TrustBuilder2 User Manual

Version 0.1

Adam J. Lee

Department of Computer Science
Univ. of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801
adamlee@cs.uiuc.edu

Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551
ajlee@sandia.gov

May 21, 2007

Abstract

This document gives a basic introduction to TrustBuilder2, a flexible framework for developing and experimenting with trust negotiation protocols and system components. The functionality of TrustBuilder2 can be easily extended by developing various plug-ins that can be loaded by the TrustBuilder2 runtime system and be used in place of, or in addition to, the system components provided with TrustBuilder2. This document discusses downloading, compiling, and installing TrustBuilder2; an architectural overview of the TrustBuilder2 runtime system; and an overview of the Clouseau policy language, which is currently supported by TrustBuilder2. We also walk through an example deployment of TrustBuilder2 that illustrates both how to configure the TrustBuilder2 runtime system and how a Java program can use TrustBuilder2 to facilitate the trust negotiation process.

1 Introduction

In trust negotiation, resource providers specify access control policies in terms of the attributes that should be possessed by authorized users, rather than the identities of these users. Users can prove ownership of certain attributes through the use of digital credentials issued by trusted entities. For example, the Department of Motor Vehicles might issue vehicle owners X.509 driver's licenses that can be used to demonstrate proof of their current age, address, or ability to drive. These types of digital credentials may also be protected by user-specified policies controlling their release to remote parties; for example, Alice might only be willing to show her driver's license credential to members of the Better Business Bureau. In this way, a request to access a given resource can result in a bilateral and iterative exchange of policies and credentials that represents a negotiation between the participating parties. To date, research interest in trust negotiation has been primarily theoretical and as such, implementations have been largely proofs of concept.

TrustBuilder2 is a flexible framework for supporting research in the area trust negotiation protocols, designed to allow researchers to quickly prototype and experiment with various approaches to trust negotiation. In TrustBuilder2, the primary components of a trust negotiation system are represented using abstract interfaces. These components include:

- *Strategy modules* used to determine the next step to take in a negotiation based upon both the contents of an incoming message (which may contain some number of credentials, policies, and other data items) and the history of the negotiation thus far.

- *Policy compliance checkers* used to determine whether a policy is satisfied by a given set of credentials and, if so, the ways in which this set of credentials satisfies the policy.
- *Query Interfaces* used to provide access to resources such as the local entity's credentials and policies, publicly available information repositories, or peer-to-peer query-processing networks.
- *Credential chain modules* used to construct and validate chains of credentials used during the negotiation process.
- *Audit modules* used to monitor the state of a negotiation, record status information, or visualize the negotiation process to enable a human user to easily interpret the process.

Any or all of these component interfaces can be implemented or extended by users of the TrustBuilder2 system, thereby making the system's functionality easily extensible. The TrustBuilder2 configuration files can then be modified to load these custom components in place of the default system components; this facilitates the use of new features without modifications to the underlying runtime system. In our implementation, we provide support for one trust negotiation strategy, a policy compliance checker based on Jess (the Java Expert System Shell), query interfaces enabling access to disk-based credential and policy repositories, a credential chain construction algorithm, two credential chain verification routines, and both graphical and text-based logging facilities. TrustBuilder2 also supports the interposition of user-defined plug-ins at communication points between system components to allow for easy monitoring of system activity or the modification of messages passed between components.

TrustBuilder2 is also agnostic with respect to the formats of credentials and policies used during the negotiation. Support for a new policy language can be added to the system by implementing a derived class of AbstractPolicyBrick, the abstract policy type supported by TrustBuilder2, and providing a compliance checker capable of determining whether a policy specified in this language is satisfied by a given collection of credentials (which are processed at an abstracted level). Similarly, support for new credential types can be added by implementing a derived class of AbstractCredentialBrick, the abstract credential type used by TrustBuilder2, and providing mechanisms for these types of credentials to be loaded by external repositories. The current version of TrustBuilder2 supports the use of an uncertified "credential" type that is useful for testing purposes; support for other credential formats can be added by writing or downloading plug-ins for TrustBuilder2.

The flexibility of TrustBuilder2 enables users not only to rapidly implement support for new features, but also provides a framework within which the trade-offs between various system configurations can be quantitatively analyzed. TrustBuilder2 allows users to keep the majority of system components constant and change only minor portions of the framework between trust negotiations; for example, a user could test the same negotiation using two different policy compliance checkers. This enables comparisons to be made between specific system components without requiring modification to the runtime system itself. The TrustBuilder2 framework provides an environment for researchers to begin considering the issues surrounding the deployment of trust negotiation protocols in production environments. All of these features will be discussed in greater detail in the later sections of this document.

Notation

In this document, the `typewriter font` is used when referencing locations in the file system (e.g., `config/`) or discussing method calls (e.g., `negotiate()`). The particular context for words typeset

in the typewriter font will always be made clear by the surrounding text. Commands that are meant to be typed at the system command prompt are typeset as follows, where *$* represents the system command prompt:

```
$ javac *.java
```

A Sans Serif font is used when discussing the classes that make up the TrustBuilder2 runtime system. For instance, the TrustMessage class is used to transport credentials, policies, and other TrustBricks between entities during a trust negotiation.

Outline

The remainder of this document is organized as follows. In Section 2, we explain how to download, compile, and install TrustBuilder2. Section 3 presents an architectural overview of the TrustBuilder2 runtime system. The primary system components of TrustBuilder2 are discussed, as are the data types used by these components, their interactions with one another, and the processes of extending or replacing these components. The details of the Clouseau policy language, which is currently supported by TrustBuilder2, are presented in Section 4. In Section 5 we walk through examples illustrating the configuration and use of the TrustBuilder2 system. We conclude in Section 6 with a list of further resources providing information about TrustBuilder2.

2 Obtaining TrustBuilder2

2.1 Downloading

TrustBuilder2 is available as open-source software under the license terms found in Appendix A. The most recent version of TrustBuilder2 can always be downloaded from the TrustBuilder2 web site at http://dais.cs.uiuc.edu/trustbuilder2. After downloading the TrustBuilder2 version 0.1 archive file for your platform (i.e., `tb2-0.1.tar.gz` for Unix systems or `tb2-0.1.zip` for Windows systems), extract the contents of the archive using an appropriate tool (e.g., `tar` or WinZip). Once the TrustBuilder2 archive is unpacked, you should have a directory named `TrustBuilder2_0-1` whose contents are described in Table 1.

File	Description
`bin/`	The output directory used by the Ant build file included in this archive. The TrustBuilder2 binary `tb2-0.1.jar` can be found in this directory.
`build.xml`	The Ant build file that can be used to generate `tb2-0.1.jar` from the source files contained in `src/`.
`config/`	General configuration files needed by TrustBuilder2.
`docs/`	All of the TrustBuilder2 documentation, including this file and the TrustBuilder2 Javadoc files, resides in this directory.
`examples/`	All source code, configuration files, policies, and credentials needed to run the examples presented in Section 5 of this document reside in this directory.
`lib/`	A directory containing libraries used by TrustBuilder2. When building TrustBuilder2 from source, this directory is automatically included in the Java classpath by `build.xml`.
`src/`	A directory containing the TrustBuilder2 source files.

Table 1: Contents of the TrustBuilder2 version 0.1 archive file.

Prior to using TrustBuilder2, you will also need to obtain a compatible version of the Java Development Kit; TrustBuilder2 requires JDK version 1.5 or higher. This software can be obtained from Sun Microsystems at http://java.sun.com/. In addition, you may need to obtain the following third-party libraries used by TrustBuilder2, each of which must be in the Java classpath:

The Bouncy Castle Cryptography API This library provides message digest functionality used by TrustBuilder2. It can be obtained free of charge from http://www.bouncycastle.org.

Jess The Java Expert System Shell (Jess), is an efficient expert system used as the foundation of the CLOUSEAU compliance checker, which is described in Section 4. Jess can be downloaded from http://www.jessrules.com; users not wishing to experiment with CLOUSEAU or build TrustBuilder2 from source will not need to obtain this library.

2.2 Compiling

The binary file `tb2-0.1.jar` is included with the TrustBuilder2 distribution in the `bin/` directory, so compilation of the TrustBuilder2 source is not necessary. However, users wishing to build the TrustBuilder2 binary from source can do so with the help of the Apache Ant program, which is freely available at http://ant.apache.org. Once Ant is installed and configured, simply issue the following command from within the `TrustBuilder2_0-1` directory:

```
$ ant build
```

This will rebuild the `tb2-0.1.jar` file and place it in the `bin/` directory. Note that the .jar files provided by the third-party libraries discussed above *must* be in the Java classpath for this build to succeed; this can be accomplished by adding these jar files to the `lib/` directory.

2.3 Installation

TrustBuilder2 has no formal installation procedure to speak of. To use TrustBuilder2 within another Java application, simply ensure that `tb2-0.1.jar` and any necessary third-party libraries are in the classpath used by your application. See Section 5.3 for an example of incorporating TrustBuilder2 with a Java application.

3 Architectural Overview

In this section, we provide a high-level overview of the TrustBuilder2 architecture. We begin with a discussion of the primary data structures used by TrustBuilder2, and then discuss the flow of data between TrustBuilder2's major system components. Readers desiring a more detailed treatment of the TrustBuilder2 architecture are encouraged to read the programmer documentation available in the `TrustBuilder2_0-1/docs/javadoc/` directory or online at http://dais.cs.uiuc.edu/trustbuilder2.

