I would expect it to begin using some of the SAML standards in later versions. [BL03]

2.4 CORBA

CORBA is a middleware infrastructure that allows processes running on distributed hosts to securely communicate over the internet. [Gro02] While this is not directly applicable to access control, communicating between processes involves many of the same problems. For example, when a CORBA object on one computer receives a stub request, it must figure out whether or not to execute the request. CORBA is used as guidance for some of the protocol issues which arise in our system.

2.5 Home Grown System

In the summer and fall of 2003, I worked on a Homeland Security Council project called Livewire which involved government and private sector employees playing a simulated cyber attack on the nation's infrastructure. One of my responsibilities was to restrict webpages on our system so only appropriate players could see them. I mention this problem because I believe it will become very common in other settings. We were using client certificates to authenticate users and keep unauthorized individuals out of the system, so we felt using passwords and .htaccess files was cumbersome and unnecessary. Since we created the client certificates ourselves, we included both unique identifiers and group identifiers that we planned to use to control access. We wanted a straightforward way of pulling a single field off the client certificate and performing a simple check to either permit or prohibit access. We also wanted to prevent users from accessing the system outside of the appropriate timeframe, and we wanted to be able to make changes to our policy quickly without disrupting the rest of the system. Finally, we did not want to use a system like Shibboleth that would first require learning the nuances of their system, then setting up a Shibboleth site in front of our servers and, ideally, a Shibboleth site at each of our clients.

We could find no system which fit all of our needs. Ultimately, we modified the Apache configuration files to hardcode which certificates were allowed into which directories. While this solution worked, it was less than optimal for two reasons: security and ease of use. Since we relied on our web server to make access control decisions, any hack of our webserver would compromise the entire system. This was a particularly dangerous vulnerability since webservers are complicated systems - thus they are prone to exploits and cannot easily be run on secure hardware. Also, modifying Apache configuration files is a tedious process that is prone to error. Making changes on the fly is impossible since the webserver must be restarted before changes take effect. What we ideally wanted was a way to write an access control policy, plug various components into Apache, and have everything work. This was a large part of my motivation for building this system.

2.6 SAML and XACML

The Organization for the Advancement of Structured Information Standards (OASIS) is a group representing both academia and industry that defines standard ways of encoding different types of information. The two OASIS standards we are concerned with are the Security Assertion Markup Language (SAML) and the eXtensible Access Control Markup Language (XACML). These are both XML[5] specifications, meaning they define what

[5]See Appendix A.1 for a description of XML.

information can and must be included in communications at various stages of an access control dialogue. SAML defines mechanisms to relay access requests, results, and any required intermediary information. This is used by many organizations as a method of communicating authorization information. However, SAML does not specifically define how the Policy Decision Point should operate. In other words, both a SAML request and a SAML response are well defined, but getting from one to the other is left up to the user to implement in any way.

XACML version 1.0 was published in 2003, 2 years after the original SAML specification. Its major feature is that it defines a common way to represent access control policies. This is the mechanism that takes a request and decides what response to return - the heart of a PDP. However, XACML also defines its own request and response formats, so it is not directly compatible with SAML. Sun Microsystems has written an implementation of XACML in Java which includes the XML parsing, basic decision making algorithms, policy combining algorithms, and the capability to add custom attributes and algorithms. But, the Sun implementation does not include the capability to run on a network and read requests over a secure socket. This is a key step in enabling XACML to be used in real-life settings.

Using the Sun XACML implementation as a core for my software, I extended the software to develop a new conception of the PDP. I designed the PDP to function as an isolated node enabling three new features: reliable policy management, PDP collaboration, and compatibility with other access control systems. Policy management works by sending XACML requests to the PDP server in order to add policies or start instances of a PDP. Since they take the form of an XACML request, they can be permitted or denied the same as any other request. PDP collaboration can allow multiple computers, and even multiple organizations, to all simultaneously control access to one resource. Since the PDP server is one distinct node, it can forward requests to other PDP servers and require approval from them all before granting access. Finally, I implemented a simple translation from SAML to XACML that will allow this PDP system to communicate with other access control programs using the common language of SAML.

2.7 Cardea

Cardea is one of the test sites currently experimenting with both SAML and XACML. A project of the NASA Advanced Supercomputing Division, Cardea employs a very similar model of access control to ours, which I will describe in the next section. It's unclear how much of their model they have implemented, but they propose having both a SAML PDP and an XACML PDP. Also, their PDPs cannot communicate with each other to both independently verify a decision, nor with other PDPs outside their system. [Lep03] Our model proposes a PDP that can handle both SAML and XACML as well as communicate with PDPs in other systems. So while Cardea approaches the same questions as we do in much the same way, the PDP described in this paper allows for more extensive operability than is found in their system.

2.8 Our Entire Project

This research was carried out in collaboration with Paul Mazzuca and Fang Pei, both of Dartmouth College. Our goal is to produce a complete access control system implementation following the diagram below. The general

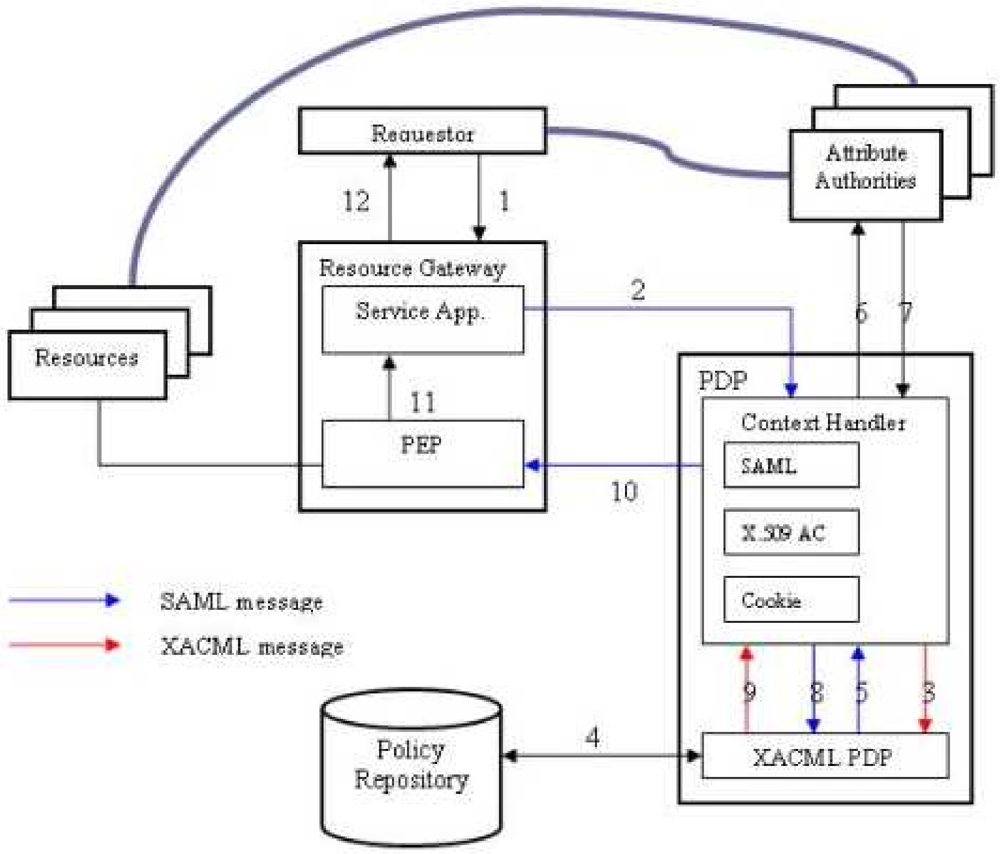

Figure 2: Our access control model

paradigm we are using is that some client makes a request to a service. A common example is a web browser requesting a webpage from a webserver. The service then collects whatever relevant attributes it knows about the client. Typical attributes could be the client's IP address, a client certificate, or a name the client has sent to the webserver. The service will bundle all this information into either an XACML request or a SAML request and then send it to the PDP. The PDP will make a decision based on the request, its internal policies, environment information such as the date and time, and any other external attributes it wishes to gather. Once it has made a decision, it will create an XACML or SAML response which it sends to the PEP. If the PEP believes the response is valid, it will release the requested resource to the service. In the webpage example, this would entail reading the contents of a file from a disk and transmitting the contents to the service. The service will then return the requested information to the client.

