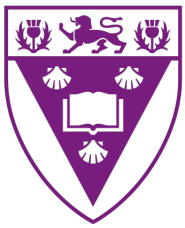


RHODES UNIVERSITY

Where leaders learn

Multi-stage Security for Physical Access Control

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Gregory Charles Michael Linklater

Grahamstown, South Africa
October 2016

Abstract

Attribute-based access control (ABAC) is a model where access to or authorization for a particular action on a particular resource is determined by evaluating attributes of the subject, resource (object) and environment against arbitrarily complex boolean rules. The current de facto standard for ABAC is XACML, a standard by OASIS, initially published in 2003 and updated in 2005 and 2013. The XACML standard details an ABAC rule language based off XML/RDF and an architecture involving five subsystems that support the access control flow. XACML is inaccessible to most due to its complex implementation and high cost, both in required initial hardware resources and licensing fees. To address this, a proof of concept access control system that uses JSON as its base serialized data representation and relies on modern web technologies such as OpenID Connect and JWT was created. Evaluating the system will find that ABAC with JSON is possible, more bandwidth efficient than XML/RDF and does not require disproportionate hardware resources. This research will contribute to the future development of a production-ready ABAC mechanism for modern internet-enabled devices and resources.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (2012 version, valid through 2016)[1]:

- **Security and privacy**
- **Security and privacy → Access control**
- **Security and privacy → Authorization**
- *Security and privacy → Systems security*
- *Software and its engineering → Software prototyping*

[1] (Association for Computing Machinery, 2012)

Acknowledgements

First, I would like take this opportunity to thank my supervisor Mr. James Connan, for continuously mentoring and taking risks on an undergraduate who would let him down many times before finally succeeding. I am grateful for years of advice stemming from "uncommon common sense", sympathy if not understanding in stressful times and your fantastic sense of humour, albeit often at my expense. You have provided me with freedom throughout my university career which allowed me the opportunity to fail and learn from my failures and the advice to get me back on the right track when I do. I considered you to be my supervisor long before this year and I thank you for teaching me that which cannot be taught.

Second, I would like to thank my father, Michael Linklater for his support, adept editing and comical cometary on computer scientist's lack of creativity in naming things.

Third, I would like to thank Mr. Christian Smith, for being a friend, mentor and willing co-conspirator in dreams of world domination.

Fourth, I would like to thank the staff of the Hamilton Building, who have always greeted me with a smile.

Finally I would like to dedicate all of the time and effort that went into creating this document to my loving fiancé Megan, without whom, I would be a shadow of who I am today. Thank you for the warm hug in times of sadness, encouraging word in times of weakness, loving smile in times of joy and stern look in times of procrastination.

"Behind every great man, there is a greater woman."

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs/CORIANT, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 90243).

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Access Control

Access control is a mechanism for ensuring that limited or otherwise sensitive resources are shared only with those who are authorized to access or use them. Many access control models exist, however the topic of this research is focused on attribute-based access control.

1.1.1 Derivative Access Control Models

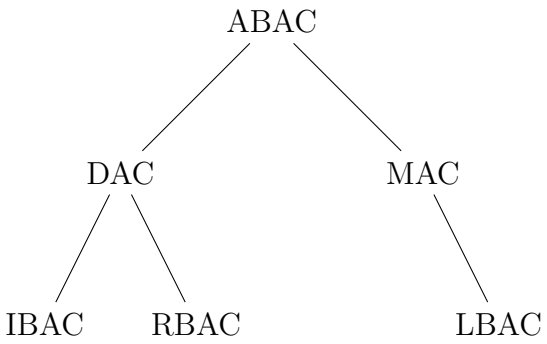


Figure 1.1: Non-Exhaustive Access Control Model Hierarchy

Figure 1.1 shows a non-exhaustive hierarchy of access control models. The common trend in all derivative access control models is that each distinct combination of action and

resource is somehow mapped to an individual. Either directly, as in the case of identity-based access control or through the abstraction of a role, as in the case of role-based access control.

Attribute based access control has no such concept and rules are defined to interact directly with the attribute or data itself.

1.1.2 What is Attribute-based Access Control?

Attribute-based Access Control (ABAC) is an access control model that allows resources to be shared securely by determining a user's authorization based on attributes of the subject, resource and environment.

Within attribute-based access control, an attribute can be defined as any serializable piece of data. Attributes of the subject relate to the identity of the entity requesting access. Attributes of the object relate to the resource that is being protected and are specified by the resource owner. Attributes of the environment can be any attribute that is not contained by one of the other two categories within attribute-based access control. Examples of this could include: the time, a global emergency switch or the atmospheric pressure at the north pole.

The attribute based access control model stipulates that attribute-based access control rules need to be functionally complete. This is necessary so rules can combine any combination of arbitrarily complex boolean conditions to grant a subject access to the resource.

1.1.3 Why Attribute-based Access Control?

Attribute-based access control provides a number of benefits over the derivative access control models, the most relevant of which are shown here:

- As shown in Figure 1.1, attribute-based access control is the root access control model. It is the superset of all other access control models and as such can provide the same functionality as all the other access control models *simultaneously* by specifying it in the rules. This relies on the relevant attributes of the model being present and available for use.

- Due to attribute-based access control lacking the abstraction of a permission or authorization, managing access in a large deployment is much simpler compared to other access control models. Instead of granting and revoking access to a particular resource, rules are written for the resource to allow access to those with the relevant attributes. All that is left to do from an access management standpoint is to maintain the attributes of the subjects.

1.2 Problem Statement

Physical access control is a problem that has been automated by technology for the last twenty five years, during which many access control paradigms have been developed and improved upon; the most recent and most comprehensive of which is attribute based access control (Hu *et al.*, 2014). To the best of the knowledge of the author, there are no standards that have been defined that implement attribute based access control aside from XACML, which was designed to integrate well with security assertion markup language (SAML) (Hallam-Baker, 2001) based identity providers such as Shibboleth. XACML has a high barrier to entry as it costs a lot to operate in terms of human resources (specialists), hardware and licensing fees for proprietary implementations. This results in the relative utility and power of attribute-based access control being reserved for large organisations that have the resources to afford it.

1.3 Research Motivation

This research is motivated by three factors:

- The lack of a simple, free and open-source implementation of attribute-based access control.
- XML/RDF — the base serialized data representation from which XACML specifies its configuration, communication and rule language — is a verbose and difficult to use standard.
- Attribute-based access control could be used for a number of applications in distributed identity.

1.4 Research Objectives

The aim of this research is to create a proof of concept attribute based access control system that will allow a resource authority to define arbitrarily complex access control policies. The research will focus primarily on migrating existing functionality in attribute based access control systems and specifications to use JSON as a base serialized data representation. Additionally, simplifying the architecture presented by XACML for use in a small deployment scenario is of primary concern.

The final system should be able to enact attribute-based access control in a manner that is equivalent or better than XACML with specific regard to core functionality (evaluating attribute-based access control rules against raw data) and security; all supporting infrastructure required to do this is included in this goal. The system should ultimately decrease the barrier to entry to attribute-based access control by simplifying it enough that it can operate on a single host.

1.5 Research Questions

The research objectives can be synthesised into the following research question:

Can a simple, modern attribute-based access control system be built for use in small deployments?

As part of answering this question, the following questions arise:

1. Is it possible to enact attribute-based access control for a single resource deployment?
2. Is it possible to enact attribute-based access control using JSON and other modern web standards?
3. Is it possible to enact attribute-based access control with a smaller architecture than XACML?

1.6 Approach

To create such a system, this research will be making use of and modifying many existing technologies and standards; in particular– an implementation of an existing OpenID Connect identity provider and JSON Web Token (JWT) and related standards. Identity providers, such as OpenID Connect implementations, securely store and control access to attributes of the subject which are required for an attribute-based access control system (Sakimura *et al.*, 2014). JWTs are used to encode data in a JSON format such that the origin and intended destination for the data is cryptographically verifiable (Jones *et al.*, 2012). This research will entail building a new attribute-based access control system that makes use of these technologies and is simple enough to be viable for small deployments.

1.7 Document Overview

The remainder of this document is organised according to the following structure:

- **Chapter 2** - *Literature Survey:* This chapter features the review, analysis and synthesis of relevant research documentation on the topics of: authentication, authorization, security practices and requirements, access control models and implementations and other supporting technologies that will be useful in completing the objectives of this research.
- **Chapter 3** - *Design:* This chapter proposes a design for a proof of concept, attribute-based access control system. Various success criteria, design considerations and alternative methodologies are discussed and the modular library structure is introduced.
- **Chapter 4** - *Implementation:* This chapter presents the final implementation of the system and all work completed as well some difficulties encountered and the design changes that were made to accommodate them.
- **Chapter 5** - *Results & Analysis:* This chapter presents the output of the system and the analysis thereof against the criteria for success defined in Chapter 3.
- **Chapter 6** - *Conclusions:* This chapter makes use of the analysis, discussion and results in Chapter 5 to make inferences about the research and evaluate them against the research objectives defined in this chapter. This chapter also presents possible derivative work to extend or address limitations of this research.

- **Appendix A** - *Logs:* This appendix displays the configuration, rules and logging output — both in JSON (the default logging format) and pretty printed for better human legibility — for the controlled test deployment of the system.
- **Appendix B** - *Test Output:* This appendix displays the test output and coverage reports for each library that makes up this system.

Chapter 2

Literature Survey

2.1 Introduction

The purpose of this chapter is to review existing documentation relevant to access control and authorization mechanisms, the security thereof and supporting technologies. The goal of this research is to create a modern attribute-based access control and authorization solution that will allow resource authorities to define complex rules — as they would occur in the natural world — in order to securely share resources and simultaneously protect them from non-authorized parties.

This document will define and discuss core concepts of security such as authentication, authorization and access control and make distinctions between them, discuss the importance of supporting systems to the core security systems mentioned such as logging and auditing and discuss existing access control models before finally discussing attribute-based access control.

2.2 Authentication and Authorization

Authentication and authorization are often confused or used interchangeably by people with limited understanding of the security discipline. Owing to the nature of this research, it is appropriate to make a clear distinction between the two.

2.2.1 Authentication

According to Matyáš & Říha (2002), authentication is the process of verifying the identity of a particular user. Most common authentication mechanisms are based on one of the following:

- Something you know: examples of this include: personal identification numbers (PIN), passwords and passphrases.
- Something you have: examples of this include: keys (digital or physical), radio frequency identification (RFID) tokens — or the more recent subset thereof, near field communication (NFC) — and identification documents such as a passport. More recently, this has been extended to include one-time passwords usually through time-based (M'Raihi *et al.*, 2011) methods or on-demand SMS to the user's cellphone (Aloul *et al.*, 2009).
- Something you are: this refers exclusively to the field of biometric identification which involves identifying a user based on unique features of the user's body, such as: fingerprints, unique facial feature recognition, iris scanning and voice print identification (Ratha *et al.*, 2001, Matyáš & Říha, 2002).

Each of the above methods has its own strengths and weaknesses. If only a single mechanism is implemented, the resulting system will always have weaknesses and therefore exploitable security flaws, regardless of which of the above authentication mechanisms are used. The generally accepted method of dealing with this is to implement more than one identification mechanism in separate categories, operating in conjunction with one another, in order to minimise the weakness of the overall system (Gunson *et al.*, 2011, Bhargav-Spantzel *et al.*, 2007). This is referred to as multi– or n– factor Authentication (MFA).