3.1 Data Types

At a high level, a trust negotiation session is an exchange of messages containing credentials, policies, uncertified claims, and other information between two parties. In TrustBuilder2, all types of information that might be exchanged between negotiating parties are represented as subclasses of the TrustBrick class, which forms the basic building block of a trust negotiation. In this way, users can extend the data types supported by TrustBuilder2 without modifying each component

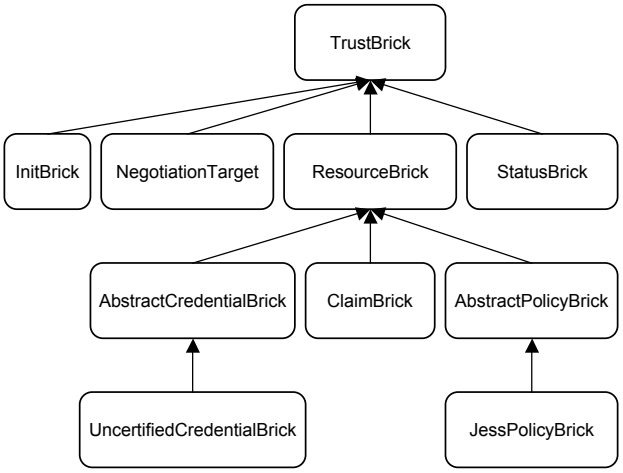

Figure 1: Class hierarchy for several important TrustBrick subclasses.

in the system; components can simply ignore TrustBricks that they do not know how to process, leaving them for other system components to handle. Entities then exchange TrustMessage objects containing one or more of these TrustBricks. Figure 1 shows the relationships between several important subclasses of TrustBrick. We now briefly discuss each of these classes.

TrustBrick All data items exchanged during a trust negotiation are of type TrustBrick. This class provides an abstract representation of data relevant to the trust negotiation process.

InitBrick InitBrick objects encode information regarding the supported configurations of TrustBuilder2, including the supported policy languages and credential types. The first message sent by each party during a trust negotiation session includes an InitBrick.

NegotiationTarget The second message sent by the initiator of a trust negotiation session includes a NegotiationTarget identifying the resource that the initiator wishes to access.

StatusBrick The last message sent during a trust negotiation includes a StatusBrick indicating whether the negotiation was successful.

ResourceBrick Any item exchanged during a trust negotiation that could possibly be protected by a release policy is a subclass of ResourceBrick. This ensures that TrustBuilder2 can properly enforce disclosure requirements on data items without necessarily understanding the data item itself.

AbstractCredentialBrick This class is an abstract representation of a credential that includes methods for operations such as examining the various fields of a credential, verifying the integrity of a credential, and checking proof of ownership values. This class allows compliance checkers to use any type of credential when attempting to satisfy a policy without requiring that they understand the representation of each credential type. Specific types of credential are encoded in subclasses of this type.

ClaimBrick ClaimBricks encode uncertified information provided by one party during the negotiation. For instance, phone numbers or email addresses might be included as claims used during a trust negotiation session.

AbstractPolicyBrick This class provides an abstract representation of a policy. Policies written in a specific language are represented as subclasses of this type (e.g., CLOUSEAU policies are encoded in JessPolicyBrick objects).

Readers interested in more detail regarding TrustBrick or its subclasses should consult the TrustBuilder2 programmer documentation.

3.2 Data Flow

Figure 2 presents a high-level architecture diagram of the TrustBuilder2 runtime system. We now describe each the major components identified in this diagram, and comment on the flow of data between components. Note that in Figure 2, components enclosed in dashed boxes are not included in the current version of TrustBuilder2; they are only meant to serve as example components that could be developed by users and added to the TrustBuilder2 data path.

TrustBuilder2 The TrustBuilder2 class provides the external interface to the TrustBuilder2 runtime system. Trust negotiation sessions are conducted through a series of calls to methods exposed by this class (see Section 5.3). Incoming TrustMessages are processed by the TrustBuilder2 object, which generates response TrustMessages.

IOManipulationModule The IOManipulationModule is the first class to process each incoming TrustMessage and the last class to process each outgoing TrustMessage. This component is capable of loading plug-ins that implement the IOManipulatorInterface interface at runtime. This allows users to develop custom components that can examine and modify the TrustMessages entering and leaving the TrustBuilder2 runtime system. The VisualizationModule is an example plug-in implementing this interface.

VisualizationModule The VisualizationModule provides an interface for writing and using custom logging and visualization components. Users can develop plug-ins for this module by implementing the VisualizerInterface interface. Once enabled, these plug-ins are then invoked whenever a call is made to the VisualizationModule. The GuiVisualizer is an example plug-in to the VisualizationModule that uses the Swing API to visualize the TrustMessages entering and leaving a TrustBuilder2 object.

StrategyModuleMediator After an incoming TrustMessage is processed by the IOManipulationModule, the TrustBuilder2 object passes it on to the StrategyModuleMediator. This class acts as the coordinator for a given trust negotiation session, as it dispatches incoming TrustMessages to the appropriate strategy module according to the strategy that was chosen for use during that session. Given an incoming TrustMessage and any state that it had previously stored, a strategy module decides what should be disclosed in a response to the remote party. Strategy modules can make any number of calls to the ComplianceCheckerMediator, CredentialChainMediator, or QueryEngineMediator components during their execution.

The StrategyModuleMediator supports the use of any number of strategy modules. Adding support for a new trust negotiation strategy involves writing a class encoding this strategy that implements the StrategyModuleInterface interface. Users can also write plug-ins that can access and/or modify any TrustMessages entering or leaving the StrategyModuleMediator. These plug-ins must simply implement the StrategyModuleCall or StrategyModuleReturn interfaces, respectively. Any number of such plug-ins can be loaded by TrustBuilder2.

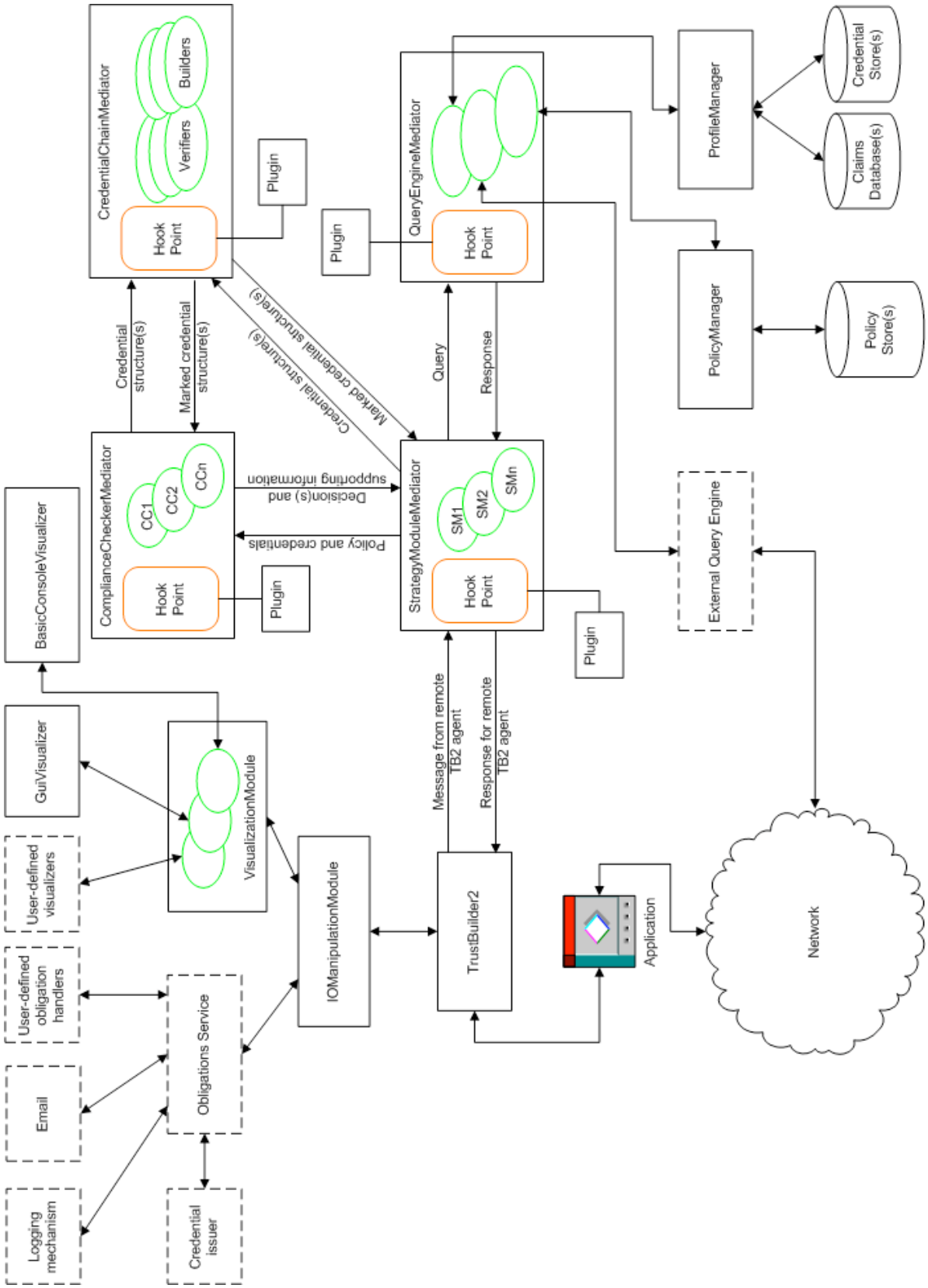

Figure 2: TrustBuilder2 architecture overview diagram.

ComplianceCheckerMediator The ComplianceCheckerMediator acts as a dispatch routine that enables access to the various compliance checkers currently loaded into the TrustBuilder2 runtime system. A compliance checker takes as input a policy, a set of credential chains, and a set of claims, and then determines whether the supplied credentials and claims satisfy the given policy. The ComplianceCheckerMediator chooses the compliance checker to use for a given set of inputs based on the type of the policy that its caller is trying to satisfy. TrustBuilder2 supports the use of both compliance checkers that find at most one satisfying set of credentials for a given policy and compliance checkers that find all satisfying sets of credentials for a given policy. See the programmer documentation for ComplianceCheckerInterface for more details on compliance checker modes of operation.

Compliance checkers are plug-in classes implementing the ComplianceCheckerInterface interface; any number of compliance checkers can be loaded by the ComplianceCheckerMediator. For a given policy language to be supported by TrustBuilder2, a compliance checker for that language must be loaded by the ComplianceCheckerMediator. Users can also develop plug-ins that can view and/or modify the inputs to or outputs of the ComplianceCheckerMediator by implementing the ComplianceCheckerCall and ComplianceCheckerReturn interfaces, respectively.