3 The Core Software

My software consists of several different components of different levels of complexity. Since I designed them sequentially, my design of earlier components fed naturally into the design of later components. So, I will describe the software in roughly the chronological order of implementation. Some of the later features were designed to replace the earlier ones, for example SSL sockets should be used instead of regular sockets, but the final software includes all the components I will describe below. The original features I added are discussed in subsequent sections.

3.1 Sun's XACML Software

As a member of OASIS and a leader in their field, Sun Microsystems is at the cutting edge of distributed network systems. After OASIS published its XACML standard, Seth Proctor, a Sun employee, led an open-source development team to implement the XACML standard in Java. The result was a powerful piece of software which implemented all the OASIS standards. Specifically, it provided for the creation of Policy Decision Points which could process request objects and return responses. Many of the standard policy combining algorithms were included. However, it would only function if all the information it required was on a single computer. As the documentation says: "This distribution does not provide any kind of support for sending requests and responses over a network (e.g., to support an online PDP service)". [Mic03b]
Sun's XACML implementation is freely available at their website, sunxacml.sourceforge.net, and it includes some helpful guides for installing and using it. My first tests with the software involved reading in sample XACML policies from local files, evaluating them, and returning a result. For example, the sample program that comes with the implementation is as follows:

```
// load the policies
FilePolicyModule policyModule = new FilePolicyModule();
policyModule.addPolicy("Policy.xml");

// setup the policy finder
PolicyFinder policyFinder = new PolicyFinder();
Set policyModules = new HashSet();
policyModules.add(policyModule);
policyFinder.setModules(policyModules);

// module to provide the current date & time
CurrentEnvModule envModule = new CurrentEnvModule();
```

```
// setup the attribute finder
AttributeFinder attrFinder = new AttributeFinder();
List attrModules = new ArrayList();
attrModules.add(envModule);
attrFinder.setModules(attrModules);

// create the PDP
PDP pdp = new PDP(new PDPConfig(attrFinder, policyFinder, null));

// now work on the request
RequestCtx request =
RequestCtx.getInstance(new FileInputStream("request.xml"));
ResponseCtx response = pdp.evaluate(request);

// for this example we'll just print out the result
response.encode(System.out);
```

As the comments indicate, creating a PDP involves several steps: loading in an XML file that describes the policy, setting up a policy finder, setting up an attribute finder, and finally starting the PDP. Then, a request can be read in from an XML file and passed along to that PDP. Finally, the response will be printed out.

3.2 Integrating Network Sockets

The simplest feature of my PDP system is the ability to read in requests over a network socket. This required creating a new class that would do the following:

- Create a PDP
- Read requests from a socket
- Process those requests
- Return the result over a socket
- Handle errors so a failure will not crash the entire system

Since creating a PDP involves specifying which policies it should use, this software needed to include a mechanism for reading this information over a socket. Thus, since the code had to respond to different requests in different ways, designing my software included designing a protocol and a mechanism to manage policies.

3.3 Simple Policy Management

I created a new class called PolicySet which would store a list of policies someone wanted to include in a PDP. This class basically consists of a HashSet of policies and some helper functions. For a user to add a policy, he or she would first store the policy on the local file system through passing the STORE command followed by the XML text of the policy. Then, the user would ADD that policy to a PDP. Although it appears to the user as if he or she is adding policies directly into a PDP, in reality the policies are simply being collected to be added all at once when the PDP is started. The user can also LIST which policies are stored on the system, or in a particular PDP.

3.4 Multiplexing PDPs

Let's return to our model for access control and envision a theoretical company implementing it. Since the PDP can be separated entirely from the rest of the system, it could ideally run on a secure system which will provide the utmost assurance that the decisions it makes can be trusted. However, since secure hardware is expensive, it would be reasonable for this company to want all of its access control decisions made on one system. Therefore, I endeavored to allow multiple PDPs using shared policies to run on the same platform. Since they would all be listening on the same port, this meant building a layer of multiplexing that would direct requests to the correct PDP.

On a low level, multiplexing PDPs meant building a Java Set of references to different PDP objects. So, I created a class called PDPSet which would store PDP objects along with a name and allow access to those PDPs in order to evaluate requests. This again involved creating a class that stored references using a HashSet and managed modifications to that HashSet with various helper functions. One issue which arose was how to handle a user who requests creating a PDP which has the same name as an existing PDP. In my implementation I ignore such requests, but theoretically one could design a policy for how to handle this issue. Later I will discuss my approach to the generalized question of how to authorize modifications to the PDP server.

The source code for these two classes can be found in the files PolicySet.java and PDPSet.java.

3.5 Original Protocol

My original text-based protocol was modeled after the File Transfer Protocol [PR85] and the Simple Mail Transfer Protocol [Pos82]. Basically, the client would connect to the server's port and issue commands followed by parameters. The options are as follows:

- **STORE** *policy* - This command should be followed by a newline, then the text of an XACML policy. The policy should be terminated by a '.' on a new line. It will save the policy on the local file system and allow the user to LOAD that policy into a PDP.
- **LOAD** *pdp policy* - This will add the specified policy to the specified PDP. Once the PDP is started, it will be created with all of the policies that have been loaded.
- **LIST** *pdp* - Lists the policies that have been loaded into the specified PDP.
- **LIST ALL** - Lists all the policies that have been stored on the system.
- **START** *pdp* - Will start running the specified PDP.
- **EVAL** *pdp* - This command should be followed by a newline, then the text of an XACML request. The request should be terminated by a '.' on a new line.

- **DEFAULT** *pdp* - This command will set the default PDP to be used in evaluating binary requests (discussed in the next section.)
- **HELP** - Will print out a summary of the commands and their functions.

The code implementing these features can be found in the Appendix in the file TextListener.java.

3.6 Binary Requests

The text-based protocol worked well in testing, but in an actual implementation the PDP would communicate mainly through an automated protocol with a PEP. Therefore, we realized it was necessary to make the process of sending a request to the PDP as simple as possible. Note that in the sample code shown in a previous section, requests are parsed into RequestCtx objects which are then passed to the PDP object. We wanted to create the simplest way to transmit a RequestCtx object from one system to another.
A RequestCtx object, and a ResponseCtx object will return their XML encoding by using the encode() method. This method takes an OutputStream as a parameter, so this allows us to stream XML over a socket. At the other end of the transaction, we can call the RequestCtx.getInstance() method and pass in the InputStream which comes out of the other end of the socket. Assuming the XML was successfully transmitted over the socket, this method will send both RequestCtx and ResponseCtx objects over a network.
After trying several approaches to the problem of reliably transmitting XML

over a socket[6], we settled on an implementation that would read all necessary input into a byte array, then convert that to a stream to send to the XML parser. This allowed us to keep the streams (running over a socket) open for multiple requests. The one drawback to this approach is that it uses a class called StringBufferInputStream [Mic03a] which is deprecated as of the JDK 1.1. If future implementations also buffer the data coming across the socket, I would recommend using a method that does not involve the StringBuffer-InputStream class. However, this method has worked fine for us in all of our tests.

3.7 Adding Threads

Once I updated the PDPNode software to listen on multiple ports for different kinds of requests, I needed to add threads so it could wait for a connection on two ports at once. Because the Java Socket.accept() method is bound to one socket, the method will pause the entire program until a connection is made if threads are not used. I used the built-in support for threads included with Java; the one trade-off in this addition is a slightly degraded performance speed.

A further complication is that these multiple threads need to share the same resource. That is, both the binary listener and the text listener need to use the HashTable of PDPs because they will be evaluating decisions using the same PDPs. So, we need to synchronize the threads. This requires creating

[6]See the Appendix for a more detailed description of our other attempts at sending RequestCtx objects across a socket

the shared data objects in the parent class and then passing in the same objects as parameters to both threaded listener classes. [Mic03c]

The final version of my software has four threads running simultaneously: one listening for text-based requests, one listening for object-based requests, one listening for object-based requests on an SSL socket, and one listening for management requests on an SSL socket.

3.8 Integrating SSL

As I describe in the Appendix, SSL is a mechanism for encrypting communications over a network. This is vital to the security of our PDP for several reasons: it prevents 3rd parties from tampering with data passing into or out of the PDP, it prevents 3rd parties from seeing private information that may be included with a request, and it authenticates both the client and server in an interaction. To take advantage of this last feature, we require both client and server authentication for all our connections.

Java includes support for SSL sockets, so I based the SSL support in my software off of previous examples. [Par02] To communicate with another computer over SSL, both systems need to obtain a certificate, set up a key store and a trust store. For each certificate a computer generates, it must add the corresponding private key to the key store. The key stores we set up are protected by a password, but they are not particularly secure. Ideally, the key store will be protected by being located on special hardware. The trust store is a collection of certificates from other computers you intend to com-

municate with. Ideally they will be signed by a CA that you trust, hence the term trust store. In our experiments we used self-signed certificates which are not optimal from a security standpoint but are easier to manage.
The final version of my SSL software is located in the file SSLListener.java.