2.2.2 Authorization

Where authentication is about verifying an identity, authorization is defined as "conveyance of privilege from one entity that holds such privilege to another entity" by Farrell & Housley (2002). While this often gets confused with authentication it is also sometimes confused with access control. Authorization is the process one goes through to grant another user the relevant privileges (or authorizations) for the actions they need to be able

to perform. The process of checking a given identity against the privileges required to do a specific action is covered by access control.

It is worth emphasising that privileges are granted on a per action, per resource basis. It is mandatory in most applications to be able to distinguish between create, read, update and delete operations on any one resource as a result of enforcing good security principals such as separation of duties (Sandhu *et al.*, 1996) and least privilege (Sandhu & Samarati, 1994).

Within the context of this research; it is not important what mechanism is used to authenticate users; only that it is secure enough (using MFA) to render the authorization and access control methods (Sections 2.4 and 2.5) of this research worthwhile and secure to within a reasonable margin of error.

2.3 Auditing and Logging

Any security system is useless in isolation; by necessity, security systems need to be exposed in order to perform their function. Access control is no different and makes the entire setup more complex as a result of having many systems that make up the net functionality, all of which are exposed in some way (Kent & Souppaya, 2006).

In the recent past with some occurrences still in operation, organisations have relied completely on perimeter security, the process of having all traffic running through a single gateway where the traffic is scanned and security policy is applied (Richardson, 2008). A moment's reflection will reveal that in a given organisation, with multiple people, each with multiple devices and each of those with multiple connections; forcing all traffic through a single gateway is no longer feasible. According to Richardson (2008): "perimeter security is dead"; this is not to say that perimeter security has been rendered unnecessary, it is just no longer enough by itself.

Sandhu & Samarati (1994), Forte (2005, 2009), Casey (2008), Kent & Souppaya (2006) all suggest a number of additional systems as detailed in this section; all of which depend on logging.

2.3.1 Importance of Auditing and Logs

Logging, within a digital context, is the term given to the automated generation of event related messages by a particular system. According to Sandhu & Samarati (1994), access control is not a complete solution for securing a system; it must be coupled with auditing — which he defines as "a *posteriori* analysis of all requests and activities of users in the system" — which requires that all requests and activities be logged for immediate or later analysis. This is further corroborated by Forte (2009) who defines auditing as the process of identifying anomalies and Casey (2008) who defines a complete security solution for a system as follows:

1. **Setting Targets:** Determine security policy for resources.
2. **Enforcing Targets:** Limit access to the resource according to the security policy and the privileges of the subject through access control.
3. **Verifying Compliance:** Verify, through auditing, that the security policy is being executed correctly through the auditing process.

2.3.2 Requirements of Logs

According to Casey (2008), Forte (2005), in order for any set of logs to be used for any of the auditing processes described in this section, the log format must conform to a number of requirements or standards:

- Logs collected need to conform to whatever standard is necessary to be admissible in the courts of the jurisdiction where the system is operating in.
- Logs must be definitively proven to show any particular incident under question.
- Logs must show the activity of the complete incident.
- Logs must be definitively proven to have integrity and not have been tampered with.
- Logs and reports created from logs must be presented in such a way that it is readable in the courts of the jurisdiction the system operates.

In order to achieve some of these goals; Casey (2008) makes a number of suggestions:

- Use a unified timestamp format. This contributes towards showing the complete activity of an incident.
- Logs should be consolidated onto a single, separate system with rigid access control rules that disallows the deletion of records. This contributes to the integrity of the logs.
- Use a real time log correlation mechanism that reduces activity into sequences of events that can be compared to attack patterns. This contributes to the trustworthiness of the logs, makes them more human readable for use in court and reduces further processing during the auditing process.

Auditing cannot exist without first being able to reliably access admissible, authentic, complete, trusted and credible logs; however, it is ultimately up to each organisation as to what level they wish to protect their resources.

2.3.3 Automated Auditing

According to Casey (2008) the average organization generates 240M distinct log entries within a 24 hour period. It is for this reason that, in most cases, it is humanly impossible to manually audit logs; however, this does not mean that organisations should — or can afford to — do away with auditing. According to the *CSI Computer Crime and Security Survey* (Richardson, 2008), of more than 500 known breaches of security in 2008, more than half required little to no technical skill to accomplish; Richardson (2008) concludes that failure to monitor systems and transaction logs was a huge factor in the success of these attacks.

The answer to this problem is that logs should not be analysed by humans. Instead, automated auditing systems should be developed to process the data in real time and notify humans to potentially malicious activity which can then be dealt with. This in itself is also not without fault; the specifics of which and the different existing methods of auditing are discussed in this section.

2.3.4 Intrusion Detection

Intrusion detection, which is sometimes also known as incident detection, is a security auditing process that allows the administrators of a system or network of systems to

detect malicious activity (ideally) as soon as it starts (Kent & Souppaya, 2006). These systems are relatively new, are flawed by nature and are not guaranteed to detect all threats; according to Kent & Souppaya (2006), they are however capable of:

- **Incident Analysis:** Determine the sequence of events for client requests and server responses for a known incident.
- **Attacks:** Detect attacks against the system according to known patterns and signatures. Attempted brute-force, replay attacks would be good examples of what an intrusion detection system can easily detect.
- **Escalation of Privileges:** The system can trigger alarms based on the number of authentication and authorization requests by a particular address, successful or unsuccessful. The level at which the alarm triggers is usually configurable.
- **Inappropriate Release of Information:** based on recent successful authorization requests and the payload size of egress file transfers or emails, the system can trigger an alarm.
- **Operational Failure:** Operational failure can sometimes be symptomatic of a system being compromised. Alarms can be configured for this as well.

Intrusion detection systems are configured in such a way that they can monitor multiple services for incidents from a single system. This also allows intrusion detection systems to detect incidents that span across multiple services.

2.3.5 Forensic Analysis

Forensic analysis is a form of security auditing and involves applying forensic techniques to analysing digital artefacts such as system and transactional logs and file metadata for the purpose of constructing accurate and definitive timelines of incidents and related information in a trustworthy manner. Related information in this case is defined as anything that would result in the information gathered and the contents of the final report being admissible in court (Casey, 2008, Forte, 2005).

While forensic analysis is about creating documents that are legally admissible; it also has other uses that assist with incident detection. According to Richardson (2008), of all

known incidents of the responders to his survey in 2008, more than half were perpetrated by valid users within the respective organisations. Forensic analysis techniques can also be used to establish a baseline activity model; using this model, one can detect deviations from the model which can be investigated individually and in a targeted manner. Using these methods one can aid internal investigations and incident detection systems and — should they be needed — produce evidence and reports that are admissible and credible in court (Gorge, 2007).

2.3.6 Legal Liability

The methods discussed above can be described in a legally offensive manner; where the object is to collect data of attacks for the purpose of making a case to lay charges against perpetrators of attacks. However, the above methods can also be used in such a way that they can protect the organisation from possible indictments as a result of being compromised.

Liability in the context of digital resources and security is increasingly becoming a discussed topic in the legal sphere; due to the relative lack of legislation on the subject globally, including South Africa. Most of what exists is determined as a result of the precedent set by cases that have gone to trial over the last few years. According to Forte (2009) and the NIST standard for log management (Kent & Souppaya, 2006), the common thread in all of these kinds of legal cases is that in the event of an incident, the organisation must be able to establish due diligence in security matters or face legal liability for the damages as the result of the incident and negligence.

From the above discussion; one can only conclude that, as logging is a direct dependency of the auditing process and the lack thereof would make any security measure untrustworthy, logging is as necessary as any of the security measures described in this document.

Within the context of this research, the logs will be enough such that they match the requirements in Section 2.3.2; however for the purpose of research; the auditing process will be manual, as there will not be a large amount of data to process and security auditing is not the focus of this research.

2.4 Access Control

Access control is defined as the mechanism that restricts access to a protected resource, physical or otherwise, to authorized users only; and in doing so allows the sharing of the resource between legitimate, authorized users. Sandhu & Samarati (1994) state that access control does not exist in isolation and relies on other mechanisms to accomplish authorization, authentication and security auditing; as discussed in Sections 2.2 and 2.3. Access control requires these external systems to be accurate and secure in order to be able to give accurate responses to authorization requests; conversely it can be said that if the supporting systems are not secure or accurate to within a reasonable margin of error, then the responses given to an authorization request will not be accurate.

Figure 2.1: Ten Domains of Security of ISO 17799 (Saint-Germain, 2005)

Within the conglomerate of interlinking systems that support access control in an organisation (Figure 2.1), it is important to note this distinction as the large number of systems in place provide a wide attack surface to malicious actors, both internal and external to the organisation. A graph of the dependencies between these systems would reveal a meshed dependency rather than a hierarchical one; if any one of the systems that support access control are compromised then the security of the entire network of systems can no longer be trusted (Matyáš & Říha, 2002). This places emphasis on security as a whole and has driven the creation of organisational security standards such as ISO 17799 (Ma & Pearson, 2005) to try and address the problem.

2.4.1 Roles of Access Control

Access control defines three roles of particular importance: the subject, the resource owner and the authority. The subject is the user that is requesting access to the resource. The resource owner is the owner of a particular resource for which users (subjects) may request access. In access control models that stem from the discretionary model (described in Section 2.4.2) the resource owner and authority use cases are usually executed by the same person. The authority role allows the user to grant privileges to other users. The distinction between resource owner and authority becomes apparent when examining access control models that originate from the mandatory model (Section 2.4.2) where security is sensitive enough that the owner of a resource does not necessarily determine who has access to it.

It is worth noting that the roles defined above are not necessarily global. Access control only works when used in conjunction with a policy of least privilege (Sandhu & Samarati, 1994). Smaller organisations that may only have a single system administrator may grant the administrator root privileges; although this practice is typically frowned upon as it creates a single point of failure.

2.4.2 Root Access Control Models

Before examining existing access control models, Yuan & Tong (2005) suggest that all existing models can be categorised as either one, or a combination of two forms: discretionary and mandatory.

Discretionary Access Control

Discretionary access control (DAC) is a means of controlling access to resources based on the user's identity and authorizations on a per user basis, or for a group of users (Sandhu & Samarati, 1994). Authorizations are granted to users by a user that has the relevant permission to do so; usually the resource owner (Sandhu *et al.*, 2000, Yuan & Tong, 2005). DAC is a common form of access control, an example of which would be OAuth 2.0 where the resource owner can give a third party an access token, which can be used to gain access to the resource.

Mandatory Access Control

Mandatory access control (MAC) is most commonly used in environments where a resource owner may not make authorization decisions (Yuan & Tong, 2005); instead access is evaluated based on static security labels (for example: a clearance level) where a person may not use a particular resource labeled "X" unless they have the label "Y" which would be greater than or equal to "X". In the simplest cases the label is an element of a hierarchical ordered set where the higher label is said to dominate itself and all others below it (Sandhu & Samarati, 1994). This is typically used in government or military related environments where secrecy within and between members of an organisation matters. Table 2.1 shows the mapping between an example set of security labels and the permission they assign for a given action.

Table 2.1: Action Matrix for MAC

	Label			
Permission	Top Secret (TS)	Secret (S)	Confidential (C)	Unclassified (U)
Top Secret (TS)	Allow			
Secret (S)	Allow	Allow		
Confidential (C)	Allow	Allow	Allow	
Unclassified (U)	Allow	Allow	Allow	Allow

2.4.3 Derivative Access Control Models

The access control models discussed in this section represent a small portion of the available models due to the fact that many of the models that may have been included are either no longer widely used in favour of other models, were never widely used or are used for specific tasks. A single model matching the latter description has been included (lattice-based access control) as the more common models (identity-based access control and role-based access control) are discretionary.