CredentialChainMediator This component is responsible for creating and verifying chains of credentials used during a trust negotiation. The CredentialChainMediator supports the use of any number of credential chain construction and verification algorithms; these can be implemented as user plug-ins conforming to the CredentialChainBuilderInterface and CredentialChainVerifierInterface interfaces, respectively. Users can also develop plug-ins that can view and/or modify the inputs to or outputs of the CredentialChainMediator by implementing the CredentialChainCall and CredentialChainReturn interfaces, respectively.

QueryEngineMediator This component accepts queries from other system components, processes these queries, and returns a result. All queries are subclasses of the AbstractQuery data type and are dispatched to an appropriate query engine based upon their query type. Any number of query engines can be loaded by the QueryEngineMediator and support for new query engines can be added by developing plug-ins that implement the QueryEngineInterface interface. Currently, QueryEngineMediator supports two query engines: the PolicyManager and the ProfileManager. Users can also develop plug-ins that can view and/or modify the inputs to or outputs of the QueryEngineMediator by implementing the QueryEngineCall and QueryEngineReturn interfaces, respectively.

PolicyManager The PolicyManager is responsible for loading a user's policies from disk and making them accessible to the rest of the TrustBuilder2 runtime system through the QueryEngineMediator. This component is very flexible in that it loads policies by scanning a specified directory for "loader files" that describe how to load various policies. These loader files specify a class to use to load a particular policy or group of policies, so adding support for new policy types or policy storage devices is as simple as writing a new plug-in class implementing the PolicyLoader interface. A more detailed explanation of how this loader mechanism works will be presented in Section 5.2.

ProfileManager The ProfileManager works much like the PolicyManager except that it is responsible for loading credentials and claims, not policies. It uses the same "loader file" mechanism as the PolicyManager to ensure extensibility and support for an arbitrary array of credential types and storage devices. The use of the ProfileManager will be illustrated in Section 5.2.

For the sake of brevity, not every component of TrustBuilder2 was discussed in this section. Readers desiring a more complete treatment of the components of the TrustBuilder2 system should consult the programmer documentation included with this distribution. Given this high-level overview of the TrustBuilder2 system architecture, we will now discuss the Clouseau policy language currently supported by TrustBuilder2.

4 The Clouseau Compliance Checker

The Clouseau compliance checker was designed to efficiently find all satisfying sets of credentials and claims for a given policy. It has been argued in the literature that a compliance checker capable of efficiently finding all satisfying sets would be advantageous, as it could allow entities to choose the most appealing way to satisfying a given policy [4]. For instance, if Alice can satisfy some policy by disclosing either a set of credentials C_1 that contains her security clearance credential (which she may consider sensitive), or a set of credentials C_2 that does not include this security clearance, it would be beneficial for Alice to discover both satisfying sets, as this would allow her to avoid disclosing her security clearance credential. Unfortunately, it has been shown that using existing compliance checkers to find all satisfying sets for a given policy is inefficient in the general case [5].

The Clouseau compliance checker is able to efficiently find all satisfying sets for a given policy because it reformulates this problem as an instance of the many pattern/many object pattern match problem. That is, if we treat policies as patterns and credentials and claims as objects, the problem of finding all satisfying sets for a given policy reduces to finding all ways that our "objects" can match the specified "patterns." The many pattern/many object pattern match problem has been studied by the artificial intelligence community due to its importance to the design of efficient expert systems. To leverage this research, the Clouseau compliance checker was built on top of Jess, the Java Expert System Shell [3]. Jess implements a variation of the Rete algorithm [1], which is an efficient algorithm for solving the many pattern/many object pattern match problem. In explaining the Clouseau language, we will omit details of Jess and the Rete algorithm; interested readers can find more details on these algorithms in [2, 3] and [1], respectively.

4.1 Representing Objects

Clouseau represents policies as Jess rules that place constraints on the credentials, credential chains, and claims needed to access various resources. Since Jess has no native support for handling these types of objects, we needed to specify the formats that they would take in Jess's working memory. The file `jess_defs.clp` included in the `config/jess` directory of the TrustBuilder2 distribution specifies these formats. The relevant portion of `jess_defs.clp` is shown below, while the entirety of this file is included as Appendix B.

Listing 1: Lines 5–25 of `jess_defs.clp`

```
5  ;; Simple credential template
   (deftemplate credential
     "Generic representation of credential fields"
     (slot id)
     (slot issuer)
10   (slot subject)
     (slot fingerprint)
     (slot owned (default false))
     (slot map (default (new java.util.HashMap))))
```

```
;; Even simpler credential chain representation
(deftemplate credential-chain
  "Generic representation of a credential chain"
  (multislot credentials))

;; Representation of a claim
(deftemplate claim
  "Used to hold uncertified claim information"
  (slot id)
  (slot type)
  (slot value))
```

Credentials are defined as objects containing a unique identifier, an issuer name, a fingerprint, a Boolean value indicating whether TrustBuilder2 could verify that the entity who disclosed this credential could demonstrate proof-of-ownership, and a data structure mapping the other field names used by the credential to their values. In the case of X.509 credentials, the issuer and subject fields of a credential would be X.500 distinguished names, and the fingerprint would be a hex-encoded MD5 hash of the credential. Credential chains are lists of unique credential identifiers: the first item in the list is the identifier of the root of the credential chain and each subsequent item n is the identifier of a credential whose parent is the credential represented by identifier $n-1$ in the list. Listing 2 describes Clouseau's internal representation of a credential chain consisting of two X.509 credentials. This chain describes a credential issued to `CN=Alice, O=FooCorp, C=USA` by `O=FooCorp, C=USA`, possibly describing a digital employee ID card.

Listing 2: The Clouseau representation of a simple credential chain.

```
(credential
  (id "Foo_Root")
  (issuer "O=FooCorp, C=USA")
  (subject "O=FooCorp, C=USA")
  (fingerprint "CE:54:41:9F:9A:3A:16:E7:93:87:6A:E3:C1:C2:0E:14")
  (owned false)
  (map ⟨Reference to a java.util.HashMap⟩))

(credential
  (id "Alice_Emp")
  (issuer "O=FooCorp, C=USA")
  (subject "CN=Alice, O=FooCorp, C=USA")
  (fingerprint "BB:65:9E:8A:0B:EC:EF:64:7C:2E:F0:4A:FE:E9:B1:42")
  (owned true)
  (map ⟨Reference to a java.util.HashMap⟩))

(credential-chain ("Foo_Root" "Alice_Emp"))
```

The Clouseau representation of uncertified claims is also defined in `jess_defs.clp` (see Listing 1). In this specification, claims are represented as Jess objects containing a unique identifier, a type, and a value. For example, Listing 3 describes a single claim containing a phone number.

Listing 3: The Clouseau representation of a phone number claim.

```
(claim
  (id "Bob_home_phone")
  (type "Phone Number")
  (value "555-555-5555"))
```

4.2 Defining Policies

While credentials, credential chains, and claims make up the objects that CLOUSEAU uses when solving the many pattern/many object pattern match problem, policies specify the patterns that are to be matched against. In CLOUSEAU, policies take the form of one or more Jess rules. The left hand side (LHS) of each rule places constraints on the credentials and claims required to satisfy the policy. The right hand side (RHS) can either assert some intermediate result that can be used by other rules making up the policy (see Section 5.2), or can assert that the policy has been satisfied. Asserting satisfaction requires that we define a "satisfaction" object type in `jess_defs.clp`; this definition is reproduced in Listing 4.

Listing 4: Lines 27–32 of `jess_defs.clp`

```
;; Used to represent a policy satisfaction
(deftemplate satisfaction
  "Holds information about policies that get satisfied"
30  (slot resource-name)
  (multislot claims)
  (multislot credentials))
```

A satisfaction object holds a name describing the resource whose access policy was satisfied, a collection of unique claim identifiers used to satisfy the policy, and a collection of unique credential identifiers used to satisfy the policy. Because the Rete algorithm underlies Jess, which is the foundation of CLOUSEAU, all possible satisfaction objects are generated simultaneously and returned to TrustBuilder2, making the CLOUSEAU compliance checker vastly more efficient than existing approaches to finding all satisfying sets for a given policy. Listing 5 illustrates an example CLOUSEAU policy.

Listing 5: An example CLOUSEAU policy.

```
(defrule rule-is-employee
  (credential (id ?i1) (issuer "O=FooCorp,C=USA")
              (subject "O=FooCorp,C=USA")
              (fingerprint "CE:54:41:9F:9A:3A:16:E7:93:87:6A:E3:C1:C2:0E:14"))

  (credential (id ?i2) (issuer "O=FooCorp,C=USA")
              (subject ?s2) (map ?m2) (owned true))
  (test (eq "Employee" (?m2 get "Type")))

  (credential-chain (credentials $?c))
  (test (is-root ?i1 ?c))
  (test (is-leaf ?i2 ?c))
=>
  (assert (satisfaction (resource-name "FileServer") (credentials ?c))))
```

This policy requires the existence of two credentials: FooCorp's root certificate and an employee ID. Note that the employee ID is required to be accompanied by a proof of ownership that has been verified by TrustBuilder2, while the FooCorp certificate is not. Further, this policy requires that TrustBuilder2 has verified that these two credentials are part of the same credential chain. The functions `is_root` and `is_leaf` are defined in `jess_defs.clp`, along with other helpful functions for reasoning about credential chains and other sets of credentials. If these necessary conditions are met, then CLOUSEAU will assert a satisfaction object indicating that access to "FileServer"

should be granted because the FooCorp root certificate and the FooCorp-issued employee ID were presented.

The above example is meant only to serve as an introduction to the syntax of CLOUSEAU; examples of more complex policies can be found in the next section of this document.

5 Using TrustBuilder2

In this section, we discuss the example programs distributed with TrustBuilder2. These examples illustrate the process of configuring TrustBuilder2, importing credentials and policies, enabling and disabling plug-ins, and integrating TrustBuilder2 with Java applications.