4 PDP Collaboration

No existing access control system that we have encountered can employ multiple Policy Decision Points. In many ways, the notion of multiple PDPs is a contradiction - how can one decision be made in multiple places? But implementing collaboration between PDPs allows for interesting scenarios in the realms of both technology and society.
Collaboration in a capitalist economy is a double-edged sword because it can both increase productivity and limit competition. Often times competitors use subtle back-channel means to communicate. [DS96] Sometimes competitors even attempt to sabotage each other by withholding important information. [Ker99] As network technology becomes more and more pervasive in everyday life, communication becomes easier to disguise and collusion becomes more difficult to adequately control. I believe a reliable, open-source PDP system, such as the one I have developed, can enable trusted cooperation between organizations and alleviate fear of illegal collusion. If a PDP policy is understandable, and the PDP software is resistant to tampering, then people can commit to a cooperative effort confident that they know the exact extent of their involvement.

Consider the following scenario: Company A, Company B, and Company C are competitors in the same industry, and none of them trust each other. They all collect copious data on their customers, and they all want to perform market research. However, they all want to include each other's data in their analyses in order to utilize the largest sample possible. Clearly they are not willing to provide each other with detailed records of their clients, but they are willing to share anonymous or aggregated data. Furthermore they want to continue sharing data in the future instead of setting up a one-time aggregated database. The government learns of this, and wants to ensure that this collaboration is limited to market research and is not a means of shrewdly colluding to fix prices.

This scenario is realistic, but with existing technology it cannot be accomplished. Currently there is no generic system which can allow multiple entities to collectively make access control decisions. Our model specifically isolates the Policy Decision Point as a separate system. This allows the PDP to perform any actions before returning a decision, including contacting other PDPs.

Since PDP collaboration is a new idea, our software approaches it in a fairly simple, yet highly extendable, fashion. I will explain how our implementation works, what issues arose, and how we chose to address them.

4.1 PDP Collaboration in Our Software

When setting up a PDP, the user can specify other ”external PDPs” that must agree on a decision before the PDP returns a result. These are specified by providing an IP address, a port on which the PDP is listening for object requests, and the name of the PDP. When a request comes in, the PDP firsts evaluates it using its own policy. Then, it contacts each external PDP, sends the request, and gets a response. Lastly, it combines all the results together into a final decision that it sends to the user. This is loosely modeled after the protocol for DNS queries - if a local DNS server cannot resolve a request, it will forward the request to another server. [Moc87] An alternative model would have been to require the PEP to obtain permission from multiple PDPs. However, we felt this would have changed the role of the PEP by requiring it to store something resembling a policy.

Manually specifying which external PDPs to collaborate with when the PDP is created is inconvenient, hard to replicate, and hard to keep records of. This can be solved by including the list of external PDPs in the XACML policy itself. That way the decision making process is documented in XML and can be replicated by loading the same policy into a different PDP. This would also allow for some degree of logic tests to be performed in choosing which external PDPs to contact. For example, maybe requests during the day require the additional consent of another PDP, but requests at night require the additional consent of a different PDP. It is hard to imagine any such scenario, but the possibility should be provided. However, for now we chose not to encode external PDPs in policies because we did not want to make any modifications to XACML as it is currently specified.

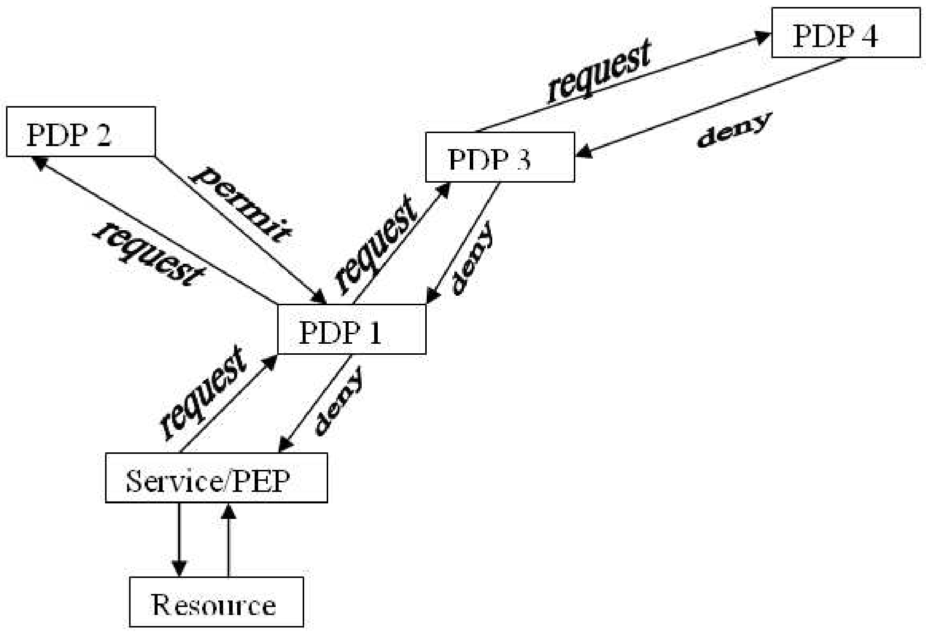

Figure 3: A sample PDP collaboration configuration

Figure 3 illustrates an interaction that might occur when a request comes in to PDP 1. It contacts PDP 2 and receives a permit decision. It then contacts PDP 3 which contacts PDP 4 and receives a deny decision. This is relayed to PDP 1 which issues a deny decision. This configuration could be the result of Companies 1, 2, and 3 jointly protecting a resource. Also, Company 3 and Company 4 might have a separate agreement to monitor and authorize each other's actions. PDP 4 could also be a regulatory agency charged with assuring Company 3 is not participating in any questionable practices.

This current scheme does not consider what happens if one of the required PDPs is unreachable. This could legitimately result in either ignoring that PDP or automatically rejecting the request. Say, for example, that one of the external PDPs is connected through an unreliable ISP in Asia and wants to be contacted solely for logging purposes. It may be reasonable to ignore this PDP if it cannot be contacted. However, in a different scenario the desired behavior may be to reject any request when all PDPs cannot be reached. Say Company A and Company B are controlling a database, and clever hackers infiltrate Company A. This allows them to create all the faulty credentials they need to successfully trick Company A's PDP into granting access.[7] Then, they launch a Denial of Service attack against Company B's PDP causing it to be unreachable from Company A. In this scenario, Company A's PDP will hopefully return a decision that denies access. The point of these examples is to illustrate that the question of what to do when a PDP is unreachable should also be specified in a policy. This will allow users to control PDP behavior in an easily customizable, replicable way. Again, we did not pursue any development in this direction because the current version of XACML does not include any such capability.

The final aspect of this scheme was to combine the responses from external PDPs. For our purposes, we simply checked to see if any denied access,

[7]In this scenario, the PDP itself is not infiltrated - just other parts of the network. In the Security Considerations section, I will discuss the possibility of the PDP itself being compromised.

otherwise the PDP will permit the request. This will probably be the most common combination algorithm desired because it requires all parties to grant access. However, it is conceivable that some scheme could require just one PDP to permit access. Or, perhaps two specific PDPs would have to permit, then one of any three others would have to permit. Since we recommend that the list of external PDPs be written in the XACML policy, it is only natural for that policy to include the rules for combining the results.

4.2 Future Developments

Communication between PDPs is a new paradigm, and hence opens many other possibilities for future development. For instance, load balancing or fault protection can be incorporated into future distributed PDP implementations. As long as all nodes speak the common language of SAML, their collaboration can produce interesting results. Using SAML also provides a means of keeping authoritative records of what decisions were made and who approved them. Since SAML objects can be signed, a permit decision that includes a digital signature would provide incontrovertible evidence that a PDP authorized an action. This could be tremendously beneficial to businesses that require strict accountability.