Identity-based Access Control (IBAC)

Identity-based access control is the process of granting access to a resource to a subject based on the subject's unique identifier, such as a user name. This model usually involves creating an "Access Control List" (ACL) for each and every resource and individually adding authorized identities to the list of each resource the identity is allowed to access.

While IBAC is easy to implement, it is clear that it is only suitable for small user populations. As more users request access to the resource, the number of identifiers for the resource will grow and become difficult to maintain, making this approach unsuitable for large scale use (Yuan & Tong, 2005).

Role-based Access Control

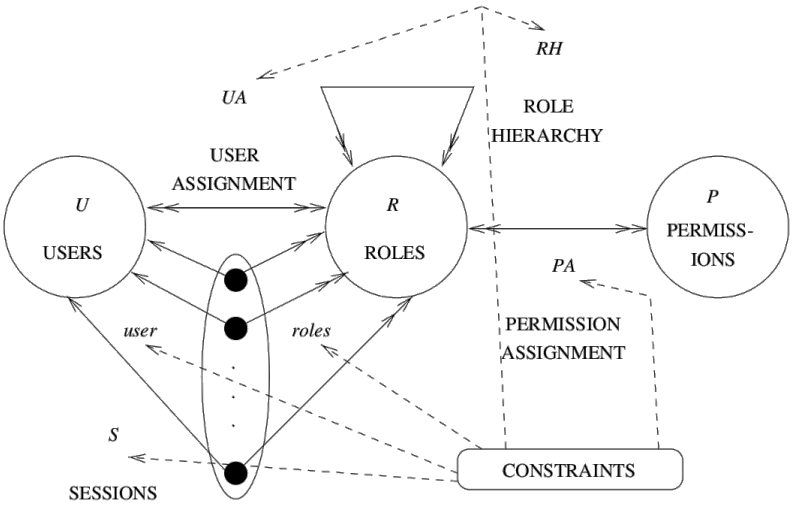


Figure 2.2: RBAC Model (Sandhu *et al.*, 1996)

Role-based access control (RBAC) does not have a generally accepted meaning as it is used in many different ways. For the purpose of this document it will be accepted that RBAC will be defined as using roles as an abstraction for the management of privileges. Flat RBAC, the simplest of the RBAC models defined by Sandhu *et al.* (2000, 1996), involves a many-to-many mapping between privileges and roles and another between roles and users; as can be seen in Figure 2.2. This form of access control provides a powerful mechanism for reducing complexity, cost and likelihood of human error due to the reduced administrator overhead and increased scalability as a result.

Lattice-based Access Control

Lattice-based access control (LBAC) was proposed for the United States' military and government use in the 1970's in order to address the lack of a MAC solution. LBAC combines an ordered set of security labels and a set of categories to form a "lattice"

(Yuan & Tong, 2005), which allowed a greater number of security classes using a smaller number of security labels.

Table 2.2: Lattice Structure for LBAC

	Label			
Category	TS	S	C	U
X	{ TS, {X} }	{ S, {X} }	{ C, {X} }	{ U, {X} }
Y	{ TS, {Y} }	{ S, {Y} }	{ C, {Y} }	{ U, {Y} }
XY	{ TS, {X, Y} }	{ S, {X, Y} }	{ C, {X, Y} }	{ U, {X, Y} }
None	{ TS, {} }	{ S, {} }	{ C, {} }	Public

Within the lattice structure, in order to be able to access a particular resource one needs to meet the requirements for both the category and security label. This allows for situations where a user may have "top-secret" clearance for a particular category but only "classified" clearance for another. If a resource is marked as being a part of a category but is otherwise unrestricted, the user must have a clearance level for that category in order to access the resource (Sandhu, 1996). Table 2.2 shows the combination of security labels and categories required to perform an action on a resource for each combination.

While LBAC tends to scale better than MAC or IBAC, it is still only worth using in security scenarios where flexibility would be detrimental; as in a military, government or any scenario where the organisation cannot afford to have resources accessed by those who don't "need to know".

The above access control models were each created with a specific purpose in mind; MAC and its derivatives were created with the military context in mind and DAC was created to model how privilege has worked in society: those who own a resource give permission to others to use it. DAC and MAC are difficult to combine and none of the above models have successfully taken the strengths and mitigated the weaknesses of both; however in some cases that do attempt it, the models start at either DAC or MAC and try to incorporate the other in some way. Jin *et al.* (2012) believes that the only way to get the functionality of DAC and MAC in the same access control system is to build from a model that is-based on neither DAC or MAC: attribute-based access control.

2.5 Attribute-based Access Control

Attribute-based access control (ABAC) could be easily described as the root access control model; the reason being that one can implement any combination of other access control models and paradigms and have them operate simultaneously (Hu *et al.*, 2014). In addition, ABAC can be easily used to manage access control in a federated environment due to the fact that the PDP can be configured to request attributes from many different authorities. The ABAC model is sufficiently flexible to be suitable for most real world access restriction problems; however, this comes at the cost of a higher complexity specification and stricter maintenance of security policy in order to be considered secure (Priebe *et al.*, 2007).

2.5.1 The NATO Problem

The North Atlantic Treaty Organisation (NATO) has a unique problem where the military forces of NATO are made up of regiments from members of the treaty who are rotated frequently. Due to the sensitive nature of operations and missions carried out by NATO, it does not make sense for NATO to maintain their own copies of the privileges of those who are assigned to that post; should one of these privileges be removed by the authority of that identity and the change is not reflected on the records kept by NATO, they would be in a situation where they could be leaking confidential information to those who do not need to know. As a result, NATO delegates authentication, authorization and access control to its members who control the resources.

Armando *et al.* (2013) describes the problem arising out of a need to securely share information between members of NATO, however the privileges of the operatives are governed by their parent organisation and are therefore not valid for another organisations resources. Duplicate privilege records result in a security risk (as described above) and differences in subjective interpretation of security labels (Section 2.4.2) result in either under- or over- protection of resources, both of which are detrimental to NATO missions.

The solution to the problem was the creation of new attribute-based access control infrastructure that uses content metadata to enforce content-based protection and release policy (CPR) when deciding about the release of information. These policies attempt to overcome the limitations of existing systems by administering ABAC where attributes referenced in the policy can come from any trusted authority. Doing so will decrease the

area where human error can occur and allow near immediate effect upon changing an operatives privileges (Armando *et al.*, 2013).

2.5.2 Model

Although literature about attribute-based access control has been present since at least 2000, there is still no authoritative answer to the question to what ABAC is (Jin *et al.*, 2012). ABAC finds itself in a similar situation to RBAC pre-1992 where the ideas existed for decades prior but were only first formalised by Ferraiolo *et al.* (1995), and later built on by Sandhu *et al.* (1996). To date, the most authoritative responses as to what ABAC is come from the NIST guide to ABAC (Hu *et al.*, 2014) and the specification from which it stems: the XACML core specification (Rissanen, 2013). Both of these documents aim to describe how the respective authors think that ABAC should work in terms of differences between it and other access control mechanisms.

Hu *et al.* (2014) states that ABAC enforces policies that are made up of complex boolean rules that can evaluate any number of different attributes. The flexibility and scalability of ABAC is attributed to the lack of privileges in the model; thus if access requirements for a resource change, the only thing that needs to be updated is the policy for that resource.

Hu *et al.* (2014) provides example attributes for ABAC in terms of RBAC and IBAC, where the subject's role or identity respectively are the "attributes" that determine if they may access a resource. Valid attributes can be anything, including but not limited to: the attributes of the subject; the attributes of the resource; any attributes of the environment; a set of policies that specify the use of the aforementioned attributes and any of the above from another internal or external trusted source. Priebe *et al.* (2007) illustrates the relationship between attributes and policy in Figure 2.3.

2.5.3 XACML

The eXtensible access control markup language (XACML) 3.0 is the latest iteration of the XACML core specification by Rissanen (2013) and the most widely accepted framework that is consistent with ABAC (Jin, 2014, Yuan & Tong, 2005, Mohan & Blough, 2010, Priebe *et al.*, 2007, Armando *et al.*, 2013). XACML is based on XML and its model employs elements such as rules, policies, rule- and policy- combination algorithms, attributes of subjects, objects and the environment, obligations and advice (Hu *et al.*, 2014).

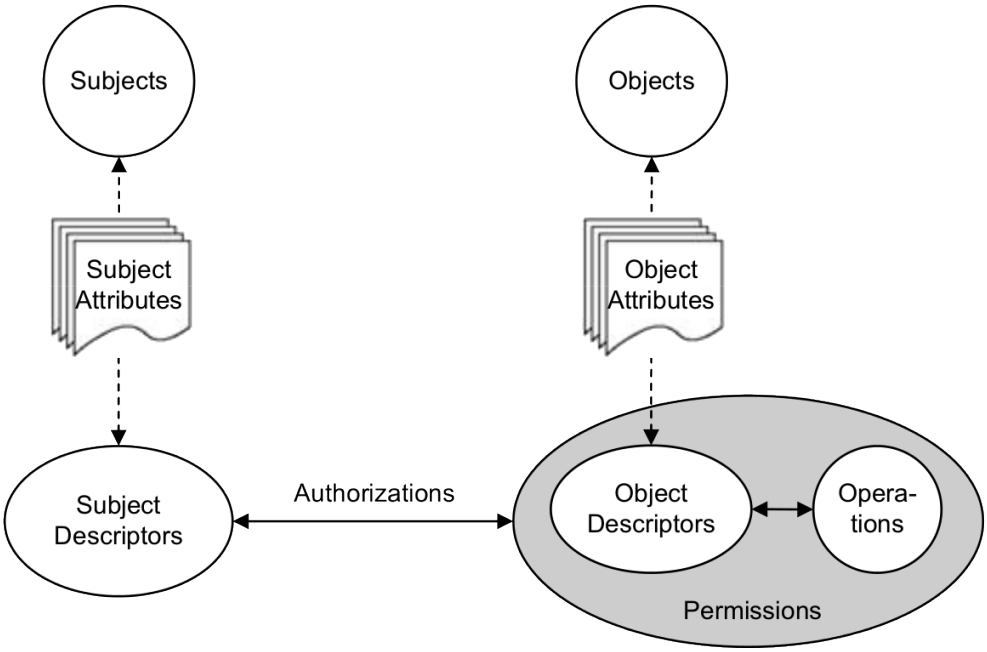


Figure 2.3: ABAC Model Overview (Priebe *et al.*, 2007)

2.5.4 Authorization

Authorization in ABAC differs from existing models such as RBAC and IBAC. Privileges have been traditionally assigned on a per role or per user basis in the case of RBAC and IBAC respectively. In the ABAC model, privileges are not explicitly granted but rather they are inferred from a user's attributes. System and security administrators instead control access to resources by defining and administering rules. Listing 2.1 shows an example of how rules are written in XACML; these rules can be written at a fine level of control and modeled on live data which results in the ability to implement any model at the policy level and not have to change any of the code. Concepts such as DAC, MAC, RBAC, IBAC, LBAC, blacklisting and whitelisting come naturally to ABAC without any of the overhead of having to manage ACLs, role hierarchies or any need to adapt the system to the new requirements in code.

2.5.5 Architecture

Within attribute-based access control systems there are a number of distinct subsystems that have their own tasks and responsibilities as shown in Figure 2.5. Within small and medium organisations, these subsystems are usually hosted on the same device; however

Listing 2.1: Example XACML Rule (Rissanen, 2013)

```
<!-- HEADERS -->
<Target>
  <AnyOf>
    <AllOf>
      <Match
        MatchId="urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match">
      <AttributeValue
        DataType="http://www.w3.org/2001/XMLSchema#string">
        med.example.com</AttributeValue>
      <AttributeDesignator
        MustBePresent="false"
        Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
        AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
        DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name"/>
      </Match>
    </AllOf>
  </AnyOf>
</Target>
<!-- FOOTERS -->
```

when an organisation outgrows the capabilities of a single device to manage access control, they can be moved to their own dedicated hardware and load balanced. The logical divide between each of these subsystems are detailed in Figure 2.4 and by Rissanen (2013) in the XACML core specification.