5.1 Compiling and Running the Examples

TrustBuilder2 comes with example client and server programs that carry out a trust negotiation over a TCP socket. To compile these example programs, open a command prompt and change directories to `TrustBuilder2_0-1/examples/`. If `tb2-0.1.jar` is located in Java's default classpath, simply type the following command to compile the examples:

```
$ javac *.java
```

If `tb2-0.1.jar` is not in Java's default classpath, the following command will compile the example programs:

```
$ javac -cp ../bin/tb2-0.1.jar *.java
```

To run an example negotiation between the client and server programs, you will need to have two command prompts open to the `TrustBuilder2_0-1/examples/` directory. In the first window, enter the following command to start the server process:

```
$ java StreamServer
```

In the second window, enter the following command to start the client process:

```
$ java StreamClient
```

If all goes well, you will see a "Negotiation success" message printed out by the client process after a slight pause. If an exception is raised while trying to run the example, it probably means that the TrustBuilder2 .jar file or some of the .jar files needed by Jess or the Bouncy Castle Cryptography API could not be found (see Section 2.1). Please obtain these .jar files and ensure that they are included in the Java classpath when trying to run the example again.

5.2 Examining the Example Configuration Files

We now explain the initialization process followed by the TrustBuilder2 runtime system by exploring the configuration files used to set up the example server and client programs provided with the TrustBuilder2 distribution. There are three main types of configuration files used by TrustBuilder2: primary configuration properties files, profile loader files, and policy loader files. We will now discuss each of these types of configuration files in greater detail.

Primary Configuration Properties Files

The file `examples/config/server/server.properties` (see Appendix C) is used by the server to provide the primary configuration parameters used to initialize a new TrustBuilder2 object. This file is a Java properties file that consists of a number of entries of the form $\langle key \rangle = \langle value \rangle$. A key describes some property that is to be set and is typically a fully-qualified class name followed by a "." followed by a property name. For instance, the value associated with the key `edu.uiuc.cs.TrustBuilder2.TrustBuilder2.root` specifies the root directory that TrustBuilder2 objects will use when searching for other files. The key/value pairs in `server.properties` serve three main purposes: specifying the configurations that are to be supported by TrustBuilder2, configuring system components loaded by default, and instructing default components to load and configure additional components.

A TrustBuilder2 configuration is a tuple that contains a single strategy, a list of supported credential types, and a list of supported policy languages. An entity can specify multiple supported configurations, though only one such configuration is used for each negotiation. During the exchange of InitBrick objects that takes place during a trust negotiation session, the participating entities agree on a system configuration that is mutually acceptable. Lines 24–27 of `server.properties` specify a single server configuration that the server is willing to support. Since only a single configuration was defined, the configuration key ends with `_1`. To define multiple configurations, configuration keys ending in `_2`, `_3`, ..., `_n` can be defined in addition to the configuration key ending in `_1`.

Listing 6: Lines 19–27 of `server.properties`

```
# A list of configuration options.  These are of the form:
20 #   <strategy module class>;<credential class 1>, ...,
#   <credential class n>;<policy class 1>, ..., <policy class m>
# Note that the numbers after the "configuration_" must start
# with 1 and increment without skipping any numbers
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.configuration_1 = \
25   edu.uiuc.cs.TrustBuilder2.strategy.MaximumRelevantStrategy; \
   edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredential; \
   edu.uiuc.cs.TrustBuilder2.compliance.jess.JessPolicyBrick
```

When a new TrustBuilder2 object is created, IOManipulationModule, SessionManager, StrategyModuleMediator, CredentialChainMediator, ComplianceCheckerMediator, and QueryEngineMediator objects are created by default. The `server.properties` file must specify configuration options needed by these components. For example, line 33 of `server.properties` disables the IOManipulationModule and lines 97–100 specify the query engines that are to be loaded by the QueryEngineMediator.

Listing 7: Lines 32–33 of `server.properties`

```
# is the IOManipulationModule enabled?
edu.uiuc.cs.TrustBuilder2.IOManipulation.IOManipulationModule.enabled = false
```

Listing 8: Lines 94–100 of `server.properties`

```
# Keys for the QueryEngineMediator.  This loads the ProfileManager,
95 # the NegotiationTargetMapper, and the PolicyManager
#
edu.uiuc.cs.TrustBuilder2.query.QueryEngineMediator.loadQueryEngines = \
```

```
  edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager, \
  edu.uiuc.cs.TrustBuilder2.query.policy.NegotiationTargetMapper, \
100 edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager
```

In addition to configuring the classes created by default, `server.properties` can also control the instantiation and configuration of other plug-in classes. For example, lines 102–133 control the configuration of the ProfileManager, NegotiationTargetMapper, and PolicyManager loaded by the QueryEngineMediator. The comments in this block of code describe the purpose of each key that is specified.

Listing 9: Lines 102–133 of `server.properties`

```
# ProfileManager stuff.
#
# The first key sets the directory that the ProfileManager will scan
105 # for loader files.  The second key tells the ProfileManager to load
# plug-ins capable of pulling in uncertified credentials and uncertified
# claims from disk.
#
edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager.loaderFileDir = \
110   config/server/profile_loaders
edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager.loadLoaders = \
  edu.uiuc.cs.TrustBuilder2.query.profile.ClaimLoader, \
  edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredentialLoader

115 # NegotiationTargetMapper stuff
#
# This tells the NegotiationTargetMapper where to look for the file
# binding NegotiationTarget names to the policies protecting those
# negotiation targets.
120 #
edu.uiuc.cs.TrustBuilder2.query.policy.NegotiationTargetMapper.mapFile = \
  config/server/mapfile.properties

# Policy Manager stuff
125 #
# Again, the first key tells the PolicyManager where to search for
# loader files.  The second key loads a plug-in that can read
# Clouseau policies from disk.
#
130 edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager.loaderFileDir = \
  config/server/policy_loaders
edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager.loadLoaders = \
  edu.uiuc.cs.TrustBuilder2.query.policy.JessPolicyFlatFileLoader
```

Due to its similarity to the `server.properties` configuration file, the client configuration properties file `examples/config/client/client.properties` will not be discussed in this manual, although it can be found in Appendix D. The reader is encouraged to examine the comments in `server.properties` and `client.properties` to gain a greater understanding of the other components that must be configured. The programmer documentation for each class also specifies every key that can be used to configure objects of that type.

Profile Loader Files

As was mentioned in Section 3.2, the ProfileManager relies on user-specified loader files to discover how to load a user's credentials and uncertified claims. These loader files are found at runtime by scanning a directory specified in the TrustBuilder2 configuration file; for example, see lines 109–110 of `server.properties` (Listing 9). Like the primary TrustBuilder2 configuration file, loader files are also Java properties files. Every loader file is required to define a value for the `loaderClass` key that specifies the ProfileManager plug-in class that will be used to process that loader file; all other keys specified in the loader file are dictated by the class specified as the value for the `loaderClass` key. As an example, the loader file `examples/config/server/profile_loaders/server_certs.ldr` is to be processed using the UncertifiedCredentialLoader class; this class loads test UncertifiedCredentialBricks from the local file system.

Listing 10: `examples/config/client/profile_loaders/charlie-docs.ldr`

```
loaderClass = edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredentialLoader
issuer = O=Acme Fabrication,C=USA
subject = CN=Charlie,O=Acme Springfield,C=USA
attr1 = Type
value1 = Sensitive Document Training
rid = docs_uncert
pid = policy_springfield_service
```

The above loader file loads an UncertifiedCredentialBrick that attests to the fact that Charlie has completed a sensitive document training course; this credential was issued by the fictitious machining company Acme Fabrication. Note also that the release of this credential is protected by the policy whose identifier is "policy_springfield_service." For more information about the exact format of these types of loader files, consult the programmer documentation for the UncertifiedCredentialLoader class.

Policies and Policy Loader Files

The PolicyManager loads policies in much the same way as the ProfileManager loads credentials and claims. That is, it searches a directory specified in the TrustBuilder2 configuration file for loader files instructing it of how to dispatch the actual loading of policies to the appropriate plug-in class. For example, on line 131 of `server.properties`, the PolicyManager is told that the server's policy loader files can be found in the `examples/config/server/policy_loaders` directory. Again, the format of these loader files is dictated by the class identified as the value for the `loaderClass` key in the file. Below is the loader file used by the server to read its policies from disk.

Listing 11: `examples/config/server/policy_loaders/policy.ldr`

```
loaderClass = edu.uiuc.cs.TrustBuilder2.query.policy.JessPolicyFlatFileLoader
preamble_1 = preamble.clp
policy_1 = policy_project_x.clp
rid_1 = policy_project_x
```

In this loader file, a single policy whose resource identifier is "policy_project_x" is loaded by the JessPolicyFlatFileLoader class. Note that this policy specifies a preamble, which is used much like a header file in a C program. Preambles contain frequently used policy fragments and allow common portions of policies to be shared between the policies specified for different resources

within an organization. The file `preamble.clp` (see Appendix E) specifies rules that check to see whether common credential chains, such as those including Acme Springfield employee IDs or Acme Fabrication issued sensitive document training certifications can be detected. If these types of credential chains are found, intermediate results are asserted that can be used in other policies to simplify the handling of these credential chains. For example, the fragment of `preamble.clp` specified below asserts a "sensitive-docs" object if a credential chain containing an Acme Fabrication issued sensitive document training certification credential can be found.

Listing 12: An excerpt from `examples/config/server/policy_loaders/preamble.clp`

```
(deftemplate sensitive-docs
  (slot subject)
  (multislot chain))

(defrule rule-has-sensitive-docs
  (credential (id ?i1) (issuer "O=Acme Fabrication,C=USA")
              (subject "O=Acme Fabrication,C=USA"))

  (credential (id ?i2) (issuer "O=Acme Fabrication,C=USA")
              (subject ?s2) (map ?m2))
  (test (eq "Sensitive Document Training" (?m2 get "Type")))

  (credential-chain (credentials $?c))
  (test (is-root ?i1 ?c))
  (test (is-leaf ?i2 ?c))
=>
  (assert (sensitive-docs (subject ?s2) (chain ?c))))
```

This preamble is concatenated with the specified policy file, `policy_project_x.clp`, prior to being used during a trust negotiation session. The contents of this policy are listed below.