5 SAML

The leading open-source implementation of SAML is written by Scott Cantor at Ohio State University and called OpenSAML. There is practically no

sample OpenSAML code available anywhere on the internet. As the OpenSAML readme file says, "[t]here are no explicit samples yet, but there are test programs and higher level code in the Shibboleth codebase that would help a novice see what some of the classes do." [Can02]
The following packages all needed to be downloaded, installed, and added to the project build path in order for OpenSAML to function, so they are all included in the final version of my software:

- xmlsec - This package provides various XML security services including the ability to sign XML documents. [GPLMvdK03]
- log4j - This is a an efficient, general-purpose logging utility. It uses the concept of inheritance to provide its users with a high degree of control over logging activity. [Bea04]
- Xerces - This is an XML parser developed by The Apache Software Foundation. It implements the XML Document Object Model which is required for using Open SAML. [Fou04c]
- Xalan - This is an XSLT processor also developed by the Apache Software Foundation. [ea03]

I also upgraded to the Beta version of the Java 1.5.0 Runtime Environment. [Mic04a] With all of these packages, I was finally able to create simple SAML objects.

5.1 Translating SAML into XACML

Currently the functions of SAML and XACML overlap in the realm of making access requests. A system employing OpenSAML would send a SAMLAuthorizationDecisionQuery to the Policy Decision Point and receive a SAMLAuthorizationDecisionStatement in response. To use XACML for our policies, on the other hand, we must send a RequestCtx into the PDP and receive a ResponseCtx. So while SAML and XACML perform a similar function in encoding requests and responses, they are not strictly compatible. To reconcile this problem, our model puts a system in front of the PDP that translates requests in other formats into XACML requests. So, a SAML request will enter our system where it is translated into an XACML request and then sent to the PDP. The challenge at this stage becomes to translate from a SAML request to an XACML request. According to Scott Cantor, no one has yet come up with a way to convert between these two. [Can03] I will first outline the parts of a SAML request in the OpenSAML implementation, and then explain the choices I made in implementing the translation.

A SAMLAuthorizationDecisionQuery has four components: a SAMLSubject, a resource, a Collection of actions, and a Collection of evidence. SAML Subjects have two parts, and either or both can be present in a SAML subject. The first is the NameIdentifier. This consists of a name, a name qualifier which is used when usernames from multiple locations can overlap, and a name format. The second is SubjectConfirmation which allows the identity of the user to be confirmed. This consists of a ConfirmationMethod which is

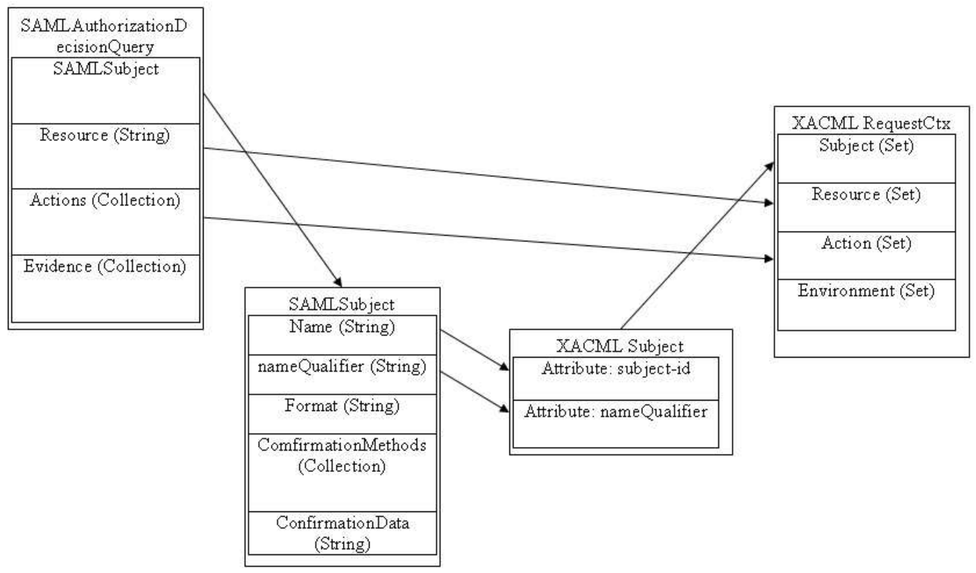

Figure 4: A SAML request and the corresponding XACML request

a URI reference which can be used to validate the subject, additional SubjectConfirmationData, and KeyInfo to allow the subject's cryptographic key to be checked. [MMea03]

A Subject in XACML is essentially a set of Attributes, with the option of specifying a subject category. If none is specified, then the default is used. Each Attribute has 4 parts: the URI to identify what kind of attribute it is, an issuer to declare who is asserting this attribute belongs to this subject, a timestamp, and finally the attribute value. This is closely analogous to, for example, a government issued ID card. Its size and layout distinguish it as an ID card (analogous to the URI), the issuer is indicated by the template, a seal, and possibly a hologram, the issue date is the same as a timestamp, and finally the person's name, birth date, and address printed on the card is the value of the attribute. A subject can have more than one attribute. Following the ID card example, a person would be a subject, and the drivers license, college ID, and credit cards in their wallet would be their multiple attributes. To us a computer example, a subject could be a UNIX user-id, and attributes could be what their password is, what groups they belong to, and what processes they are running.

SAML is built upon the assumption that each request will have only one subject - this is reasonable in most cases, particularly those of single-sign-on scenarios which is one of the major goals of SAML. However, XACML extends the abstraction of access control to allow for multiple subjects. Therefore, in translating between SAML and XACML we will always build requests with one subject.

So, returning to the challenge of converting from SAML to XACML. An XACML request includes four components: a Set of subjects, a Set of resources, a Set of actions, and a Set of environments. The action and the resource can be translated almost directly from the SAML request, and SAML does not contain any environment information, so we can ignore that. The difficult part is converting a SAML subject into an XACML subject.

XACML as it is currently defined does not include any means of verifying information in a subject statement, so we'll ignore the ConfirmationMethod and ConfirmationData sections of the SAML subject. Therefore, we are left with a name, a name qualifier, and a format. The format isn't applicable to

XACML because the subjects we will handle will always have a subject-id, and additional attributes are evaluated separately. We will construct two attributes - for one the value will be the name, for the other the value will be the name qualifier. When we bundle these two Attributes in a Set, they are ready to be included in an XACML request.

Our final step is to then construct the XACML request so it can be passed off to the PDP. In implementing this I ran into Java exceptions when I tried printing out the initial SAMLAuthorizationDecisionQuery. It appears other people have encountered this same error [Dor03] so I avoid printing this out in my code. Aside from that, this conversion appears to work for all the SAML requests I tested it on. The algorithm may require some modifications for more complicated transactions since I don't utilize all the elements of SAML and also only use SAMLAuthorizationDecisionQuery objects instead of utilizing all of the components of SAML communication. However, this has worked for our purposes and provided the first PDP that can handle both SAML and XACML requests.

6 PDP Management

The next major feature of this PDP system is an improved mechanism for managing policies and individual PDPs. Let's review the essential actions we can perform on a PDP. We can store a policy, load a policy, set a default policy, list policies, start a PDP, and evaluate a request. The management features we'll consider are storing, loading, setting a default, and starting.

We need to ensure that only people who are properly authorized can perform these actions. But we, of course, already have a system to determine if someone is authorized to perform an action - XACML.

To use XACML as the authorization tool for PDP managment, I needed to do two things - formulate a way for management requests to be written in XACML, and create management policies to evaluate these requests. An XACML request contains a subject, an action, and a resource. The subject will be used to identify who is making the request. For management requests, the action will be one of the actions listed above - store, load, default, start.[8] Since all these actions (except store) are performed on individual PDPs, the resource will be the name of a PDP. For example, if I want to load the policy *myPolicy2* into the PDP *myPDP* we would send a request where the action is *load myPolicy2* and the resource on which we're performing this action is *myPDP*. One complication arises in the action of storing a policy. Remember this requires passing in the XML text of the policy. So, in this case the text of the policy will be the second action passed in.

If we are modifying policies based on the decision of policies, we of course need some "root" policy to start from. When this software first starts, it reads a configuration file that specifies where the root policy is. Ideally this will exist on the local filesystem, but theoretically the software could be extended to put the root policy anywhere. Using this approach, the PDP node becomes

[8] I intentionally chose not to include EVAL in this feature for reasons I will explain in the Security Considerations section.

even more independent. Once started, an administrator should be able to make any modifications by communicating with the PDP. In other words, making changes does not require rewriting a configuration file, and the PDP should never have to be restarted.

The idea of using a system to manage itself is by no means new to computer science. In fact, the basic tenant of a computer that differentiates it from other electronics in that it can reprogram itself. But as far as we know, this is the first time this approach has been applied to an XML-based access control system.

This code can be found in the file ManagementListener.java. It currently runs over an SSL socket so the text of policies cannot be intercepted by an adversary.