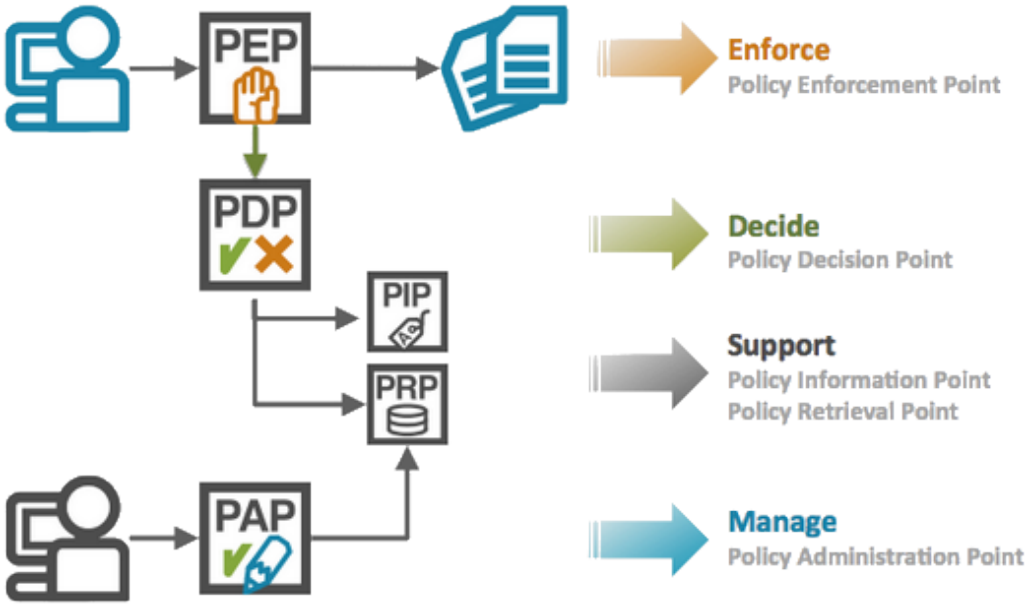


Figure 2.4: XACML Access Control Architecture (Nair, 2013)

PEP The policy enforcement point (PEP) refers to the software that accepts input from a subject (credentials or token) and formulates an authorization request which is sent to the policy decision point (PDP). The PEP is also responsible for receiving the response of the authorization request and allowing or denying the subject access to the resource.

Listing 2.2: Example XACML Request Context (Rissanen, 2013)

```
<?xml version="1.0" encoding="UTF-8"?>
<Request xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  xmlns:xsi="http://www.w3.org/2001 XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17 http://docs.oasis-open.org/xacml/3.0/xacml-core-v3
    -schema-wd-17.xsd"
  ReturnPolicyIdList="false">

  <Attributes Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject">
    <Attribute IncludeInResult="false" AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id">
      <AttributeValue DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name">
      bs@simpsons.com</AttributeValue>
    </Attribute>
  </Attributes>

  <Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource">
    <Attribute IncludeInResult="false" AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
        file://example/med/record/patient/BartSimpson
      </AttributeValue>
    </Attribute>
  </Attributes>

  <Attributes Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action">
    <Attribute IncludeInResult="false" AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
      read</AttributeValue>
    </Attribute>
  </Attributes>

</Request>
```

PDP The policy decision point accepts an authorization request and makes a decision as to allow or deny the request of the subject. In order to make a decision, the PDP first requires information about the subject requesting access. Figure 2.5 shows the overall process the PDP goes through to determine a user's authorization.

1. The subject needs to be identified by an external authentication mechanism.
2. The PDP fetches relevant information on the subject, object and environment required to make a decision from the policy information point (PIP). The information is used to construct a request context document, an example of which is visible in Listing 2.2.
3. The PDP fetches the relevant policies that apply to the resource in question from the policy retrieval point (PRP). The policy contains a number of rules that need to be evaluated against the request context. An example of one of these rules can be found in Listing 2.1
4. The PDP evaluates the rules defined by the policy and if the subject has sufficient authorization to meet what is required by the policy then a positive response is returned to the PEP.
5. Whether or not access to the resource was granted; the request and all details pertaining to it should be logged by a separate subsystem and kept for auditing.

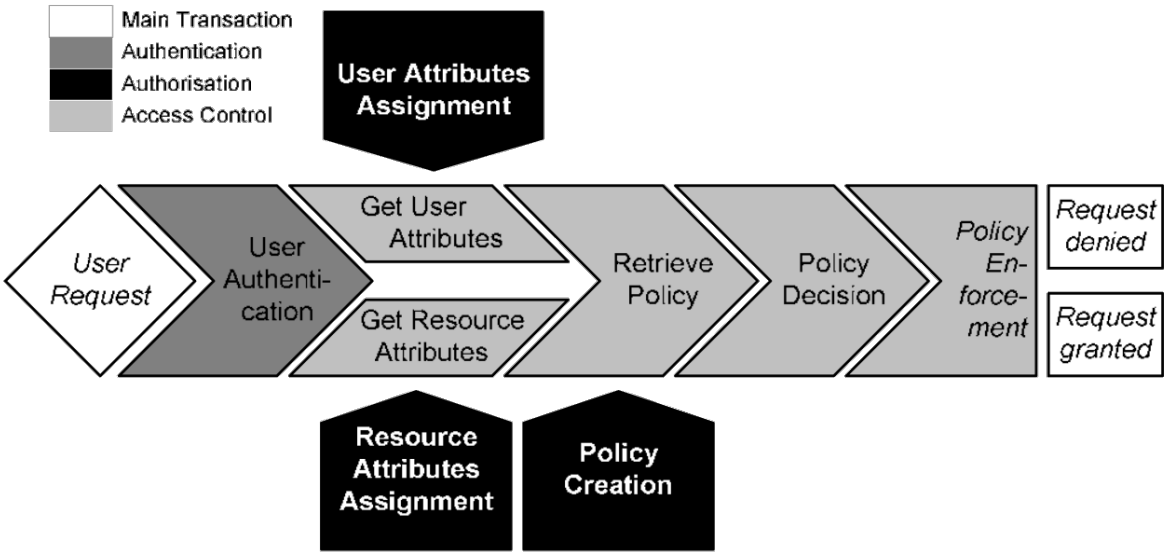


Figure 2.5: ABAC Service Chain (Priebe *et al.*, 2007)

PIP The policy information point (PIP) provides the attributes required to determine if a subject has sufficient privilege to match a particular policy

PRP The policy retrieval point (PRP) provides a mapping between resources and the policies that protect them.

PAP The policy administration point (PAP) is where the system administrator applies policies to protect resources. What is important to note with the PAP is that the process for accessing the PAP is circular in nature; to access the PAP the administrator must first go through the access control process via the PEP that is built into the PAP. This further enforces the point that all of the subsystems involved in access control are interlinked and if one is compromised, the trust of the entire network of systems is assumed to be compromised.

2.6 Modern Web Technologies

The following technologies are intended to provide key functions for use in the creation of a simpler attribute-based access control system.

2.6.1 JSON vs. XML

JSON provides a number of key features compared to XML that this research would benefit from taking advantage of:

- According to Nurseitov *et al.* (2009), a comparison of deserializing equivalent representations of the same data in JSON and XML show that JSON is approximately 57 times faster than XML, on average. The same study revealed that this comes at the cost of a higher CPU usage in comparison as a result of asynchronous multi-core processing in the deserialization procedure. This is possible due to the simplicity of the structure of JSON.
- According to Hameseder *et al.* (2011), a comparison of the encoded size of equivalent representations of the same data in JSON and XML show that JSON is approximately 57% of the size of XML, on average. The size difference between the two translates into an exponential reduction in bandwidth usage as a result as the size of the object being transmitted grows.

The functional difference that allows these benefits in JSON where XML does not is in JSON's lack of a static typing mechanism. JSON supports five types of data: numbers, booleans, strings, objects and arrays (which are just a subset of objects with a specialised serial representation). XML/RDF supports the ability to define any "type" that is required for use in a particular domain, which comes at the cost of a more verbose syntax.

Legibility

Although comparisons of JSON and XML are largely subjective due to the difficulty of objectively evaluating opinion, many posit — the author included — that JSON is better than XML in this regard. Nottingham (2012) attributes the argument between the two to a fundamental difference in their underlying meta-models. JSON and XML are based on common programming language data structures and the "legendarily complex" Infoset respectively. He continues, stating that one could — with great difficulty — create a new meta-model and map it to JSON and XML, however it is still unlikely to "feel native" to either and that the solution to the problem is to choose one or implement both separately; these solutions result in having made no progress on the issue, or introducing additional problems respectively.

On the topic of providing a solution, Nottingham (2012) suggests the following:

- Use JSON for Web data formats.
- Don't produce or consume XML for data.
- Provide excellent "client bindings" for statically typed languages as required.

JSON provides an alternative serialized data representation format that is able to replace the functionality provided by XML/RDF in XACML. JSON and Javascript are the *lingua franca* of the modern web and is the most viable alternative for XML/RDF in this research.

2.6.2 JSON Schema

JSON Schema is a standard for applying a schema to schemaless JSON data (Galiegue & Zyp, 2013). JSON Schema provides much of the functionality to JSON that RDF does to XML. JSON Schema introduces complex types and validation into JSON's meta-model as well as remote schema referencing support similar to that provided by XML/RDF through supporting standards such as JSON Pointer and Patch (Bryan & Zyp, 2011, Bryan & Nottingham, 2013).

The key benefit to using JSON and JSON Schema over XML/RDF is that there is no detriment. The two can be used interchangeably to provide the same functionality to the two different meta-models in the way that "feels native" to each model. This reduces the choice between the two to one based on subjective opinion.

2.6.3 JSON Web Token

JSON Web Token (JWT) is a JSON format for encoding claims data for transmission in space constrained environments such as HTTP headers and URI query parameters (Jones *et al.*, 2012). JWTs transmit encoded data such that they can be signed using the JSON Web Signatures (JWS) standard (Jones *et al.*, 2011) and encrypted using the JSON Web Encryption (JWE) standard (Jones & Hildebrand, 2015). The signing and encryption processes make use of JSON Web Keys (JWK) standard (Jones, 2015b) which are a JSON data structure that represents cryptographic keys. JWS, JWE and JWK rely on the JSON Web Algorithms (JWA) standard (Jones, 2015a) to specify algorithms and their unique identifiers. All five of these standards are combined and administered by the JSON Object Signing and Encryption (JOSE) work group in the IETF.

2.6.4 Attribute-based Access Control and Identity

ABAC relies on attributes in order to grant access to users attempting to access resources. Identity providers are dedicated systems for managing and providing attributes relating to subjects or users. In order to be able to administer ABAC, an identity provider is mandatory as the functionality provided by identity providers that allows ABAC administrators the ease of having a single point where attributes are managed, is the same functionality that allows that information to be used universally.

OpenID Connect

OpenID Connect (OIDC) is a standard for a federating, single sign-on (SSO) identity provider that makes use of JSON and the JOSE standards (Sakimura *et al.*, 2014). OIDC is a simple identity layer on top of the OAuth2 DAC protocol (Hardt, 2012) that enables clients or relying parties to verify the identity of the end-user based on authentication performed by the identity provider, as well as access basic profile information about the end-user. The system within ABAC that communicates with the OIDC identity provider would take the role of the relying party in the identity transaction.