Listing 13: `examples/config/server/policy_loaders/policy_project_x.clp`

```
;; Allow access to FTEs in 246x if have sensitive docs training
;;
(defrule fte-docs-246x
  (employeeId (empType "FTE") (org ?org) (chain $?c1))
  (test (and (<= 2460 ?org) (> 2470 ?org)))
  (sensitive-docs (chain $?c2))
=>
  (assert (satisfaction (resource-name project_x)
                        (credentials (dup-elim ?c1 ?c2)))))

;; Allow access if FTE of 24xx, has sensitive docs training, and
;; has an access exception issued by Alice or Bob
;;
(defrule fte-docs-24xx-exn
  (employeeId (empType "FTE") (org ?org) (chain $?c1))
  (test (and (<= 2400 ?org) (> 2500 ?org)))
  (sensitive-docs (chain $?c2))
  (accessException (project "Project X") (issuer ?iss) (chain $?c3))
  (or (test (eq "CN=Bob,O=Acme Springfield,C=USA" ?iss))
      (test (eq "CN=Alice,O=Acme Springfield,C=USA" ?iss)))
=>
  (assert (satisfaction (resource-name project_x)
                        (credentials (dup-elim ?c1 ?c2 ?c3)))))
```

Note that the above policy does not explicitly handle the examination of any credential chains, as this is all taken care of by the policy fragments specified in `preamble.clp`. Rather, this policy checks for the existence of intermediate results produced by `preamble.clp` and grants access to sensitive "Project X" resources if the requester can prove that they are either (i) a full-time employee of Acme Springfield that has an Acme Fabrication issued sensitive document training certification and works in department 2460–2469 or (ii) a full-time employee of Acme Springfield that has an Acme Fabrication issued sensitive document training certification, works in department 2400–2499 and was granted an "access exception" for "Project X" by either Alice or Bob. To see an example of a policy that does not use a preamble, examine the files `one.ldr` and `springfield_service.clp` in the `examples/config/client/policy_loaders` directory.

The Negotiation Target Mapfile

In addition to the three types of configuration files previously discussed, servers must also utilize another file that allows the NegotiationTargetMapper to associate an authorization policy with each of the resources that the TrustBuilder2 object will protect access to. In the case of our example server, this file associates the policy "policy_project_x" with the resource identifier "project_x." The contents of this mapfile, which is identified by lines 122 and 123 of `server.properties`, are listed below.

Listing 14: `examples/config/server/mapfile.properties`

```
# Properties file for the NegotiatonTargetMapper

target_1 = project_x
pid_1 = policy_project_x
```

5.3 Integrating TrustBuilder2 with a Java Application

Now that we have seen how the various configuration files and loader files needed by TrustBuilder2 are organized, it is time to explore the process of using TrustBuilder2 from within a Java application. We will start by examining the file `examples/StreamServer.java`, which can also be found in Appendix F.

Server Integration

The `StreamServer` class is a simple server process that awaits a single client connection, conducts a trust negotiation with the client, and then terminates. The first portion of `StreamServer.java` handles the process of creating the server's TrustBuilder2 object, accepting a client connection, and preparing the input and output streams that will be used to communicate with the client.

Listing 15: Lines 36–49 of `StreamServer.java`

```
36          // Create the server's TrustBuilder2 object using the configuration
            // file specified above.
            final TrustBuilder2 server = new TrustBuilder2(SERVER_CONFIG);

40          // Set up a server socket and wait for the client's connection
            ServerSocket socket = new ServerSocket(PORT);
            Socket clientSocket = socket.accept();
```

```
            // Once a client is connected, set up the input and output streams
            // used to communicate with the client.
            final ObjectInputStream input =
                new ObjectInputStream(clientSocket.getInputStream());
            final ObjectOutputStream output =
                new ObjectOutputStream(clientSocket.getOutputStream());
```

Notice that when creating a TrustBuilder2 object, one must provide it with a configuration file; in this case, the configuration file is the file `examples/config/server/server.properties` that we examined in Section 5.2. After the client has connected and communication channels have been established, the server blocks until it receives the first TrustMessage from the client. This message contains an InitBrick that the Server processes using its `processInitMessage()` method. InitBricks contain configuration information used to establish the parameters of a trust negotiation, including the types of credentials and policy languages supported by each party. The output of the `processInitMessage()` method is another TrustMessage containing an InitBrick which is then returned to the client. This allows the client to finalize its configuration.

Listing 16: Lines 52–65 of `StreamServer.java`

```
            // STEP 1: Block until the client sends the first TrustMessage of
            // the negotiation.
            inMsg = (TrustMessage)input.readObject();

            // STEP 2: This first message should contain an InitBrick that
            // specifies the client's portion of the configuration for this
            // session. The server processes this InitMessage using the
            // processInitMessage() function of TrustBuilder2. The result
            // of this function call is a result TrustMessage containing the
            // final configuration for this TrustNegotiation session. This
            // can then be sent to the client.
            outMsg = server.processInitMessage(inMsg);
            output.writeObject(outMsg);
            output.flush();
```

After the session configuration has been finalized with the client, the server blocks until it receives another TrustMessage from the client. This TrustMessage should contain a NegotiationTarget specifying the resource that the client wishes to negotiate for access to. If no NegotiationTarget is specified, the server terminates with an error message. If a NegotiationTarget is found, the server enters its main processing loop.

Listing 17: Lines 67–93 of `StreamServer.java`

```
            // STEP 3: Read in the first "real" message of the trust
            // negotiation from the client. This message will specify a
            // negotiation "target" indicating the resource that the client
            // wishes to access.
            inMsg = (TrustMessage)input.readObject();
            final NegotiationTarget target =
                StrategyUtils.getNegotiationTarget(inMsg);
            if(target == null){
                System.err.println("No negotiation target supplied");
                return;
            }
```

```
            // STEP 4:  This is the main loop of the negotiation.  At this
            // point, we process client inputs and send our responses out to
            // the client until one party decides that the negotiation is over.
            while(inMsg.getContinue() && outMsg.getContinue())
            {
                // process incoming message and send response
                outMsg = server.negotiate(inMsg);
                output.writeObject(outMsg);
                output.flush();

                // If the negotiation will proceed, get input
                if(outMsg.getContinue()){
                    inMsg = (TrustMessage)input.readObject();
                }
            }
```

In this loop, the server processes the most recently received TrustMessage from the client using its `negotiate()` method and sends the resulting TrustMessage back to the client. The first time through this loop, the `negotiate()` method locates the policy corresponding to the NegotiationTarget specified by the client and returns a TrustMessage containing this policy, which is then sent to the client. In subsequent invocations, it generates a response for the client that contains policies and credentials that are to be disclosed based upon how the server's strategy decides to respond to the policies and credentials disclosed by the client. As long as the server feels that the negotiation should continue, it reads in another message from the client and then returns to the top of this processing loop. When either entity decides that the negotiation cannot proceed further, the server terminates.

Client Integration

The integration of TrustBuilder2 with the client process is detailed in `examples/StreamClient.java` (see Appendix G). The client begins the trust negotiation process by creating an instance of TrustBuilder2 to use during the negotiation, establishing a connection to the server, and preparing the input and output streams that will be used for communications. The client then calls its `generateInitMessage()` method to generate a TrustMessage object containing an InitBrick that specifies the various configurations supported by the client. This TrustMessage is then sent to the server.

Listing 18: Lines 36–59 of `StreamClient.java`

```
            // Create the client's TrustBuilder2 object using the
            // configuration file specified above
            final TrustBuilder2 client = new TrustBuilder2(CLIENT_CONFIG);

            // Open up a socket to the server.  Once the socket is connected
            // wrap the input and output streams associated with that socket
            // using an ObjectInputStream and an ObjectOutputStream
            final Socket sock = new Socket(HOST, StreamServer.PORT);
            final ObjectOutputStream output =
                new ObjectOutputStream(sock.getOutputStream());
            final ObjectInputStream input =
                new ObjectInputStream(sock.getInputStream());

            long start = System.currentTimeMillis();
            TrustMessage inMsg, outMsg;
```

```
            // STEP 1:  The client should first generate an InitMessage that
            // will be sent to the server.  This message contains the client's
            // list of supported TrustBuilder2 configurations as well as
            // session-specific signature material.  See the JavaDoc for
            // edu.uiuc.cs.TrustBuilder2.message.InitBrick for more information.
            outMsg = client.generateInitMessage();
            output.writeObject(outMsg);
            output.flush();
```

At this point, the client blocks until it receives a TrustMessage back from the server containing an InitBrick that finalizes the configuration that will be used for this trust negotiation session. This message is processed using the client's `finishSessionEstablishment()` method, which fails if the configuration suggested by the server is not compatible with at least one of the configurations specified by the client. If this method succeeds, the client uses its `generateResourceRequest()` method to create a TrustMessage containing a NegotiationTarget that specifies the resource that the client would like to access. In this case, the client wants to access the resource whose identifier is "project_x." This TrustMessage is then sent to the server.

Listing 19: Lines 61–80 of `StreamClient.java`

```
            // STEP 2: We should now read in a response TrustMessage from the
            // server. This TrustMessage will contain an InitBrick sent by the
            // server containing enough information to establish the trust
            // negotiation session between the client and server.  We can now
            // finish establishing a session between the client and server.
            inMsg = (TrustMessage)input.readObject();
            if(!client.finishSessionEstablishment(inMsg)){
                System.err.println("Error establishing session");
                return;
            }

            // STEP 3:  Now we create a resource request to send to the
            // server.  This resource request indicates the "target" of our
            // trust negotiation.  In this case, we want to access a file
            // repository associated with "Project X" which is identified by
            // the resource identifier "project_x".
            outMsg =
                client.generateResourceRequest(inMsg.getSessionId(), "project_x");
            output.writeObject(outMsg);
            output.flush();
```

At this point, the client blocks until a response TrustMessage is received from the server and then enters its main processing loop. This loop works in exactly the same manner as the server's main processing loop that was described earlier.