7 Security Considerations

Since the goal of this PDP Node software is to improve the security of other systems, it is imperative that it be as invulnerable to exploitation as possible. I will describe the security decisions I made, and the features I included in this software. I will also discuss possible improvements that can be made in the future. Note that the best possibility of completely securing the PDP system would be to implement it on secure hardware. I discuss my research into this possibility in the Appendix.

7.1 Core Software

The major security feature of the core software is its SSL capability. This will prevent anyone from intercepting requests or responses and also prevent replay attacks. Also, since we require client authentication, the possibility of an intruder masquerading as the PDP, or fooling the PDP into communicating with a bogus service or PEP, is almost nil.

The question of how the PEP, PDP, and service first find each other, however, has not yet been addressed in this paper. Presumably, in most installations now conceivable, the same administrators will set up each component. In this case, the IP addresses or hostnames of each can be hardcoded, and the relevant certificates can be loaded into the appropriate trust stores. An IP address alone is not very secure, but using certificates as an additional means of authentication should make connections reliable. However, as more and more people begin to take access control seriously, the desire for, and abundance of PDP servers will expand. In this case, a service and PEP may have different choices for which PDP to utilize. Here some secure, trusted variation of a DHCP [RDBV+03] server could be implemented to mitigate associations with different PDPs.

7.2 PDP Collaboration

Since PDP collaboration uses the same interfaces as the regular request-and-response protocol, it presents few additional security risks. As I discussed earlier, policies for handling unreachable PDPs and combining PDP results should be specified by an administrator, so managing the reliability of other systems and the connections to those systems is left in the hands of the those setting up the PDP.

The major vulnerability PDP collaboration presents could come in a situation where two PDPs are controlling the same resource. If one of the PDPs is hacked, then an attacker could use that PDP to manufacture "permit" decisions without consulting the other PDP. This decisions would appear valid to the PEP, so an attacker could gain access to the protected resource. There are two approaches to preventing this possibility: securing the PDP or forcing the PEP to check with multiple PDPs. We recommend the former solution because it best fits with out model. But, if PDP hacks become a problem, the latter solution may have to be examined.

7.3 PDP Management

As I mentioned earlier, the PDP will evaluate all management requests against a root policy - including modifications to the root policy itself. So, the security of the root policy is essential to assuring that the other policies are legitimate. Here I faced a trade-off between security and convenience: locking the root policy to prevent it from ever being modified would lessen the chances of a hacker compromising the entire system, but would force administrators to stop and restart the PDP if they ever wanted to change the root policy. I chose the convenience solution because it allowed for greater flexibility and ease of use. Also, if this PDP is implemented on secure hardware then the question of hacks becomes almost a moot point.

Up until now I have not addressed the issue of denial of service attacks. This is the malicious flooding of a system with seemingly legitimate requests designed to crash a server or prevent valid requests from being processed. Since the PDP Management feature is most vulnerable to a denial of service attack, I will discuss protective measures in this section.

By nature, it is difficult to differentiate between legitimate traffic and malicious traffic because there is nothing inherently malicious in the packets of a denial of service attack - just the sheer volume of those packets. The best defense against these attacks is at the firewall level where packets from certain IP addresses or packets destined for certain ports can be dropped. This is an effective defense because very little time is required to process the header of an TCP/IP packet and perform simple checks.

But, what happens if an adversary gets past our firewall and bombards our PDP with XACML requests? Remember that we do not check evaluation requests against our root policy to see who is allowed to make requests. So, our system will dutifully process and reply to these requests. While this may not be the ideal implementation, the alternative is no better. If we must authorize the requestor before we process the request, then we've already dedicated a substantial amount of processing power and time. Remember that processing an XACML request requires parsing XML, searching for an applicable policy, and matching that policy to the parsed request. All of these operations are costly in terms of memory and latency. So even if we refuse to evaluate the individual requests made by a request in a denial of service attack, the attack will still achieve its purpose of clogging our system. Therefore, we leave security against denial of service attacks to alternate means. In reality, most PDP systems could sit comfortably behind a strict firewall because they only need to communicate with a few external systems as shown in our model. So accepting only connections from those system's IP addresses, and only listening on the limited number of ports that this software uses, will protect against all but the cleverest denial of service attacks.

8 Conclusion

The concept of the Policy Decision Point has been evolving over the last decade. In most software, it is intertwined with other components or ignored altogether. But the PDP is starting to be defined as a separate system in packages like Shibboleth. XACML takes this to the next level by providing a language for writing, combining, and evaluating access control policies. This software is an implementation of a PDP system built upon the Sun XACML implementation and other open source projects. Also, I approach the PDP as not just a separate system, but as one node in a network. This allows for PDP collaboration and interaction with PDPs in other access control systems. Also, it enables all management to take place remotely so the PDPs' operations can be thoroughly trusted.

This paper also suggests avenues for future research. The management scheme I created works in many circumstances, but it should be standardized. With XACML, any environment information can be considered in making decisions, so factors like disk size, remaining quota space, or processor load can be considered in writing management policies. I recommended that collaboration information also be stored as part of a policy, but the exact parameters of this mechanism need to be developed. Also, the PDP should be capable of keeping signed records to prove what decisions were made by what PDPs. To keep the PDP decisions reliable, the system should be implemented on secure hardware. This could be accomplished by running the Java Virtual Machine on a secure platform or by compiling the PDP code to run on the IBM 4758. Finally, my biggest recommendation is for people to use this technology. Only by exposing these systems to thousands and millions of users can we learn what challenges await.

9 Acknowledgements

I was fortunate to have the help and support of several people during this project. First and foremost, my adviser Ed Feustel provided help and guidance from the beginning of the project - without him none of this would have happened. Paul Mazzuca and I spent many hours together developing the communications protocol and helping each other debug code. Fang Pei developed the model I included in this paper which provided the basis for our implementation. Seth Proctor was a definitive source for us in understanding XACML, and he provided software guidance throughout the project. I'd like to thank John Marchesini for his help in understanding the IBM 4758. Finally, thanks to Katie Lieberg for helping me make my thesis understandable to people other than just myself.

10 Glossary

- **Certificate** - A public key and some other information signed with a private key. These are used to certify that a public key belongs to some entity.
- **DDOS** - A Distributed Denial of Service attack is one that floods a computer with bogus traffic from many different points on the internet.
- **DNS** - The Domain Name Server maps domain names to IP addresses. For example, if you type 'dartmouth.edu' into a web browser, it will first make a DNS query to find out the IP address of Dartmouth's webserver so it can send the request to the right place.
- **DOM** - The Document Object Model is an interface for using and manipulating the data in an XML document.
- **Firewall** - Software that regulates who is allowed to connect to a system.
- **IP Address** - A unique number of the format xxx.xxx.xxx.xxx which identifies a connection to the internet. These are used in routing packets of data from one point on the internet to another.
- **PKI** - Public Key Infrastructure is a way of managing public key exchange through the use of certificates. The key component of PKI is a

Certificate Authority that manufactures new certificates and manages revocation lists.

- **TCP/IP** - The Transmission Control Protocol / Internet Protocol is the mechanism for transferring data over the internet. Data is divided into packets, sent through the infrastructure of the internet, then re-assembled at the receiving computer.

- **SAML** - The Security Assertion Markup Language is an OASIS standard defining the XML format for exchanging various pieces of information regarding access control.

- **SSL** - The Secure Socket Layer is a protocol to encrypt communications over the internet. See Appendix A.2 for a more detailed description.

- **XACML** - The eXtendable Access Control Markup Language is an OASIS standard defining the XML format for access control requests, response, and most importantly policies.

- **XML** - The eXtensible Markup Language is a way of encoding ”self describing data.” See Appendix A.1 for a more detailed description.

References

[Bas03] Microsoft Knowledge Base. Description of the secure sockets layer (ssl) handshake, 2003. http://support.microsoft.com/default.aspx?scid=kb;

[Bea04] Mathias Bogaert and James P. Cakalic et al. Log4j project, 2004. http://logging.apache.org/log4j/docs/index.html.

[BL03] Fabrizio Boccola and Cristiano Leoni. Permis, 2003. http://www.permis.org/en/index.html.

[Bor99] Stephane Bortzmeyer. Free java : is native code the way?, 1999. http://lists.debian.org/debian-java/1999/debian-java-199911/msg00044.html.

[Can02] Scott Cantor. Readme.txt for opensaml, 2002. available from http://www.opensaml.org/.

[Can03] Scott Cantor. Re: Samlauthorizationdecisionquery to requestctx (xacml), 2003. https://mail.internet2.edu/wws/arc/mace-opensaml-users/2003-10/msg00002.html.