The choice to use JSON as the base for all development and research to come promotes the use of these web technologies — that are designed and built with the same meta-model and paradigm in mind — the most reasonable choice to provide the required functionality for this research.

2.7 Chapter Summary

It is clear that in order to implement real-world access control requirements, one must use attribute-based access control. RBAC, IBAC and LBAC have all been designed with a set purpose in mind and, as a result, are not effective outside of their intended purpose. ABAC on the other hand, is able to express all of these models — and more — simultaneously with its ability to accept arbitrarily complex rules in a dynamic and contextual manner and will do so with negligible additional administrative overhead due to the decoupling of policy from enforcement.

XACML has proven itself to contribute a suitable model that can be used to implement ABAC and it will provide a starting point for a new ABAC standard. Areas for improvement in the new standard include but are not limited to:

- XACML (the language excluding supporting architecture) was designed using XML and RDF/XML (Klyne & Carroll, 2006). These components are still part of the ongoing development of specifications for the "semantic web", however they are both needlessly verbose, restrictive and not easily legible (even to developers). There is room for improvement in using more recent technologies such as JSON and related technologies such as JSON Schema (Galiegue & Zyp, 2013), Pointers (Bryan & Zyp, 2011) and Patch (Bryan & Nottingham, 2013) to aid management of policies and rules and solve the aforementioned problems with cleaner syntax.

- The XACML architecture is explicit in creating a logical divide between the components; however, existing implementations of XACML implement each of the services separately creating a physical divide instead of a logical one. Most use cases of an ABAC engine would not require remotely hosting each service separately as doing so would be too costly. There is room for improvement in allowing the user the choice of running all subsystems on a single machine and communicating through faster means such as RAM; this would provide a significant improvement over co-hosting the separate services and communicating through the loopback connection.

- As mentioned in Section 2.5.5, the cost of the flexibility gained from ABAC is the increased complexity of the overall system. With particular reference to XACML, this creates a barrier to entry for most small to medium organisations that cannot afford to pay specialists fees to manage and maintain access control on their behalf. There is room for improvement in that one can — in conjunction with the aforementioned points — decrease this barrier to entry by simplifying the architecture and configuration and automating the build process for the system while still maintaining a manual process for advanced users.

- As described in Section 2.2, access control does not exist in isolation; in order to satisfy the requirements of access control: OpenID Connect (Sakimura *et al.*, 2014) and OAuth2 (Hardt, 2012) with multi-factor authentication (Aloul *et al.*, 2009) will be used as authentication mechanisms and a simple logging system which will fulfill the requirements specified in Section 2.3.2. These components have little to do with the access control process; however, using authentication as an example, XACML was not designed to only interface with the security assertion markup language (SAML) (Hallam-Baker, 2001), however it was designed with it in mind. There is room for improvement in providing a standard interface for authentication, log storage and formatting that can be used to switch these mechanisms at will by the system administrator.

From the discussion in this chapter, there is sufficient evidence to suggest that there are many who would benefit from a simplified ABAC solution for small deployment.

The following chapter describes the design and considerations that will be followed to build a functional proof of concept according to the goals specified in Chapter 1.

Chapter 3

Design

3.1 Introduction

This chapter provides a proposed working solution to the problem detailed in Chapter 2. The aim of this research is to design and implement a proof of concept system for enacting attribute-based access control for use in modern TCP/IP enabled systems. The proof of concept should improve on existing solutions in the areas of simplicity and ease of use while maintaining an equivalent or greater standard of security. In order to be considered attribute-based access control, the proof of concept must also satisfy the requirement of functionally complete rule syntax in order to be combined using arbitrarily complex boolean logic. Therefore the improvements to be made will be focused on:

1. Simplifying the rule syntax
2. Generalising the system architecture to decrease repeated functionality.

These are both necessary to simplify and improve the ease of use of the system when compared to its predecessor. Other considerations for the system to further facilitate its improved ease of use include:

- Automated deployment.
- Default configuration for use in small deployments, while providing options for large deployment.

3.2 Criteria for Success

The system will need to enact functionally complete attribute-based access control for a single resource in order to be considered a success. The following requirements will be used to guide the design process of the proposed system:

3.2.1 Maintain the existing standard

This research seeks to improve on certain aspects of the existing system, namely XACML. Aside from the improvements made, the core functionality and security of XACML should be maintained. The improvements done should also not detract from the functionality provided by XACML, with particular reference to the access control rules and architecture. The rule syntax has to be functionally complete and the architecture needs to provide for all of the subsystems detailed in the XACML specification. This is the largest assessment point as this research aims to provide similar functionality. This will be evaluated by qualitatively comparing the new system to XACML.

3.2.2 Maintain security requirements

The considerations regarding security of this research are relatively small compared to the wider topic. Any security concerns not addressed here, such as human error and non-software related security concerns, are considered to be out of the scope of this research. The design of the system should take into account the following considerations:

1. All communication over TCP/IP should be appropriately encrypted.
2. The system should verify, to an acceptable level, the identity of all parties involved in the authorization transaction.
3. The system should verify, to an acceptable level, the true origin and destination of all communication in the authorization transaction.
4. The system should sanitize all communication it receives during the authorization transaction.
5. The system should provide admissible, authentic, complete, trusted and credible logs of all attempted authorization transactions.

6. All of the above should be tested to an acceptable level.

With these in place, the resulting system should be well tested, able to be audited and therefore considered to have established due diligence.

3.2.3 Improved Ease of Use

With particular reference to the rule syntax and architecture of the proposed system, the aim of the research is to simplify and improve the ease of use compared to the existing system as seen in Section 2.5.3. The goal of this is to make the system usable in small deployments without requiring a disproportionate amount of hardware or administrative skill. The indicator that will be used for this success criterion is a single resource deployment of the system that passes all of its integration tests.

3.3 Javascript, Node JS and JSON

Javascript will be the language of implementation of the proposed system due to personal familiarity and ubiquity across browsers, servers through Node JS and data representation in the form of JSON. Javascript on the server-side through Node JS provides a middle ground between the overhead cost of using an interpreted language and ease of development, testing and code legibility. Efficiency lost can be minimised through the combination of these specific technologies as there is no cost of conversion between them and the exchange is handled natively.

3.4 Modular Design

Utilizing a modular design lends ease of maintenance to the proposed system. By restricting the scope of a component in a larger system, one is able to better test, extend and replace components if the need arises.

3.4.1 Testing

Due to the evaluation of the research depending largely on its ability to be tested and the fault intolerant nature of security applications, it is most efficient to write the application in a way that minimizes the overhead of testing. Design practices from test-driven development and the object-oriented paradigm can be used to achieve this. Developer test-driven development and design (DTDD) provide a method of programming that reduces programming error in the final product and allows for such errors — as a result of subsequent work — to be quickly and easily detectable. When this is combined with object-oriented programming the resulting classes should be written in such a way that they can be tested in both isolation and as a system.

In order to further facilitate ease of testing, the aspect of purity from the functional programming paradigm can be adapted for use in an object-oriented environment. While purity and the object-oriented paradigm are mutually exclusive, practices as a result of purity can be used in object-oriented code to make it easier to test. In particular, while there will be memory or other side-effects as a result of member methods, keeping the purpose of a particular method singular adds to the maintainability and ease of testing the code.

Using these principles will result in code that is less wasteful of resources, more correct and has the additional benefit of being able to tell immediately where and when improving, fixing or refactoring code has caused unexpected behaviour.

3.4.2 Extensibility

In addition to testing, the application needs to be extensible enough to accommodate the environment or domain that it is deployed in. Most of the responsibility for this aspect of the research lies with the identity provider and is therefore out the of scope of the research; however, this does need to be considered when dealing with attributes of the environment. The appropriate measures need to be taken to allow for users to include their own environment attributes for use in authorization.

3.4.3 Summary

The following design principles will be used in the implementation of the system:

1. As far as possible, the research will be separated into independent libraries that can be tested, maintained and replaced individually, if necessary.
2. The design will be primarily object-oriented. Functionality will be separated into classes which can be tested in isolation.
3. Classes and their static and member methods will be written to follow existing design patterns where possible.
4. Static and member methods will be written with a singular purpose in mind to aid ease of testing.
5. Tests will be written to cover as much of the program as possible.
6. The system will be designed to accommodate the needs of any single domain's needs regarding specific domain-related attributes.

3.5 Rule Syntax

Rules are what drive any attribute-based access control system. Rules are also the most frequent point of human contact and are therefore the most frequent source of error due to the flawed nature of humans. A simpler and more legible rule syntax for an attribute-based access control system can reduce the opportunity for human error by making any such errors more obvious.

As discussed in Section 3.2.1, while the rule syntax can be made simpler and more legible, it cannot be done at the expense of its functional completeness. There are a number of approaches to be considered:

3.5.1 New Language

It is possible to create a new language for the purpose of providing the functionality to write simple, easily legible, functionally complete attribute-based access control rules. This approach is the most complete among those detailed here, it allows a solution that maximises simplicity and legibility while maintaining functional completeness at the cost of being the most complex to implement.

3.5.2 Existing Language

Using an existing attribute-based access control language is possible, and provides the benefit of being established and well tested beforehand. There is also the possibility of existing libraries for such languages that can be used to process the language directives.

JSON Schema

JSON Schema is an existing standard that provides a mechanism for specifying an application-level schema for schema-less NoSQL databases. JSON Schema can combine sub-schemas using arbitrarily complex boolean logic and can be used to define itself. While access control is not the intended purpose of the standard, it has all of the functionality to perform the role of a rule language and is simpler and more legible than XACML. While it is simpler and more legible, JSON Schema is still complex and can contribute to malformed input as a result of human error. JSON Schema is a viable and easy solution to the problem as it aids progress in making attribute-based access control simpler and easier, however it is not the ideal solution to the problem.

XACML

The aim of this research is to simplify and improve ease of use of attribute-based access control compared to XACML, therefore using the XACML rule language is counter productive to the goal of the research as it would provide no net benefit.

3.5.3 Hybrid Approach

A new and existing language can be used together to create an ideal viable solution that is a midpoint between the relative complexity and ease of the above solutions. JSON Schema, as an intermediate representation of the rule, can be used to validate data conformance to a schema; or rather can be used to validate attribute conformance to a rule. A new language can be constructed that allows rules to be defined in a syntax that approximates natural language with a syntax similar to SQL. The compiler for such a language would render descriptions of a rule into a rule defined in the JSON Schema format, which can be used to validate attributes. The benefits of this approach include more segmentation of code resulting in it being more testable and easier to maintain and it provides

an ideal solution as described in Section 3.5.1 in a less complex way; albeit at the cost of adding an extra step with additional computation.

The computational cost of adding an extra step to the rule evaluation can be minimised in the early stages of evaluating the rule. When the rule is processed, the intermediate form of the rule can be cached and used multiple times without parsing the new rule.

3.6 Architecture

The architecture of the system can be split into three logical parts:

1. The **Authorization Engine** is responsible for the authorization decision to allow or deny access given a rule and a set of subject, object and environment attributes.
2. The **Store** is a generic API for the storage, generation and retrieval of data required in the authorization transaction. The store is generic in that the type of the data does not influence its handling.
3. The **Barrier** is responsible for providing the necessary infrastructure to interact with end-users, authenticating them, fetching the necessary information from the stores to pass to the authorization engine and enforcing access control decisions.

These parts will each be implemented as separate libraries for reasons discussed in Section 3.4. In addition, splitting the responsibilities of the overall system means that the system can be deployed on a single host or the various logical parts can be split across many according to the needs of the resource owner.