Listing 20: Lines 82–101 of `StreamClient.java`

```
            // STEP 4:  Now, our trust negotiation session is fully established.
            // At this point, we will read in an input from the server.  We
            // then process the incoming TrustMessage using our local
            // TrustBuilder2 object and send the response to the server.  If
            // it looks like our negotiation will continue for another round,
            // we will then block until we receive another TrustMessage from
            // the server.
```

```
            inMsg = (TrustMessage)input.readObject();
            while(inMsg.getContinue() && outMsg.getContinue())
            {
                // process incoming message
                outMsg = client.negotiate(inMsg);
                output.writeObject(outMsg);
                output.flush();

                // if the negotiation will continue, get the next message
                if(outMsg.getContinue()){
                    inMsg = (TrustMessage)input.readObject();
                }
            }
```

Concluding Remarks

The above example illustrates the ease with which TrustBuilder2 can be integrated with Java applications. It should be noted that in the above example, the client and server processes communicate over a standard TCP socket, which is not encrypted. In actual deployments using TrustBuilder2, all communications should take place over an encrypted and authenticated channel, such as a TLS-enabled socket, to prevent eavesdropping and tampering attacks during a trust negotiation session. Further, developers should familiarize themselves with the programmer documentation for TrustBuilder2. It is important to understand the preconditions and postconditions for each method of TrustBuilder2, as well as the situations in which errors could arise prior to integrating TrustBuilder2 with other applications.

5.4 Hands-on Experiments

We now present two short tutorials explaining the effects of certain modifications to the TrustBuilder2 configuration files. In particular, we will explain how the client can enable the graphical logger to visualize the TrustMessages exchanged with the server and how to disable the release policies protecting the client's credentials. If you have not yet run the example client and server processes, as was discussed in Section 5.1, please do so now to verify that they are installed correctly.

Enabling Client-Side Visualization

To enable visualization using the GuiVisualizer we will need to edit the client configuration file `client.properties`. Note that lines 34–46 load not only the IOManipulationModule but also the VisualizationModule and the GuiVisualizer. However, line 35 disables the use of the IOManipulationModule, so although the GuiVisualizer is installed and loaded, it is never called.

Listing 21: Lines 34–46 of `client.properties`

```
# is the IOManipulationModule enabled?
edu.uiuc.cs.TrustBuilder2.IOManipulation.IOManipulationModule.enabled = false

# Which IOManipulators should be loaded by default? This should
# be a comma-separated list of fully qualified class names
edu.uiuc.cs.TrustBuilder2.IOManipulation.IOManipulationModule.load = \
  edu.uiuc.cs.TrustBuilder2.IOManipulation.viz.VisualizationModule

# The following visualization/logging modules will be loaded at
```

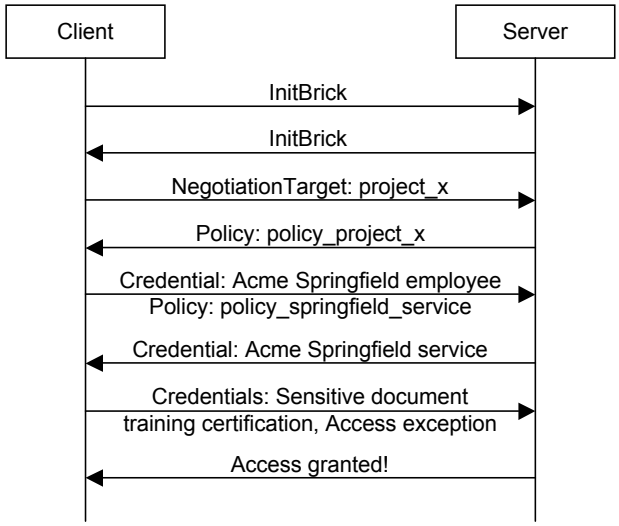

Figure 3: A simplified view of the example trust negotiation.

```
# startup by the VisualizationModule.  The value for this key should
# be a comma separated list of fully- qualified class names.
45 edu.uiuc.cs.TrustBuilder2.IOManipulation.viz.VisualizationModule.load = \
   edu.uiuc.cs.TrustBuilder2.IOManipulation.viz.GuiVisualizer
```

To enable visualization, simply edit line 35 of this file to set the "enabled" property of the IOManipulationModule to "true." Save the modified `client.properties` file and then run the example server and client applications. You should now see a visualization of each incoming and outgoing TrustMessage processed by the client's TrustBuilder2 object. Clicking on the icons representing InitBrick, AbstractPolicyBrick and AbstractCredentialBrick objects will show further details regarding the contents of these TrustBricks. The messages making up this trust negotiation session should have followed the same pattern as Figure 3. Note that only leaf credentials are indicated in Figure 3.

Disabling the Client's Release Policies

Recall from Section 5.2 that the release policy protecting a given credential is associated with the credential at the time that the ProfileManager loads the credential. One the client side, the ProfileManager uses the loader files in the `examples/config/client/profile_loaders` directory to load the client's credentials from disk. By commenting out the `pid` lines in the `charlie-docs.ldr` and `charlie-exn.ldr` files, we will ensure that the ProfileManager does not associate a release policy with the client's sensitive documents training or access exception credentials. After editing these files, run the example client and server again. If the visualizer is still enabled, you can verify that the messages exchanged by the client and server follow the same pattern as Figure 4.

6 Further Information

For general information regarding TrustBuilder2, including future development plans and publications related to TrustBuilder2, please consult the TrustBuilder2 web page at http://dais.cs.uiuc.edu/trustbuilder2. Those wishing to extend TrustBuilder2 or develop their own plug-ins

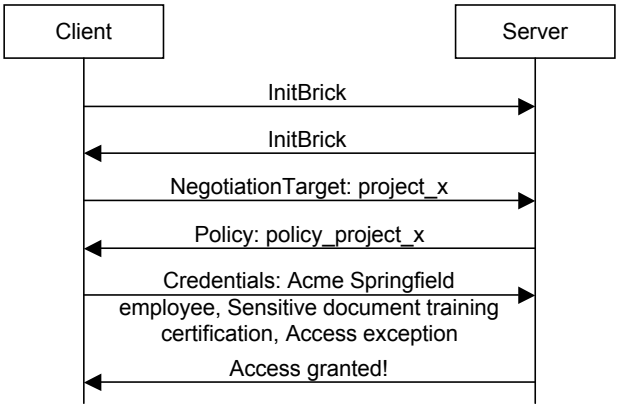

Figure 4: A simplified view of the example trust negotiation after the client's release policies have been disabled.

should be familiar with the TrustBuilder2 programmer documentation available in the `docs/javadoc` directory of the TrustBuilder2 distribution or online at http://dais.cs.uiuc.edu/trustbuilder2.

References

[1] C. L. Forgy. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[2] E. J. Friedman-Hill. *Jess in Action*. Manning Publications Co., 2003.

[3] E. J. Friedman-Hill. Jess, the rule engine for the Java platform. Web site, Jan. 2007. http://www.jessrules.com.

[4] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, June 2002.

[5] B. Smith, K. E. Seamons, and M. D. Jones. Responding to policies at runtime in TrustBuilder. In *5th International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, June 2004.

Appendix

A TrustBuilder2 Software License

Copyright (c) 2007 Sandia Corporation. Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains certain rights in this software.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Sandia Corporation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

B jess_defs.clp

```
;;;;;;;;;;;;;;;;
;; Data types ;;
;;;;;;;;;;;;;;;;

;; Simple credential template
(deftemplate credential
  "Generic representation of credential fields"
  (slot id)
  (slot issuer)
  (slot subject)
  (slot fingerprint)
  (slot owned (default false))
  (slot map (default (new java.util.HashMap))))

;; Even simpler credential chain representation
(deftemplate credential-chain
  "Generic representation of a credential chain"
  (multislot credentials))

;; Representation of a claim
(deftemplate claim
  "Used to hold uncertified claim information"
  (slot id)
  (slot type)
  (slot value))

;; Used to represent a policy satisfaction
(deftemplate satisfaction
  "Holds information about policies that get satisfied"
  (slot resource-name)
  (multislot claims)
  (multislot credentials))

;; Query that selects out satisfaction objects
(defquery get-satisfactions (satisfaction ))


;;;;;;;;;;;;;;;;;;;;;;;
;; Helpful functions ;;
;;;;;;;;;;;;;;;;;;;;;;;

;; Tests if the given item is in a list
(deffunction in-chain (?id $?chain)
  return (neq (member$ ?id ?chain) FALSE))

;; Tests if the given value is the nth item in a list
(deffunction is-nth (?id ?n $?ids)
  return (eq ?id (nth$ ?n ?ids)))

;; Is this the root of a credential chain?
(deffunction is-root (?id $?ids)
  (is-nth ?id 1 $?ids))

;; is this the leaf of a credential chain?
(deffunction is-leaf (?id $?ids)
  (is-nth ?id (length$ ?ids) $?ids))
```

```
;; Removes duplicate items from a list
(deffunction dup-elim ($?items)
  (bind ?new-list (create$ (nth$ 1 ?items)))
  (bind ?idx 2)
  (foreach ?i ?items
    (if (eq (member$ ?i ?new-list) FALSE) then
      (bind ?new-list (insert$ ?new-list ?idx ?i))
      (bind ?idx (+ ?idx 1))))
  (return ?new-list))
```