[Car01] Steven Carmody. Shibboleth working group overview and requirements document, 2001. http://shibboleth.internet2.edu/docs/draft-internet2-shibboleth-requirements-01.html.

[CER03] CERT. Cert(r) advisory ca-2003-26 multiple vulnerabilities in ssl/tls implementations, 2003. http://www.cert.org/advisories/CA-2003-26.html.

[Cla98] James Clark. Xp - an xml parser in java, 1998. http://www.jclark.com/xml/xp/.

[Cla02] Andy Clark. Wrappedoutputstream, 2002. http://www.psl.cs.columbia.edu/xues/ep-javadoc/psl/xues/ep/util/WrappedOutputStream.html.

[Cor03] Maxim/Dallas Semiconductor Corp., 2003. http://www.ibutton.com/ibuttons/java.html.

[DLP+01] J. Dyer, M. Lindemann, R. Perez, S. W. Smith, L. van Doorn, and S. Weingart. Building the ibm 4758 secure coprocessor. *IEEE Computer*, 34:57–66, 2001.

[Dor03] Thomas Dorner. Samlauthorizationdecisionquery to requestctx (xacml), 2003. https://mail.internet2.edu/wws/arc/mace-opensaml-users/2003-10/msg00000.html.

[DS96] Maura P. Doyle and Christopher M. Snyder. Information sharing and competition in the motor vehicle industry. 1996. http://www.federalreserve.gov/pubs/feds/1997/199704/199704pap.pdf.

[ea03] Scott Boag et al. Xalan - java version 2.5.2, 2003. http://xml.apache.org/xalan-j/index.html.

[EC02] Marlena Erdos and Scott Cantor. Shibboleth-architecture draft v05. Technical report, 2002. http://shibboleth.internet2.edu/docs/draft-internet2-shibboleth-arch-v05.pdf.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The ssl protocol version 3.0, 1996. http://wp.netscape.com/eng/ssl3/ssl-toc.html.

[Fou01] The Free Software Foundation. The gnu jaxp project, 2001. http://www.gnu.org/software/classpathx/jaxp/.

[Fou04a] The Apache Software Foundation. The jakarta site - apache tomcat, 2004. http://jakarta.apache.org/tomcat/.

[Fou04b] The Apache Software Foundation. Socket samples, 2004. http://xml.apache.org/xerces2-j/samples-socket.html.

[Fou04c] The Apache Software Foundation. Xerces2 java parser readme, 2004. http://xml.apache.org/xerces2-j/index.html.

[GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. Soap version 1.2, 2003. http://www.w3.org/TR/soap12-part1/.

[GNU02] GNU. Guide to gnu gcj, 2002. http://gcc.gnu.org/onlinedocs/gcj/index.html.

[GPLMvdK03] Christian Geuer-Pollmann, Berin Lautenbach, Axl Mattheus, and Erwin van der Koogh. Apache xml security, 2003. http://xml.apache.org/security/index.html.

[Gre01] Anthony Green. gcj + xerces/xalan/tomcat/rhino/etc, 2001. http://gcc.gnu.org/ml/java/2001-08/msg00157.html.

[Gro02] Object Management Group. Corba basics. 2002. http://www.omg.org/gettingstarted/corbafaq.htm.

[Han01] Robert Hansen. Hardening htaccess. 2001. http://www.securityfocus.com/infocus/1368.

[Har03] Ann Harrison. Denial-of-service victims share lessons learned, 2003. http://www.cnn.com/2000/TECH/computing/06/16/dos.security.idg/index.html.

[Ker99] David Kern. A recent case study. 1999. http://www.aaas.org/spp/secrecy/Presents/Kern.htm.

[Lep03] Rebekah Lepro. Cardea: Dynamic access control in distributed systems. Technical Report NAS-03-020, NASA, 2003. www.nas.nasa.gov/Research/Reports/Techreports/2003/PDF/nas-03-020.pdf.

[Mic01a] Sun Microsystems. Class objectinputstream. Technical report, 2001. http://java.sun.com/j2se/1.3/docs/api/java/io/ObjectInputStream.html.

[Mic01b] Sun Microsystems. Interface serializable. Technical report, 2001. http://java.sun.com/j2se/1.3/docs/api/java/io/Serializable.html.

[Mic03a] Sun Microsystems. Class stringbufferinputstream, 2003. http://java.sun.com/j2se/1.4.2/docs/api/java/io/StringBufferInputStream.html.

[Mic03b] Sun Microsystems. Sun's xacml implementation. 2003. http://sunxacml.sourceforge.net/guide.html.

[Mic03c] Sun Microsystems. Synchronizing threads. Technical report, 2003. http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html.

[Mic04a] Sun Microsystems. J2se 1.5.0 beta 1, 2004. http://java.sun.com/j2se/1.5.0/index.jsp.

[Mic04b] Sun Microsystems. Java 2 platform, micro edition (j2me), 2004. http://java.sun.com/j2me/.

[Mic04c] Sun Microsystems. Java api for xml processing (jaxp), 2004. http://java.sun.com/xml/jaxp/index.jsp.

[Mic04d] Sun Microsystems. Java card technology, 2004. http://java.sun.com/products/javacard/.

[Mic04e] Sun Microsystems. Javabeans activation framework, 2004. http://java.sun.com/products/javabeans/glasgow/jaf.html.

[Mic04f] Sun Microsystems. Javamail api, 2004. http://java.sun.com/products/javamail/.

[MMea03] Eve Maler, Prateek Mishra, and Rob Philpott et al. Assertions and protocol for the oasis security assertion markup language (saml) v1.1. Technical report, OASIS, 2003. http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf.

[Moc87] P. Mockapetris. Doman names - implementation and specification (rfc 1035). Technical report, 1987. ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt.

[MSM04] J. Marchesini, S.W. Smith, and M.Zhao. Keyjacking: The surprising insecurity of client-side ssl. Technical Report 2004-489, Dartmouth College, 2004. http://www.cs.dartmouth.edu/ sws/abstracts/msz04.shtml.

[NS04] Sidharth Nazareth and Sean Smith. Using spki/sdsi for distributed maintenance of attribute release policies in shibboleth. Technical Report TR2004-485, Dartmouth College, 2004. http://www.cs.dartmouth.edu/reports/abstracts/TR2004-485/.

[Par02] Ian Parkinson. Custom ssl for advanced jsse developers, 2002. http://www-106.ibm.com/developerworks/java/library/j-customssl/.

[Pea02] Siani Pearson. Trusted computing platforms, the next security solution. 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-221.pdf.

[Per01] Denis Perchine. Problem compiling sun xml parser with gcj, 2001. http://gcc.gnu.org/ml/java/2001-02/msg00208.html.

[Pos82] J. Postel. Simple mail transfer protocol, 1982. http://www.freesoft.org/CIE/RFC/821/index.htm.

[PR85] J. Postel and J. Reynolds. File transfer protocol, 1985. http://www.freesoft.org/CIE/RFC/959/index.htm.

[RDBV$^+$03] Ed. R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic host configuration protocol for ipv6 (dhcpv6), 2003. ftp://ftp.rfc-editor.org/in-notes/rfc3315.txt.

[Roh01] Tim Rohaly. What is the correct order of objectinputstream / objectoutputstream... Technical report, 2001. http://www.jguru.com/faq/view.jsp?EID=333392.

[Wel04] Harald Welte. The netfilter/iptables project, 2004. http://www.netfilter.org/.

[YBP$^+$04] Franois Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (xml) 1.0 (third edition), 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

A Some Terminology

This will briefly explain some of the concepts and technologies that will be referred to throughout this paper.

A.1 XML

XML [YBP+04] is a means of encoding information to include both data and field names in one document. For example, a way of encoding your name and e-mail address in XML could be the following:

```
<name>Geoffrey Stowe</name>
<emailPrefix>geoffrey.stowe</emailPrefix>
<emailDomain>dartmouth.edu</emailDomain>
```

If a standard database were to store the same information, it would probably use a much simplified form such as:

```
Geoffrey Stowe,geoffrey.stowe@dartmouth.edu
```

If an application wishes to store data for its own retrieval, XML is clearly a wasteful approach because of the extra storage space it requires. However, it is useful in transmitting information between different systems because it does not rely on specific spacing, character encoding, etc. So, it is widely used on the internet where communication can take place between different programs running on different platforms used by people speaking different languages.

When XML is used in practice, specifically in the applications described in this paper, it must be parsed by a program which extracts relevant information and makes it available for use. Since all communications between the components of our model use XML, we encountered many complications due to XML-specific issues in implementing our model.