3.6.1 Barrier

The barrier is the main point of contact for the end user as it is both the first and final point of contact. Its largest responsibilities are facilitating the sequence of the authorization process and enforcing the access control decision at the end of an authorization transaction. Figure 3.1 demonstrates the process that the barrier must follow in order to authorize a subject's access request to a particular resource.

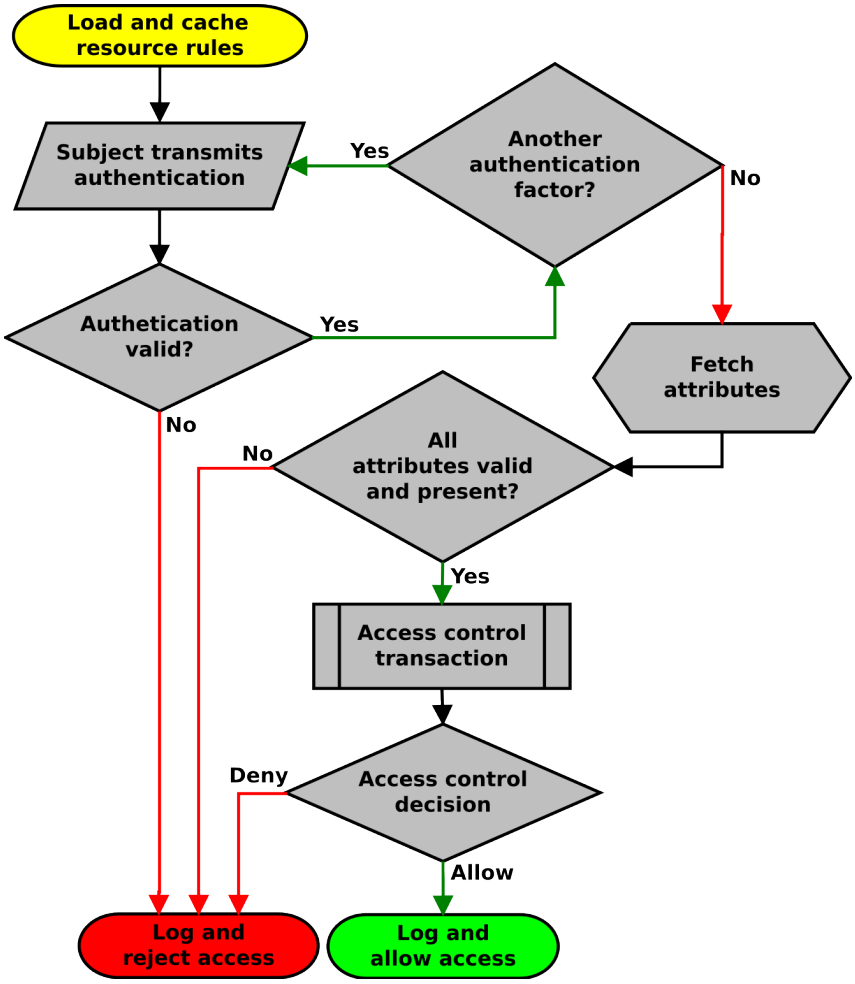


Figure 3.1: Barrier Authorization Transaction Flow Chart

Authentication

The barrier should provide the necessary framework to execute any desired authentication within a given domain. Likely options for this include: token based authentication such as RFID cards, time-based "one time pins", various biometric authentication solutions such as finger print and palm print readers and iris scanners and password authentication. The barrier should also be able to implement multiple of these authentication factors as required within a given domain.

Fetching Rules and Attributes

The barrier is responsible for initiating communication between the **Store** and **Authorization Engine** and maintaining the sequence of the authorization process. The barrier

must be able to fetch rules and attributes from local or remote stores, transform them into the necessary input format of subsequent systems and transmit them to their required destinations while maintaining the security of the data.

Access Control

While the **Authorization Engine** is responsible for the final access control process, the barrier is responsible for providing the data that is necessary for that process and enforcing its decision. The responsibility for assembling, transforming and validating the data that is provided to the authorization engine belongs to the barrier as this would result in a decreased load for the authorization engine. This is because a single barrier will always serve a single resource whereas an authorization engine may be serving multiple barriers depending on the setup within a given domain. The result of this is that the computational cost associated with preprocessing the data is distributed among the barriers within a given domain while maintaining a single source of truth for access control decisions.

Enforcement

Lastly, the barrier is responsible for enforcing the access control decision it receives from the **Authorization Engine**. In order to do this, the barrier must provide the necessary framework to interact with any software or hardware that is responsible for physically or digitally protecting the resource.

Logging

As part of maintaining due diligence, the barrier must log all authorization requests it receives from end users. As the central system for the authorization transaction, it makes the most sense that the barrier handles this responsibility. In order to fit into the structure of modern internet security, the barrier should provide the necessary framework to allow the destination of logs to be configurable. This is useful if the domain where the barrier is present uses a centralised log facility. Logs should also conform to the standard logging practices detailed in Section 2.3.

3.6.2 Authorization Engine

The authorization engine's primary responsibility is serving the **Barrier** by making access control decisions given a set of rules and subject, object and environment attributes and returning the decision for enforcement. In order to execute this task, the authorization engine must be able to parse rules passed to it and be able to evaluate the rules against the set of attributes. The authorization engine should be constructed in a way that it is able to be deployed both locally — alongside the barrier in the simplest case — or separately as a centralised system that services multiple barriers over the network. In order to meet this requirement, the authorization engine needs to be designed to operate in a stateless manner where it accepts input, processes it and produces a result.

3.6.3 Store

The store is an API for storing and accessing stored rules and attributes. The store is a simplification of the XACML equivalents which are broken up into multiple similar subsystems for each type of data that is being stored. This is a result of the store disregarding the types of the data in favour of a system whereby data is accessed using an identifier such as a name or JSON Pointer.

The store should allow for handlers of different data sources such as flat file and different database storage. The key difference between the store API and any traditional database is that the store must handle attributes that are subject to change. For example, it makes little sense to store the time in a database; instead the time should be fetched or generated when it is needed. Similarly, in the case of specialised hardware that collects data continuously, it makes little sense to store all of the data that it collects when only specific data is needed and can be fetched in real time. In order to compensate for this, the store must allow the functionality to define generator functions to fetch data when it is needed.

The store should also be able to be deployed separately from the barrier and be queried remotely. In this particular mode of operation, these stores can be used to provide the single point of truth for any given attribute.

3.7 Chapter Summary

This chapter provides insight into the function of the overall system with specific regards to its three components: the barrier, authorization engine and store. The success criteria of the overall system have been defined as well as the requirements and roles for each of the subsystems. In addition, key concerns regarding the manner in which to proceed for rule syntax, security and overall design have been addressed. The barrier received the most specification and analysis at this point as it is the key focus of the inversion of control experienced in the XACML specification and therefore has the widest surface area for potential attack.

The following chapter details the practical implementation and operation of the architecture defined in this chapter.

Chapter 4

Implementation

4.1 Introduction

This chapter explains the implementation of the proposed system and all completed features toward the design specified in Chapter 3. This chapter defines the development environment, core libraries used and details the implementation of each of the subsystems (as they are referred to in this chapter) that make up the final system. The system consists of three subsystems which are the **Authorization Engine**, **Store** and **Barrier**. Within each of these sections, where relevant, difficulties, changes in design to resolve them and limitations to the function of the individual subsystems are discussed and explained.

4.2 Development Environment

The evaluation of modern web technologies in Section 2.6 led to the following choices relating to the development environment for the system:

4.2.1 JSON

Section 2.6.1 conducted an investigation into reasons why JSON is a more suitable serialized object data notation than XML/RDF for this application. In addition to these reasons, standards such as JSON Schema and JOSE are designed with JSON in mind as their object data notation. JSON is also more human readable than XML/RDF, which

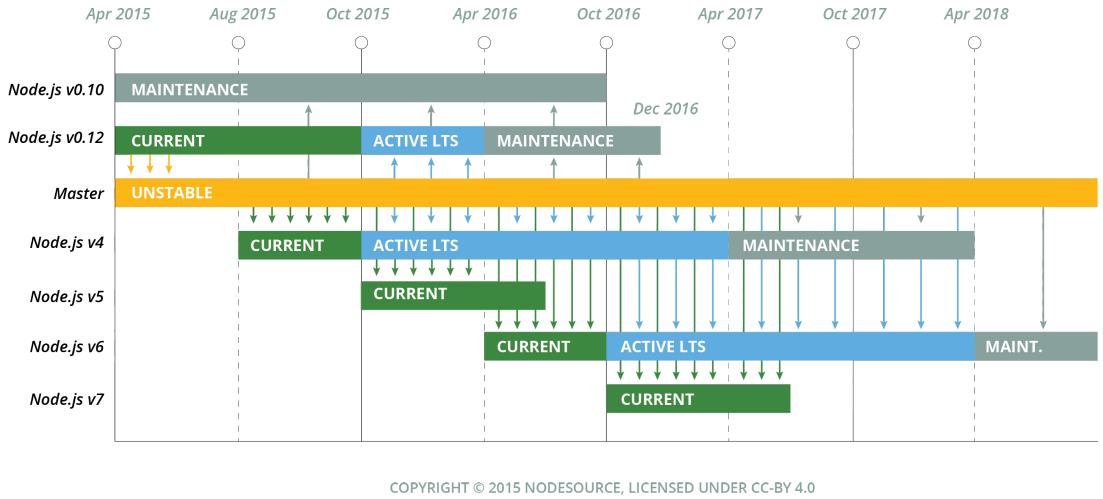


Figure 4.1: Node JS Long Term Service Release Schedule

contributes toward the goal described in Section 3.2.3. As JSON is a serialized object data notation, it requires support from the programming language to be deserialized and used — either natively or through the use of a library. JSON is ubiquitous enough that libraries exist for most languages to use JSON to the full extent of its scope.

4.2.2 Javascript (ES6)

As discussed in Section 3.3, ES6 has been selected as the programming language which the research will be implemented in. Javascript or ECMAScript 5 (ES5) is most often used in browsers as a client-side scripting language; however, it can be used to create server-side applications through Node JS. Both browsers and Node JS support javascript natively, however unlike most stable versions of browsers such as Google Chrome and Mozilla Firefox, the latest versions of Node JS support most of the functionality of ES6. This makes Javascript simpler, easier to maintain and easier to document when programming in the object-oriented paradigm.

4.2.3 Node JS 6.8.1

At the time of writing, v6.8.1 is the latest available release of Node JS. Common sense dictates that secure environments are less fault tolerant than others and therefore prefer

stability and predictability over additional features. To accommodate this need for stability, the Node Foundation maintains two development branches: Long Term Service (LTS) and Current, which are at v4.6.0 and v6.8.1 respectively. Node JS v4.6.0 supports 59% of ES6 features which is insufficient for this research (Kapke, 2016). The decision was made to use the latest version of Node JS — which supports 99% of ES6 features — instead of the LTS version, despite the loss of stability to maintain the support of ES6 for this research. Figure 4.1 shows the Node JS LTS release schedule, this indicates that in October 2016 there will be a new LTS release based on v6 of Node JS. Until then the current latest version will be used.

4.3 External Libraries

NPM, Inc. provides the Node Package Manager (NPM) service which is an index of all available libraries (or packages) for Node JS. All of the following libraries are available and were retrieved from NPM during the development process.