C server.properties

```
# Configuration file for our example server program
#

#-----------------------------------------------------------------
#----------------- GENERIC TrustBuilder2 Options -----------------
#-----------------------------------------------------------------

# TrustBuilder2 root directory.  This should contain a trailing
# "/" (for unix and mac) or "\\" (for windows)
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.root = ./

# The properties file that log4j will use.  This is relative
# to the above root path
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.log_config = config/log.properties

# The type of SecureRandom random number generator to use
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.SecureRandom = SHA1PRNG

# A list of configuration options.  These are of the form:
#   <strategy module class>;<credential class 1>, ...,
#   <credential class n>;<policy class 1>, ..., <policy class m>
# Note that the numbers after the "configuration_" must start
# with 1 and increment without skipping any numbers
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.configuration_1 = \
  edu.uiuc.cs.TrustBuilder2.strategy.MaximumRelevantStrategy; \
  edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredential; \
  edu.uiuc.cs.TrustBuilder2.compliance.jess.JessPolicyBrick


#---------- IOManipulation Options start here ----------

# is the IOManipulationModule enabled?
edu.uiuc.cs.TrustBuilder2.IOManipulation.IOManipulationModule.enabled = false



#---------- State management options start here ----------

# How long does an unused session last before it expires (in millis)?
edu.uiuc.cs.TrustBuilder2.state.SessionManager.expire = 3600000

# How often should we check for expired session objects (in millis)?
edu.uiuc.cs.TrustBuilder2.state.SessionManager.cleanup = 120000



#---------- Strategy module keys go here ----------

# Stuff for the StrategyModuleMediator
#
# This key loads the "Maximum Relevant Strategy" supplied with
# TrustBuilder2.
#
edu.uiuc.cs.TrustBuilder2.strategy.StrategyModuleMediator.loadStrategies = \
  edu.uiuc.cs.TrustBuilder2.strategy.MaximumRelevantStrategy
```

```
#---------- Credential chain processing keys start here----------

# Keys for the CredentialChainMediator
#
# The first key loads a single credential chain verifier that verifies
# credential chains starting at the chain root and working down the
# chain to the leaf.  The second key loads a credential chain building
# plug-in.
#
edu.uiuc.cs.TrustBuilder2.verification.CredentialChainMediator.loadVerifiers = \
  edu.uiuc.cs.TrustBuilder2.verification.RootToLeafVerifier
edu.uiuc.cs.TrustBuilder2.verification.CredentialChainMediator.loadBuilders = \
  edu.uiuc.cs.TrustBuilder2.verification.SimpleChainBuilder



#---------- Compliance checker keys start here----------

# Keys for the ComplianceCheckerMediator
#
# The first key loads the compliance checker for the Clouseau
# language.  The second key is actually a key for that compliance
# checker that tells it where to locate the base definitions file
# discussed in Section 4 of the TrustBuilder2 user manual.
#
edu.uiuc.cs.TrustBuilder2.compliance.ComplianceCheckerMediator.loadComplianceCheckers = \
  edu.uiuc.cs.TrustBuilder2.compliance.jess.JessComplianceChecker
edu.uiuc.cs.TrustBuilder2.compliance.jess.JessComplianceChecker.baseDefinitions = \
  config/jess/jess_defs.clp



#---------- Query processing keys start here----------

# Keys for the QueryEngineMediator.  This loads the ProfileManager,
# the NegotiationTargetMapper, and the PolicyManager
#
edu.uiuc.cs.TrustBuilder2.query.QueryEngineMediator.loadQueryEngines = \
  edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager, \
  edu.uiuc.cs.TrustBuilder2.query.policy.NegotiationTargetMapper, \
  edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager

# ProfileManager stuff.
#
# The first key sets the directory that the ProfileManager will scan
# for loader files.  The second key tells the ProfileManager to load
# plug-ins capable of pulling in uncertified credentials and uncertified
# claims from disk.
#
edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager.loaderFileDir = \
  config/server/profile_loaders
edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager.loadLoaders = \
  edu.uiuc.cs.TrustBuilder2.query.profile.ClaimLoader, \
  edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredentialLoader

# NegotiationTargetMapper stuff
```

```
#
# This tells the NegotiationTargetMapper where to look for the file
# binding NegotiationTarget names to the policies protecting those
# negotiation targets.
#
edu.uiuc.cs.TrustBuilder2.query.policy.NegotiationTargetMapper.mapFile = \
  config/server/mapfile.properties

# Policy Manager stuff
#
# Again, the first key tells the PolicyManager where to search for
# loader files.  The second key loads a plug-in that can read
# Clouseau policies from disk.
#
edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager.loaderFileDir = \
  config/server/policy_loaders
edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager.loadLoaders = \
  edu.uiuc.cs.TrustBuilder2.query.policy.JessPolicyFlatFileLoader
```

D client.properties

```
# Configuration file for example client program
#

#-----------------------------------------------------------------
#----------------- GENERIC TrustBuilder2 Options -----------------
#-----------------------------------------------------------------


# TrustBuilder2 root directory.  This should contain a trailing
# "/" (for unix and mac) or "\\" (for windows)
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.root = ./

# The properties file that log4j will use.  This is relative
# to the above root path
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.log_config = config/log.properties

# The type of SecureRandom random number generator to use
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.SecureRandom = SHA1PRNG

# A list of configuration options.  These are of the form:
#   <strategy module class>;<credential class 1>, ...,
#   <credential class n>;<policy class 1>, ..., <policy class m>
# Note that the numbers after the "configuration_" must start
# with 1 and increment without skipping any numbers
edu.uiuc.cs.TrustBuilder2.TrustBuilder2.configuration_1 = \
  edu.uiuc.cs.TrustBuilder2.strategy.MaximumRelevantStrategy; \
  edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredential; \
  edu.uiuc.cs.TrustBuilder2.compliance.jess.JessPolicyBrick



#----------- IOManipulation Options start here -----------

# is the IOManipulationModule enabled?
edu.uiuc.cs.TrustBuilder2.IOManipulation.IOManipulationModule.enabled = false

# Which IOManipulators should be loaded by default?  This should
# be a comma-separated list of fully qualified class names
edu.uiuc.cs.TrustBuilder2.IOManipulation.IOManipulationModule.load = \
  edu.uiuc.cs.TrustBuilder2.IOManipulation.viz.VisualizationModule

# The following visualization/logging modules will be loaded at
# startup by the VisualizationModule.  The value for this key should
# be a comma separated list of fully- qualified class names.
edu.uiuc.cs.TrustBuilder2.IOManipulation.viz.VisualizationModule.load = \
  edu.uiuc.cs.TrustBuilder2.IOManipulation.viz.GuiVisualizer



#----------- State management options start here -----------

# How long does an unused session last before it expires (in millis)?
edu.uiuc.cs.TrustBuilder2.state.SessionManager.expire = 3600000

# How often should we check for expired session objects (in millis)?
edu.uiuc.cs.TrustBuilder2.state.SessionManager.cleanup = 120000
```

```
#---------- Strategy module keys go here ----------

# Stuff for the StrategyModuleMEdiator
#
# This key loads the "Maximum Relevant Strategy" supplied with
# TrustBuilder2.
#
edu.uiuc.cs.TrustBuilder2.strategy.StrategyModuleMediator.loadStrategies = \
  edu.uiuc.cs.TrustBuilder2.strategy.MaximumRelevantStrategy



#---------- Credential chain processing keys start here ----------

# Keys for the CredentialChainMediator
#
# The first key loads a single credential chain verifier that verifies
# credential chains starting at the chain root and working down the
# chain to the leaf.  The second key loads a credential chain building
# plug-in.
#
edu.uiuc.cs.TrustBuilder2.verification.CredentialChainMediator.loadVerifiers = \
  edu.uiuc.cs.TrustBuilder2.verification.RootToLeafVerifier
edu.uiuc.cs.TrustBuilder2.verification.CredentialChainMediator.loadBuilders = \
  edu.uiuc.cs.TrustBuilder2.verification.SimpleChainBuilder



#---------- Compliance checker keys start here ----------

# Keys for the ComplianceCheckerMediator
#
# The first key loads the compliance checker for the Clouseau
# language.  The second key is actually a key for that compliance
# checker that tells it where to locate the base definitions file
# discussed in Section 4 of the TrustBuilder2 user manual.
#
edu.uiuc.cs.TrustBuilder2.compliance.ComplianceCheckerMediator.loadComplianceCheckers = \
  edu.uiuc.cs.TrustBuilder2.compliance.jess.JessComplianceChecker
edu.uiuc.cs.TrustBuilder2.compliance.jess.JessComplianceChecker.baseDefinitions = \
  config/jess/jess_defs.clp



#---------- Query processing keys start here ----------

# Keys for the QueryEngineMediator.  This loads the ProfileManager
# and the PolicyManager
#
edu.uiuc.cs.TrustBuilder2.query.QueryEngineMediator.loadQueryEngines = \
  edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager, \
  edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager

# ProfileManager stuff.
#
# The first key sets the directory that the ProfileManager will scan
# for loader files.  The second key tells the ProfileManager to load
```

```
# plug-ins capable of pulling in uncertified credentials and uncertified
# claims from disk.
#
edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager.loaderFileDir = \
  config/client/profile_loaders
edu.uiuc.cs.TrustBuilder2.query.profile.ProfileManager.loadLoaders = \
  edu.uiuc.cs.TrustBuilder2.query.profile.ClaimLoader, \
  edu.uiuc.cs.TrustBuilder2.test.UncertifiedCredentialLoader

# Policy Manager stuff
#
# Again, the first key tells the PolicyManager where to search for
# loader files.  The second key loads a plug-in that can read
# Clouseau policies from disk.
#
edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager.loaderFileDir = \
  config/client/policy_loaders
edu.uiuc.cs.TrustBuilder2.query.policy.PolicyManager.loadLoaders = \
  edu.uiuc.cs.TrustBuilder2.query.policy.JessPolicyFlatFileLoader
```