A.2 Public Key Cryptography and Certificates

Cryptography is the science of altering data in the hopes that only intended recipients can understand it. An example taken from the pages of spy novels could be one spy sending another a list of numbers like 12:56, 16:32, etc. The receiving spy would then find some prearranged book and take the 56th word on the 12th page, the 32nd word from the 16th page, etc. and reconstruct the message. This is called a symmetric code because both the sender and receiver need to have the same ”key” (which is a book in this case) in order to communicate.

In the late 1970's, three researchers at MIT named Ronald Rivest, Adi Shamir, and Leonard Adleman developed a revolutionary new method of encryption. It used two keys - one public and one private. If something is encrypted with the public key, it can only be decrypted with the private key, and vice versa. The benefit of this system is that anyone can know your public key. In order to send a message to someone, you find their public key (by asking them for it or by looking it up in a directory, for example) and use it to encrypt a message to them. Then, the recipient can use their key to decrypt the message. This process can also work in reverse - someone can encode a message using their private key. Then, anyone else can decode it using that person's freely available public key. This is called signing a document because anyone can verify that only a person with access to the private key could have produced the signed document.

While this system facilitated a new era of secure communication, it still has a basic flaw - how does one know a public key is legitimate? Say Alice wants to send a message to Bob, so she sends him an email asking for his public key. But, a clever intruder whom we'll call Trudy intercepts that message so Bob never gets it. Trudy then send Alice a different public key (for which she has the corresponding key) pretending to be Bob. Alice then encrypts the message with Trudy's public key and sends this to Bob. Trudy again intercepts the message and is able to decode it. This is a version of what is known as the "man in the middle" attack.

The solution to this attack is to employ what are known as certificates. A certificate is essentially a public key which some trusted authority has signed. A system of Certificate Authorities and other public key verification mechanisms is called a Public Key Infrastructure (PKI.) While browsing the internet, you have probably seen a message about a certificate verified by a company like Verisign, Entrust, or some other certificate authority. This is a way of verifying that you, the buyer, are using the proper public key to make a purchase. The actual mechanics of this communication are described in the next section.

A.3 SSL

If you've ever bought something online, chances are you've used SSL. [FKK96] Designed by Netscape, it is a protocol to allow for secure communication between a client and a server. It can authenticate both the client and the server, so both sides can be assured they are only communicating with an entity that holds a certain private key. A slightly simplified version of the process is the following: [Bas03]

- A client will contact a server and say it wants to use SSL for the upcoming transaction.
- The server then sends a certificate back to the client (remember that this certificate will contain the public key of the server.) The server can also request the client's certificate if it wishes to verify the identity of the client. If it does so, it will send a structured message including a random number which the client will use to validate itself.
- The client verifies that the server has sent a valid certificate, and it makes up a pseudo-random which will be used as the session key. It also signs the random number sent from the server and then sends this information along with its certificate to the server.
- The server receives the client's certificate and uses that public key to decrypted the signed number the client sent. If it matches the number the server originally sent, it knows with reasonable certainty that the

client does indeed possess the private key it claims to.

- At this point, both the client and server have the session key. They then use this to encrypt any information they want to send to each other.

The reason a session key is generated is that this allows for faster encryption. RSA is a costly algorithm relative to symmetric encryption, so SSL takes advantage of the strengths of both symmetric and asymmetric encryption. SSL is widely used because people are confident it is practically impossible to break. Although problems have been found in some implementations of SSL [CER03] the underlying protocol is sound. An attacker would presumably need to either forge a certificate or guess the random session key in order to eavesdrop on an SSL session. Both of these tasks are quite difficult if not impossible to do in any reasonable amount of time.

A.4 Denial of Service Attacks

A denial of service attack is basically a flood of data directed at some system on the internet. These attacks are commonly referred to as distributed denial of service (DDOS) attacks because often times they come from multiple sources. The key to a successful DDOS attack is to find a request you can send to a server that uses a non-negligible amount of system resources. For example, perhaps a poorly designed webserver will wait 1 second from the time an initial connection is made for the client to request a webpage. While it is waiting, it will not handle any other requests. More realistically, perhaps a webserver puts aside a small amount of memory every time a connection is made to prepare for transferring data. In either case, flooding the server with millions of bogus connection requests will stop it from functioning for legitimate users. In the first case, it will be constantly waiting in response to the bogus requests. In the second case, it will allocate all of its memory for the bogus requests and crash.

In February, 2000, a wave of DDOS attacks were launched against popular internet sites such as Yahoo, eBay, and ZDNet, shutting many of them down for hours at a time. Since then, webservers have employed sophisticated means of protecting against such attacks. [Har03] But although these common methods of launching DDOS attacks are now no longer as effective, the threat is still very real on the internet. When some particularly infectious viruses proliferate through the internet, the increased traffic has the effect of a DDOS attack and can shut down networks. For this reason any secure system should be designed with the possibility of a DDOS attack in mind.

B Choosing a Development Environment

I chose to use Java as my principle programming language because it was the language used for Sun's XACML implementation, and because I had the most experience with it in the recent past. To choose a development environment, I downloaded and tried out 4 Java IDE programs: NetBeans, Eclipse, Sun One, and Code Warrior. I decided to go with Eclipse for a variety of reasons:

- Sun One had a trial period after which it cost money

- Eclipse worked right off the bat unlike NetBeans, and the program is about a quarter the size of NetBeans
- Eclipse was recommended by a few peers
- I have used Code Warrior in the past, but have experienced a lot of unexplainable problems
- The Eclipse debugger seemed more robust than the debuggers included with Code Warrior or NetBeans
- Eclipse successfully imported the sample PDP and basic XACML policies and requests with a minimal amount of debugging

Eclipse ultimately saved me a lot of time by managing library imports, providing built-in support for Javadoc, and automatically finding errors and suggesting fixes. It is highly recommended to anyone embarking on a large-scale coding project.

C Other Attempts at Creating Object Streams

Before developing the method of exchanging RequestCtx objects that we use in our software, we tried several other approaches. I describe them here because they represent some common problems other researchers might encounter, and suggest areas for future improvement.

C.1 ObjectOutputStream

The first way we tried to send the object was with the native Java ObjectOutputStream and corresponding ObjectInputStream. These are subclasses of the regular InputStream and OutputStream, so they can theoretically send objects over a socket in much the same way that text can be sent. However, the object must be serializable by implementing the Serializable interface in order for the object streams to work. [Mic01a] Interestingly, the Serializable interface does not require implementing any methods, but rather "serves only to identify the semantics of being serializable." [Mic01b] RequestCtx objects do not implement the serializable interface, so instead of changing the core XACML code I created a wrapper class. This class ReqWrapper had a RequestCtx object as its only private variable. One interesting problem we ran into was an apparent bug in Java that requires ObjectOutputStream objects to be created before ObjectInputStream objects. [Roh01] However, even after fixing this problem we were unable to stream these wrapper functions, probably because all subclasses of a serializable class need to be serializable.

C.2 Streaming XML

Our next approach was to stream the XML of a RequestCtx object over a socket and rebuild the request to the other end. To accomplish this, we used a similar set-up as with the Object Streams: a stream running in each direction over a socket between the PEP and PDP. We then used the encode() function built into the request to stream its XML encoding to the PDP which rebuilt the request by calling the getInstance() method. We then planned to process the request and return the response over the same socket. However, in testing this process, the socket kept closing before sending the response back. We traced this problem to the native Java XML parser which is part of JAXP. [Mic04c] The method javax.xml.parsers.DocumentBuilder.parse() takes an InputStream as input, and it appears it closes that InputStream at some point. So, we were left with a catch-22. Getting the XML parser to work required closing the stream, but closing the stream resulted in the socket closing which meant we could not return a response.

Some research confirmed our suspicion and presented a possible solution: Xerces. Xerces is an XML parser for Java implemented by Apache. [Fou04c] Included in its packages is a class called socket.KeepSocketOpen which can be used to stream multiple XML files over a socket. It works by wrapping the stream in a WrappedInputStream [Cla02] and sending bytes over the socket to be recombined at the other end. [Fou04b] Using this approach the socket will not get closed after a single XML document has been sent. Therefore, we could probably could have used this model to implement our system. However, this would have required changing the core Sun XACML implementation as well as rewriting our communications paradigm. Also, since we were aiming to utilize SAML for sending our requests, we decided against pursuing Xerces-specific classes too extensively.