4.3.1 JSON Document

JSON Document is a "fast, declarative, standards-based Javascript (ES6) data modeling and manipulation" library (Smith *et al.*, 2016b). It implements standards such as JSON Schema, JSON Patch and JSON Pointer to provide a simple, yet powerful interface for creating self- initializing and validating models. JSON Document has been designed with the concept of interchangeable back-ends in mind and is also built with the test-driven development process. This library is useful to the proposed system in being able to specify a schema for configuration and store data, and being able to validate loaded data with little effort. Additionally, the JSON Schema component of this library provides all of the necessary functionality needed for it to be used in the **Authorization Engine** subsystem to evaluate rules against attributes.

Listing 4.1 shows an example of a simple JSON Schema validation program.

4.3.2 Testing Framework Libraries

For the purpose of unit and integration testing of this research, the decision was made to use Mocha (Holowaychuk, 2016), Chai (Luer, 2016), Sinon (Johansen, 2016) and

Listing 4.1: JSON Schema Validation Example with JSON Document

```
let schema = new JSONSchema({
  type: 'object',
  properties: {
    foo: { maxLength: 5 }
  }
})

schema.validate({ foo: 'too long' })
// => { valid: false, errors: [{...}] }
```

Sinon-Chai (Denicola, 2016). This decision was made based on personal familiarity, the maturity and wide usage of each of these libraries and their fulfillment of the requirements for testing. Each library will be discussed individually.

Mocha

Mocha provides an asynchronous testing framework to facilitate isolated unit and integration tests. Listing 4.2 shows an example of the Mocha testing framework in use with Chai's assert. Testing under Mocha is done in a hierarchical manner, allowing multiple levels of describe blocks to be used to describe the scope of the tests. Through the use of code hooks, Mocha allows for each level of describe block to define "before" and "beforeEach" functions that are executed, in order, before the testing occurs. This setup results in each test being executed within a "clean" environment that has not been modified by any previous test. This narrows the focus of the test which increases the accuracy of the test output.

In the event of a test failing, Mocha produces an error report for the failing test. The error reports are shown at the end of all of the test output and are numbered so they can be linked back to the tests that produced them. The error reports contain the reason for which the test failed as well as a stack trace, where applicable. Given that each test is executed in a "clean" environment, this aids in highlighting the source of the problems causing irregular behaviour and therefore improves the speed at which these problems can be solved.

For these reasons, Mocha is capable of fulfilling the role of testing framework in this system and will contribute — in combination with other libraries — to the legibility and maintainability of the code.

Listing 4.2: Unit Testing Example with Mocha and Chai

```
const assert = require('chai').assert
describe('Array', () => {
  describe('indexOf()', () => {
    it('should return -1 when the value is not present', () => {
      assert.equal(-1, [1,2,3].indexOf(4));
    })
  })
})
// => Array
// =>   indexOf()
// =>     ✓ should return -1 when the value is not present
// =>
// =>
// => 1 passing (9ms)
```

Chai

Chai is a testing assertion library that can be used with any Javascript testing framework. Chai provides three particular API's to assist in assertions for testing, namely: Should, Expect and Assert. These API's all provide similar functionality with subtle differences, although the main difference being the syntax they provide. The key benefit of using Chai as an assertion library is that it provides a way of testing that resembles natural language, as can be seen in Listing 4.3.

- **Assert** provides the most common syntax that reflects the assertion style found in other languages and assertion libraries for those languages. This particular API is not limited in what it can be used to test and has a natural language "sentence structure"; however, the sentence structure of this API is verb-subject-object which is foreign to English speakers.
- **Expect** provides a new syntax or style of assertion. Like the Assert API, this API is not limited in what it can be used to test; however, it provides a subject-verb-object (SVO) "sentence structure" that can be related to the English natural language. This relation and structure similar to English further improves the legibility and therefore the maintainability of the code.
- **Should** provides the simplest of the available syntaxes, however this comes at the cost of functionality. The Should API is only capable of dealing with static variables and is not able to evaluate functions. When this functionality is required, the Assert or Expect API's should be used instead. Like the Expect API, Should also uses a SVO "sentence structure". The difference between the two is that "should" is a direct property on the variable itself and therefore lacks the function call parentheses which cause a break in reading for the Expect and Assert API's.

Listing 4.3: Example of Chai Should, Expect and Assert

```
// Should
chai.should() // setup
foo.should.equal('bar')
tea.should.have.property('flavours').with.length(3)

// Expect
let expect = require('chai').expect // setup
expect(foo).to.equal('bar')
expect(baz).to.be.a(string) // baz is a function that is evaluated

// Assert
let assert = require('chai').assert // setup
assert.equal(foo, 'bar')
assert.property(tea, 'flavours')
```

The dual functions of Chai, in that it provides assertions and improves legibility through a natural language-like syntax, qualify it for use in this research.

Sinon

Sinon is a library that assists with testing asynchronous Javascript code. The library is useful for testing function callbacks and how functions or methods are used; this is done using the two main API's that are provided:

- **Spies** can be used to create anonymous functions or wrap existing functions and record arguments, return values, the value of *this* and exceptions thrown (if any) for every call made to the function that is being spied on. Any of the information recorded by a spy can be used in tests.
- **Stubs** have the same functionality as Sinon spies with two differences. Instead of wrapping existing functions or methods, stubs replace the function so as to prevent that code from being executed. Stubs that replace methods and functions can also simulate responses, such as throwing errors or returning a value after being called. Stubs can also be used to test behaviour in the code such as error handling and recovery and handling irregular input as a result of an error elsewhere in the code.

Listing 4.4 gives an example of the functionality of Sinon in the Mocha testing framework.

Sinon-Chai

Sinon-Chai is an extension of Chai that provides assertion functionality in the Chai syntax to the Sinon mocking framework. This library provides the convenience of not having to

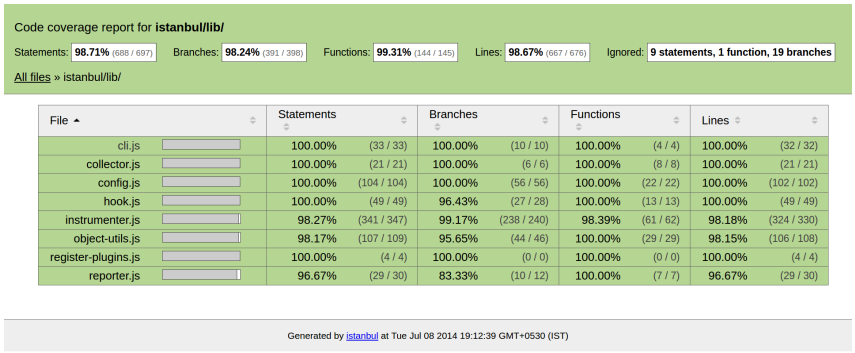
Listing 4.4: Example of Sinon Stubs and Spies

```
// pre-test setup
// Spy Example
it('should execute the callback when finished', () => {
  let callback = sinon.spy()
  publisher.send('message', callback)
  assertTrue(callback.called)
})

// pre-test setup
// Spy on existing method
let spy
beforeEach(() => {
  spy = sinon.spy(publisher, "subscribers")
})

afterEach(() => {
  publisher.subscribers.restore() // unwrap the spy
})

it('should get a list of subscribers to send the message to', () => {
  publisher.send('message') // send calls subscribers in the method body
  assertTrue(spy.called)
})

// pre-test setup
// Stub
it('should send messages to all subscribers despite errors', () => {
  let stub = sinon.stub().throws()
  let spy = sinon.spy()
  publisher.subscribe('my topic', stub)
  publisher.subscribe('my topic', spy)

  publisher.send('message')
  assertTrue(spy.called)
  assertTrue(stub.called)
  assertTrue(stub.calledBefore(spy))
})
```

use a different assertion style for Sinon that differs significantly from other tests that use Chai assertions.

4.3.3 Test Coverage with Istanbul

Istanbul is a utility for analysing the amount of code that has corresponding tests (Anantheswaran, 2016). The output of Istanbul is a coverage report that details the testing coverage of the code at the directory, file and line levels. Figure 4.2 shows an example of the HTML report generated by the code test coverage utility. The result of the analysis shows the test coverage of statements, branching logic, functions and lines at each level as a percentage of the total as well as which of those are not tested at the line level view. Istanbul presents two key benefits to this research: it aids TDD by highlighting untested code and the statistics generated from the coverage report can be used to demonstrate decreased programing error.



(a) Istanbul File Coverage Report (HTML)

(b) Istanbul Line Coverage Report (HTML)

Figure 4.2: Istanbul Example Coverage Report (Anantheswaran, 2016)

4.3.4 Logging with Bunyan

Bunyan was the library chosen to fulfill the logging requirements in this system (Mick, 2016). The reason for this is that Bunyan is the only mature logging framework for Node JS that accepts that logs are not meant to be read by humans — and by default logs in JSON, which is machine readable with the correct libraries.

Bunyan provides support for all of the features required for security logging detailed in Section 2.3.2 and so its correct usage satisfies part of the goals in Section 3.2.2. A mapping between the features of Bunyan and the requirements they satisfy is as follows:

- All logs default to showing exactly when and where the event that caused the log occurred. This is demonstrated by: the presence of the log name showing the context in which the event occurred; the presence of the hostname and process ID (pid) showing on which physical and logical device the event occurred; the log message showing the nature of the event and the presence of a timestamp showing

Listing 4.5: Logging Example with Bunyan

```
1 const bunyan = require('bunyan')
2 let log = bunyan.createLogger({name: 'myapp'})
3 log.info('hi')
4 log.warn({lang: 'fr'}, 'au revoir')
5 // => {"name":"myapp","hostname":"dev.local","pid":40161,"level":30,"msg":"hi","time":"2016-10-12T18:46:23.851Z","v":0}
6 // => {"name":"myapp","hostname":"dev.local","pid":40161,"level":40,"lang":"fr","msg":"au revoir","time":"2016-10-12T18
      :46:23.853Z","v":0}
```

when the event occurred. These can be seen in Listing 4.5 which shows an example of unprocessed log output of Bunyan. This demonstrates that Bunyan can show each distinct incident, the completeness of the information within and therefore their admissibility in a court of law.

- Logs are written to their final destination in order of occurrence. All logs are timestamped and discrepancies can be detected quickly. Logs of any and all complete incidents can be found. Bunyan also has the ability to attach additional data to the log output for a more complete view of an incident. These show that Bunyan is capable of creating logs to show the complete activity an incident.

- By default, Bunyan outputs the logs it collects to "stdout"; however, Bunyan can output its logs to any Node JS stream. It is possible to implement the Node JS stream interface to send logs to a centralised log storage service and/or intrusion detection system. If logging is done in this manner, and access to the centralised log storage service was appropriately controlled, the requirement for proving integrity can be considered satisfied.

- While Bunyan by default assumes that the logs will be processed by another automated service and therefore produces JSON logs, these logs can be processed and made more human readable. Bunyan is packaged with a command-line interface (CLI) that reads Bunyan JSON logs and "pretty prints" them. This satisfies the requirement for logs to be legible in court by non-specialist personnel.

In the simplest case, logging to a file or multiple files that are appropriately backed up will satisfy the requirements for testing a proof of concept system. As the focus of this research is attribute-based access control, there will be no automated auditing component in the proof of concept. Instead, the system will be audited by hand.

4.3.5 JOSE

JOSE is a JSON Object Signing and Encryption library created by Anvil Research (Smith *et al.*, 2016a). This library implements and provides abstractions for cryptographic keys, JWT's and the signing and verification thereof. This is useful to the research in that the library is written and reviewed by specialists in the field and is less likely to have errors or vulnerabilities as a result of more scrutiny.