E preamble.clp

```
;;
;; See if they've had sensitive documents training
;;
(deftemplate sensitive-docs
  (slot subject)
  (multislot chain))

(defrule rule-has-sensitive-docs
  (credential (id ?i1) (issuer "O=Acme Fabrication,C=USA")
              (subject "O=Acme Fabrication,C=USA"))

  (credential (id ?i2) (issuer "O=Acme Fabrication,C=USA")
              (subject ?s2) (map ?m2))
  (test (eq "Sensitive Document Training" (?m2 get "Type")))

  (credential-chain (credentials $?c))
  (test (is-root ?i1 ?c))
  (test (is-leaf ?i2 ?c))
=>
  (assert (sensitive-docs (subject ?s2) (chain ?c))))


;;
;; Extract information about employees
;;
(deftemplate employeeId
  (slot subject)
  (slot empType)
  (slot org)
  (multislot chain))

(defrule rule-is-employee
  (credential (id ?i1) (issuer "O=Acme Fabrication,C=USA")
              (subject "O=Acme Fabrication,C=USA"))

  (credential (id ?i2) (map ?m2)
              (issuer "O=Acme Fabrication,C=USA")
              (subject "O=Acme Springfield,C=USA"))
  (test (eq "Branch Office" (?m2 get "Type")))

  (credential (id ?i3) (map ?m3) (subject ?s3)
              (issuer "O=Acme Springfield,C=USA"))
  (test (eq "Employee" (?m3 get "Type")))

  (credential-chain (credentials $?c))
  (test (is-leaf ?i3 ?c))
  (test (is-root ?i1 ?c))
  (test (is-nth ?i2 2 ?c))
=>
  (assert (employeeId (subject ?s3)
                      (empType (?m3 get "EmpType"))
                      (org (?m3 get "Org"))
                      (chain ?c))))


;;
```

```
;; Get information about exceptions
;;
(deftemplate accessException
  (slot subject)
  (slot issuer)
  (slot project)
  (multislot chain))

(defrule rule-has-exception
  ;; Exceptions come from a chain Acme->Acme Springfield->
  ;; Employee->Exception holder
  (credential (id ?i1) (issuer "O=Acme Fabrication,C=USA")
              (subject "O=Acme Fabrication,C=USA"))

  (credential (id ?i2) (map ?m2)
              (issuer "O=Acme Fabrication,C=USA")
              (subject "O=Acme Springfield,C=USA"))
  (test (eq "Branch Office" (?m2 get "Type")))

  ;; This is the exception-issuer's credential
  (credential (id ?i3) (map ?m3) (issuer "O=Acme Springfield,C=USA"))
  (test (eq "Employee" (?m3 get "Type")))

  ;; This is the exception itself
  (credential (id ?i4) (subject ?s4) (issuer ?iss4) (map ?m4))
  (test (eq "Exception" (?m4 get "Type")))

  ;; check constraints on chain
  (credential-chain (credentials $?c))
  (test (is-root ?i1 ?c))
  (test (is-nth ?i2 2 ?c))
  (test (is-nth ?i3 3 ?c))
  (test (is-leaf ?i4 ?c))
=>
  (assert (accessException (subject ?s4)
                           (issuer ?iss4)
                           (project (?m4 get "Project"))
                           (chain ?c))))
```

F StreamServer.java

```java
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

import edu.uiuc.cs.TrustBuilder2.TrustBuilder2;
import edu.uiuc.cs.TrustBuilder2.messages.NegotiationTarget;
import edu.uiuc.cs.TrustBuilder2.messages.TrustMessage;
import edu.uiuc.cs.TrustBuilder2.util.StrategyUtils;

/**
 * Server for a test application that needs to incorporate trust negotiation
 * using TrustBuilder2.  This server accepts client connections over a TCP
 * socket.  All communication between the client and server takes place using
 * ObjectInputStreams and ObjectOutputStreams that wrap the input and output
 * streams associated with the socket.
 *
 * @author Adam J. Lee (adamlee@cs.uiuc.edu)
 */
public class StreamServer
{
    /** The configuration file used by the server */
    public static final String SERVER_CONFIG = "./config/server/server.properties";

    /** The port that the server will listen on */
    public static final int PORT = 8080;

    /**
     * Main method, the server does it's bit in here...
     *
     * @param args
     */
    public static void main(String[] args)
    {
        try{
            // Create the server's TrustBuilder2 object using the configuration
            // file specified above.
            final TrustBuilder2 server = new TrustBuilder2(SERVER_CONFIG);

            // Set up a server socket and wait for the client's connection
            ServerSocket socket = new ServerSocket(PORT);
            Socket clientSocket = socket.accept();

            // Once a client is connected, set up the input and output streams
            // used to communicate with the client.
            final ObjectInputStream input =
                new ObjectInputStream(clientSocket.getInputStream());
            final ObjectOutputStream output =
                new ObjectOutputStream(clientSocket.getOutputStream());
            TrustMessage inMsg, outMsg;

            // STEP 1:  Block until the client sends the first TrustMessage of
            // the negotiation.
            inMsg = (TrustMessage)input.readObject();

            // STEP 2:  This first message should contain an InitBrick that
```

```java
            // specifies the client's portion of the configuration for this
            // session.  The server processes this InitMessage using the
            // processInitMessage() function of TrustBuilder2.  The result
            // of this function call is a result TrustMessage containing the
            // final configuration for this TrustNegotiation session.  This
            // can then be sent to the client.
            outMsg = server.processInitMessage(inMsg);
            output.writeObject(outMsg);
            output.flush();

            // STEP 3:  Read in the first "real" message of the trust
            // negotiation from the client.  This message will specify a
            // negotiation "target" indicating the resource that the client
            // wishes to access.
            inMsg = (TrustMessage)input.readObject();
            final NegotiationTarget target =
                StrategyUtils.getNegotiationTarget(inMsg);
            if(target == null){
                System.err.println("No negotiation target supplied");
                return;
            }

            // STEP 4:  This is the main loop of the negotiation.  At this
            // point, we process client inputs and send our responses out to
            // the client until one party decides that the negotiation is over.
            while(inMsg.getContinue() && outMsg.getContinue())
            {
                // process incoming message and send response
                outMsg = server.negotiate(inMsg);
                output.writeObject(outMsg);
                output.flush();

                // If the negotiation will proceed, get input
                if(outMsg.getContinue()){
                    inMsg = (TrustMessage)input.readObject();
                }
            }

            System.out.println("Negotiation completed.");

        }

        // Uh oh!
        catch(Exception e){
            System.err.println("Error!");
            e.printStackTrace(System.err);
        }

    }  //-- end main(String[])

}  //-- end class StreamServer
```

G StreamClient.java

```java
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

import edu.uiuc.cs.TrustBuilder2.TrustBuilder2;
import edu.uiuc.cs.TrustBuilder2.messages.StatusBrick;
import edu.uiuc.cs.TrustBuilder2.messages.TrustMessage;

/**
 * Client for a test application that needs to incorporate trust negotiation
 * using TrustBuilder2.  This client connects to the server over a TCP socket.
 * All communication between the client and server takes place using
 * ObjectInputStreams and ObjectOutputStreams that wrap the input and output
 * streams associated with the socket.
 *
 * @author Adam J. Lee (adamlee@cs.uiuc.edu)
 *
 */
public class StreamClient
{
    /** The configuration file used by the client */
    public static final String CLIENT_CONFIG = "./config/client/client.properties";

    /** Host to talk to */
    public static final String HOST = "127.0.0.1";


    /**
     * Drives the demo
     *
     * @param args
     */
    public static void main(String[] args)
    {
        try{
            // Create the client's TrustBuilder2 object using the
            // configuration file specified above
            final TrustBuilder2 client = new TrustBuilder2(CLIENT_CONFIG);

            // Open up a socket to the server.  Once the socket is connected
            // wrap the input and output streams associated with that socket
            // using an ObjectInputStream and an ObjectOutputStream
            final Socket sock = new Socket(HOST, StreamServer.PORT);
            final ObjectOutputStream output =
                new ObjectOutputStream(sock.getOutputStream());
            final ObjectInputStream input =
                new ObjectInputStream(sock.getInputStream());

            long start = System.currentTimeMillis();
            TrustMessage inMsg, outMsg;

            // STEP 1:  The client should first generate an InitMessage that
            // will be sent to the server.  This message contains the client's
            // list of supported TrustBuilder2 configurations as well as
            // session-specific signature material.  See the JavaDoc for
            // edu.uiuc.cs.TrustBuilder2.message.InitBrick for more information.
```

```java
            outMsg = client.generateInitMessage();
            output.writeObject(outMsg);
            output.flush();

            // STEP 2: We should now read in a response TrustMessage from the
            // server. This TrustMessage will contain an InitBrick sent by the
            // server containing enough information to establish the trust
            // negotiation session between the client and server.  We can now
            // finish establishing a session between the client and server.
            inMsg = (TrustMessage)input.readObject();
            if(!client.finishSessionEstablishment(inMsg)){
                System.err.println("Error establishing session");
                return;
            }

            // STEP 3:  Now we create a resource request to send to the
            // server.  This resource request indicates the "target" of our
            // trust negotiation.  In this case, we want to access a file
            // repository associated with "Project X" which is identified by
            // the resource identifier "project_x".
            outMsg =
                client.generateResourceRequest(inMsg.getSessionId(), "project_x");
            output.writeObject(outMsg);
            output.flush();

            // STEP 4:  Now, our trust negotiation session is fully established.
            // At this point, we will read in an input from the server.  We
            // then process the incoming TrustMessage using our local
            // TrustBuilder2 object and send the response to the server.  If
            // it looks like our negotiation will continue for another round,
            // we will then block until we receive another TrustMessage from
            // the server.
            inMsg = (TrustMessage)input.readObject();
            while(inMsg.getContinue() && outMsg.getContinue())
            {
                // process incoming message
                outMsg = client.negotiate(inMsg);
                output.writeObject(outMsg);
                output.flush();

                // if the negotiation will continue, get the next message
                if(outMsg.getContinue()){
                    inMsg = (TrustMessage)input.readObject();
                }
            }

            long end = System.currentTimeMillis();

            // STEP 5:  Check the status of our negotiation with the server
            // and print out a message.
            final StatusBrick status = inMsg.getStatus();
            if( (status != null) && (status.getTrustEstablished()) ){
                System.out.println("Negotiation success.");
            }
            else{
                System.out.println("Negotiation failure.");
            }

            // close connection
```

```
            sock.close();

            // print out how long it took
            System.out.println("Negotiation took " + (end - start) + " ms.");
        }

        // Oops!
        catch(Exception e){
            System.err.println("Error!");
            e.printStackTrace(System.err);
        }


    }  //-- end main(String[])

}  //-- end class StreamClient
```