Our temporary solution was to close the socket connection after the request was sent. Then, the client would reconnect and receive the response to their request. This was a way to get around the problem of our socket closing, but it also provided the benefit of enabling a request queue in the PDP. Under this paradigm the client would "drop off" a request for the PDP to process, then come back when it needed the result. This would allow the PDP to prioritize requests and work on the most important ones first. Also, it could conceivably search more thoroughly to find attributes to support a request if it didn't have other requests to process. This approach, however, presents serious security problems, the most major of which is how the PDP ensures it returns a response to the same entity that made the request.

C.3 SOAP

One final approach we tried to improve upon the communication protocol currently in our code was to use the Simple Object Access Protocol (SOAP). The SOAP protocol is designed to exchange structured data between different processes running on different systems. [GHM+03] In other words, it solves many of the problems we encountered trying to stream XML directly over a socket.

To use SOAP, we decided to try setting up a SOAP server and a SOAP client. In the framework of our software, the PDP would function as a SOAP server, and it could process requests from either the service or the PEP. The SOAP server we chose to experiment with was Tomcat, an Apache project. [Fou04a] Installing the Tomcat servlet engine required adding the SOAP jar file and the JavaBeans Activation Framework [Mic04e]. It also required JavaMail [Mic04f] in order to support SMTP transfers which are required by SOAP. I never succeeded in resolving classpath errors in my Tomcat installation, so I did not integrate it with our framework. I also chose to stop pursuing the option because it did not fit explicitly with our communications model. If the PDP were the only "server" then both the service and PEP would have to initiate contact. Our model allows for the PDP to initiate contact with the PEP. And although the PEP needs to handle requests from the PDP, it should not be a server in the standard sense because it handles a limited range of requests from a small group of clients. Also, it will not return any information to the PDP.

Future access control implementations should consider using SOAP because of its versatility in transmitting XML data. However, the model of access control components may have to be altered to accomplish full integration with SOAP.

D Implementing in a Secure Environment

Even if the communication channels between a PDP and the outside world are secured through double-authenticated SSL encryption, a PDP can be compromised by attacking the platform it runs on. For example, if a PDP is running on an ordinary Windows desktop that an attacker has compromised, incoming requests could be intercepted and always accepted. Or, if the policies can be altered, decisions can no longer be considered valid. So, the PDP system should ideally be implemented on a secure platform that will ensure it makes consistent, reliable decisions.

I attempted to implement all my software on a secure platform, but was unable to find an appropriate system. I will first survey some of the secure hardware presently available, then discuss methods of implementing this PDP system on a secure platform.

D.1 Secure Hardware

The theory behind the field of trusted computing is to find ways of establishing trust with the system you're communicating with. It relies on some trusted hardware which can be used to create a basis of trust for software. Currently, secure hardware runs the gamut from tiny memory chips to large processors. This technology is still in its infancy, yet even now it can provide benefits to business. [Pea02]

Since our PDP system involves making decisions that cannot be tampered with, we would require a secure processor. If we only used cryptographic tokens, for example, we could ensure that policies were not tampered with between storing them on the system and reading them into the PDP. However, if we were not running the PDP on trusted platform, a policy could be tampered with once it is decoded but before it is processed by the actual PDP system. Also, the PDP system could be altered to produce fraudulent results. In a recent paper, John Marchesini describes a method to steal private keys from web browsers by using a DLL Inject. This will place an attacker between a program and its library functions allowing the attacker to view and possibly alter the functioning of a running process. [MSM04] This same approach could be used to tamper with the PDP.

D.2 Using the IBM 4758 Secure Coprocessor

There is essentially one option for a completely secure processor: the IBM 4758 secure coprocessor. This is an ordinary microprocessor housed in an environment that is both physically and electronically protected. The 4758 features battery-backed RAM so any unauthorized attempt to access the memory will erase anything in storage. This is just one of the hardware tamper responses that will detect intrusion attempts and automatically respond by erasing sensitive information. The boot process, and all subsequent operations, follow a careful sequence of trusted operations to ensure no unauthorized code is being executed. [DLP$^+$01] The end result is a processor that will execute precisely the software you load onto it, without intruders being able to detect what operations it is performing, no matter how hostile an environment it runs in.

The 4758 runs a stripped down version of the linux kernel, so my first efforts concentrated on using Java on the 4758. It does not have the capacity to run the full Java Virtual Machine, so it was not directly compatible with my software. However, Java produces other versions of the Java Runtime Environment, one of which is called the Java Micro Edition (JME). [Mic04b] Researchers at Dartmouth have succeeded in running this version on the 4758. However the JME does not include an XML-parser, so we could not use the JME directly.

If not everything can run inside the 4758, an alternative is to run part of the code on the secure platform and other parts on the host machine. For example, policies could be read in through the secure platform, compute a hash, sign the hash, and then stored externally. When the program needs to use them, it will read them in from the external source, compute the hash, then compare it to the signed hash. Only if these two match will the program proceed. However, we would need to at least perform XML parsing outside the secure environment leaving it vulnerable to the attacks described above. So although we could ensure that the xml text which the processor received has not been tampered with, we cannot ensure that the parsed XML tree is valid. The only way we could ensure this is by parsing the XML text inside a secure environment. It would seem reasonable for an XML parser to work with the Java Micro Edition, but we were unable to find one. Therefore, short of writing our own XML parser from the libraries in the JME, there was nothing we could do to get our system running securely in Java on the 4758.

D.3 Other Secure Hardware/Java Card

Java Card technology is being developed by Sun as a stripped down version of Java designed to run on cell phones, pay-tv boxes, or any type of smart card. [Mic04d] This Maxim/Dallas Semiconductor Corporation is currently producing a product called iButton. This is similar to a cryptographic token, but it includes the ability to run Java applets using Java Card technology. The iButton includes up to 134Kbytes of RAM for its Java applets and even runs a garbage collector. [Cor03] However, like the Java Micro Edition, Java Card does not support XML.

D.4 Compiling the Java Code

Since I could not run the PDP system in Java on the 4758, I shifted my efforts to compiling the Java code to native machine language. If successful, this would allow the code to run on the Linux kernel inside the 4758. Java is designed to compile to byte-code which can be run on any platform by the Java Virtual Machine. However, it is a programming language like any other and as such can theoretically be compiled to machine code. The only viable Java compiler is GCJ, developed by the same people who make the GCC compiler for C and C++. This compiler provides the capability to compile either Java source code or Java class files into native machine code. [GNU02] I had some limited success with gcj, but was not able to compile my entire project to native machine code. I was able to successfully compile and run a "Hello World" program as well as the PolicySet class I wrote and included in the PDPNode package. However, any classes that used another class (such as the PDPSet class which has a set of PDP elements) produced errors during gcj compilation. This is because the libraries they referenced were not compiled or not linked in properly. Like regular command line C or C++ compilers, libraries used in a program must be compiled separately and before compiling the main program. I was able to successfully compile a test class that involved an external class, but could not do this for any of the major parts of my project. To successfully compile the entire PDPNode package, I would need to compile the 9 libraries it uses.

Let's look at compiling the Sun XACML library as one example. The distribution includes a make file to build the code using javac. So, I changed the line that sets the default compiler from javac to gcj. When I made this, however, it produced hundreds of errors due to missing classes. So this one library would require tracking down hundreds of external references and compiling them all in the proper order before I could compile the entire library.

I next checked the internet to see if other people had experienced, and possibly overcome, similar problems. It appears problems involving gcj are somewhat common judging from internet message board postings. But gcj seems to especially have problems with XML parsers, some of which have not been resolved. [9] I found one reference to successfully compiling an XML parser called XP [Bor99], and one person who compiled JAXP, a Java XML parser developed by GNU. [Bor99] However, XP does not appear to support DOM which is required for SAML. [Cla98] JAXP also does not support DOM. [Fou01]

It appears Xerces has been successfully compiled [Gre01] which is encouraging because it is a major feature required by openSAML. However, it appears

[9]See for example [Per01]

Xalan has never been compiled,[10] and without it there is no hope of compiling openSAML.

I knew that even if I compiled everything, I still might have run into capacity problems when trying to run this on the 4758. XACML.jar is 175k, opensaml.jar is almost 100k, Xalan is over 1MB, log4j is almost 400k, xmlsec is another 350k, and Xerces.jar is another 1MB. This is over 3MB just for the libraries. This size may be feasible on the newer versions of IBM's cryptocards, but not on the original 4758.

This concluded my attempts at getting my PDP system to run on a secure platform. It is my hope that the next generation of secure hardware will be capable of running the Java Virtual Machine. Since more and more devices that humans interact with are running Java, its security and privacy will become increasingly important in daily life.

[10]See for example [Gre01]