4.3.6 Lodash

According to Dalton (2016), Lodash is a Javascript utility library providing "modularity, performance and extras". Lodash provides a number of utilities for iterating of Javascript data structures using methods with interfaces similar to those provided in the functional programming paradigm such as map, reduce and zip. Due to the fact that Lodash is a mature research and well tested with a testing coverage of 99.75% at the time of writing, lodash can be considered void of irregular behaviour. Lodash will prove useful to the research for simplifying data manipulation.

4.4 Revisions

It was decided that to demonstrate a viable proof of concept attribute-based access control system, only the most basic functionality need be provided while focusing on the main goal of simplifying the system and maintaining the security requirements as listed in Section 3.2.2. While the system may not have many of the features originally intended, the design of the system is such that they can be added at a later stage with minimal refactoring of the code. Each of the revisions made with this decision in mind are discussed individually.

4.4.1 Architecture

It was originally intended that the subsystems discussed in Sections 4.6 and 4.7 were to be deployable alongside the barrier (locally) or as standalone services by creating a wrapper application to publish the functionality by means of REST endpoints. Deploying the authorization engine and store separately allows for each to load balanced for scalability

in large deployments. Due to lack of relevance to the subject of this research, the development of standalone implementations of these subsystems has been moved to future work. This research will implement the simplest solution for the sake of the proof of concept; all of the subsystems will be deployed on a single host alongside the barrier.

4.4.2 Rules

This research would have made use of a hybrid approach to access control rule syntax as described in Section 3.5.3, however it was decided that simplifying the rule syntax could be achieved with an existing language, albeit to a lesser extent. The hybrid approach is still achievable at a later stage and therefore the creation of a new language which would compile ABAC rules written in this to the intermediate representation has been moved to the future work section of this document.

4.5 Attribute-Based Access Control Rules

As mentioned in Section 3.5.2, JSON Schema has all the required features to act as an attribute-based access control language and is used in this research for that reason. Listing 4.6 shows an example schema as it would be used in order to enact access control. The rule shown in the listing can be read as follows:

> *Allow access between 07h30 and 17h00, if the subject is a staff member of the Computer Science or Information Systems departments.*

4.5.1 Limitations

JSON Schema is not the ideal language for rules because it still contains a number of redundant controls such as the "required" keyword. The keyword is redundant because within ABAC, if an attribute is specified, it is both necessary and required. JSON Schema also includes a simple type system that corresponds to the types available in Javascript, such as strings, numbers, booleans, functions and objects. This type system further complicates the syntax as it results in the mandatory use of the "properties" keyword. This results in two levels of JSON nesting for each nested object. A language written

Listing 4.6: Example ABAC Rule using JSON Schema as an intermediate representation

```
{
  type: 'object',
  required: ['subject', 'environment'],
  properties: {
    subject: {
      type: 'object',
      required: ['staff', 'department'],
      properties: {
        staff: {
          type: 'boolean',
          enum: [true]
        },
        department: {
          type: 'string',
          enum: ['Computer Science', 'Information Systems']
        }
      }
    },
    environment: {
      type: 'object',
      properties: {
        time: {
          type: 'object',
          required: ['hours', 'minutes'],
          anyOf: [
            {
              properties: {
                hours: {
                  type: 'number',
                  minimum: 7,
                  maximum: 17
                },
                minutes: {
                  type: 'number',
                  minimum: 30
                }
              }
            },
            {
              properties: {
                hours: {
                  type: 'number',
                  maximum: 17,
                  minimum: 8
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

to remove these complexities and simplify the ABAC rule syntax could address this by filling in the necessary "properties", "required" and "type" keywords as needed.

This however, is not mandatory. JSON Schema alone is an improvement on the XACML rule syntax with regards to being simpler, more efficient and easier to use as a result of its generous spacing and the decreased control characters in JSON. Users are cautioned to extensively test access control rules written with JSON Schema.

4.6 Authorization Engine

The authorization engine library exposes three methods for use by the barrier.

1. **Register**: This allows the barrier to register rules on the authorization engine. Rules are parsed and validation functions are compiled at the time they are registered. This causes a longer setup time when the process is first started. After setup is complete, time taken to serve an authorization request is decreased as a result of not having to parse the rule each time.
2. **Attributes**: This allows the barrier to query the authorization engine for a list of the attributes required to evaluate a particular rule. This is used by the barrier to perform a pre-evaluation validation against the attributes returned from the identity provider and store. The array of attributes are returned as JSON Pointer strings for use by the store API.
3. **Enforce**: This accepts the name of the rule and a set of subject, object or resource and environment attributes and evaluates them against the rule that is cached under the name supplied. This produces a boolean output to allow or deny the user access. Unlike XACML, the response can only be allow or deny; there are no ambiguous responses, such as XACML's "indeterminate" or "not applicable" responses.

The authorization makes use of the JSON Schema implementation from JSON Document (Section 4.3.1) in order to parse rules and generate validation functions.

4.6.1 Limitations

- The authorization engine prepares rules for use in access control immediately after registration. Due to the scope being limited to local deployment, the authorization

engine currently makes no effort to check if the rule has been updated or not. If the rule is updated, it needs to be re-registered on the authorization engine. The easiest way to accomplish this is for the process to be restarted.

- The authorization engine checks for malformed input, however this does not protect against syntactically correct, semantically incorrect rules.

4.7 Store

The store provides three methods for use by the barrier.

1. **Get Attribute**: This accepts an array of JSON Pointer strings which are evaluated against the object containing the current available attributes. The attribute or attributes described by the JSON Pointer string are returned.
2. **Get Rule**: This allows the barrier to get the JSON Schema object that describes an attribute-based access control rule by name.
3. **Get Configuration**: Similar to the "Get Rule" method, this allows the barrier to fetch configuration from the store corresponding to its own name.

When the store is constructed, it does a recursive directory scan of the directories it is provided in its constructor. Any Javascript module in those directories that exports an object is parsed and merged onto an object stored in memory. As discussed in Section 3.6.3, attributes in these Javascript files can be defined in terms of a function. Functions defined this way must take no parameters and must produce a serializable output. When such an attribute is requested from the store, the function is evaluated and the result is returned to the barrier as static data.

4.7.1 Revisions

It was originally planned that the store, in addition to the points mentioned in Section 4.4.1, would provide a means to change the method of data storage to a centralised location in order to support scalability. Using a database to store attributes, rules and configuration would also allow the possibility of a web, mobile or desktop application for

remote administration of individual barriers. The result is that identical barriers in large deployments could be deployed using a pair of environment variables; one for the address of the store and another to select which configuration it should use.

These planned features were deemed unnecessary for a proof of concept implementation and have been moved into the future work section of this document.

4.7.2 Limitations

- Due to the store being limited to local deployment alongside the barrier, the store parses and constructs the rules, configuration and attributes when it is instantiated. The store makes no attempt to check if any of those have been updated or not, which affects both the barrier and authorization engine's operation. As discussed in Section 4.6.1, the simplest way to ensure that the changes take effect is to restart the process.
- The store does not make any attempt to validate the data that it loads. As long as the data is in a valid syntax it will load it into memory and execute functions to generate attributes as they are requested. As the store data is loaded from local files, this does not impact security as long as the host system is secure. The task of validating store data falls to the barrier and authorization engine.

4.8 Barrier

The barrier is the central system within this architecture, and has a single method for external use: **Enforce**. All other complexity is handled within. The enforce method operates as follows:

1. The enforce method accepts an identifier argument, such as one read from a RFID tag.
2. The OpenID Connect (OIDC) authorization code flow in order to access the subject's attributes.
3. Once the OIDC authorization code flow is complete, the subject attributes are combined with the object and environment attributes retrieved from the store.

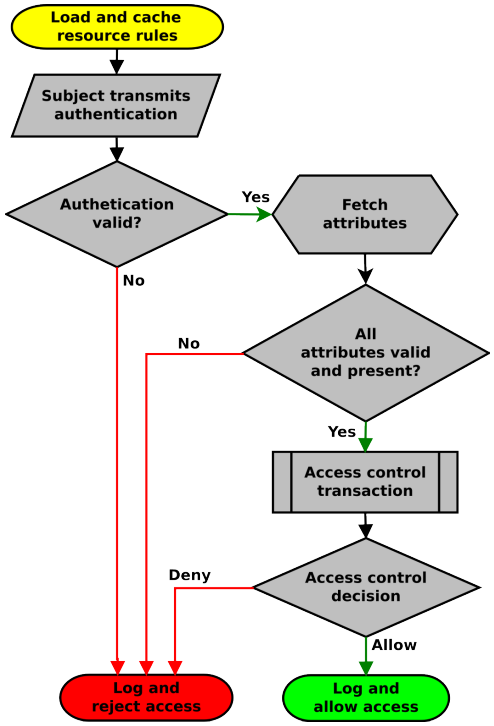


Figure 4.3: Final Barrier Authorization Transaction Flow Chart

4. All of the attributes gathered are validated against the list of required attributes for the rule from the authorization engine.

5. The attributes are passed to the authorization engine and the access control decision is awaited.

6. The access control decision is enforced and access is granted or denied.

Each part of the above transaction is logged. The barrier constructor can be passed many forms of input:

- The absolute or relative path to a configuration document or store directory. In the case of the latter the barrier defaults to using "/config" as the JSON Pointer string to retrieve its configuration.
- An array of absolute or relative paths to store directories. The barrier retrieves its configuration in the same way as above.
- An object containing the configuration for the barrier, or the absolute or relative path to one or many store directories and optionally a JSON Pointer string for the location of the desired configuration.

- An instance of the “Config” class.

The configuration descriptor can take on many forms. Either the configuration can be provided within the object itself, or the object can reference the configuration it should use from the store. The least that is required for the barrier’s construction is a path to a store directory that results in the necessary configuration being loaded for use by the barrier.

4.8.1 Revisions

Section 3.6.1 and Figure 3.1 show the original plan of the barrier was to include support for multi-factor authentication (MFA), however this proved difficult due to the decision to handle authentication as part of the Open ID Connect authorization flow. For this reason MFA was discontinued and moved to future work. A revised flow chart diagram reflecting implementation of the barrier can be seen in Figure 4.3.

4.8.2 Limitations

The decision was made to defer the authentication of users to the OpenID Connect identity provider. In order to achieve this a customised protocol was added to the identity provider. The protocol enables browser-less user login using a single identifier that would be contained on a RFID token, however the protocol was added without any additional security mechanisms. The identifier alone cannot be considered enough to authenticate users as the identifier on an RFID token can be stolen or intercepted too easily by a malicious actor. To combat this in the short term, the identity provider should not allow dynamic registration of relying parties and should be configured to only accept requests on this protocol from specified IP addresses. This, however, only provides a countermeasure for legitimate tokens from being used to access resources they shouldn’t; it does not secure the identity information of the user stored on the identity provider.

4.9 Chapter Summary

This chapter provides insight into the precise implementation detail of the system architecture for this research as it was implemented as opposed to how it was planned. The

development environment and core, but non-exhaustive list of libraries used for this are explained, as well as the respective reasons for their use. The authorization engine, store and barrier were individually explained, discussed and analysed with respect to the API they each provide and the restrictions to their functionality. Lastly, changes in design as a result of adapting to difficulties experience have been discussed, both on a global (across all subsystems) level and individually.

The following chapter details the evaluation of the system against the goals and design discussed in Chapter 3.

Chapter 5

Results & Analysis

5.1 Introduction

This chapter evaluates the system implementation that was discussed in Chapter 4 against the design and success criteria defined in Chapter 3. The chapter begins by reporting the results observed in the form of logs and tests before continuing to analyse the system design against the success criteria. The results and analysis show that the system is functional and working according to the original specification — albeit with less than the initially planned features. Finally the strengths and points of failure for the system are discussed.

5.2 Results

The following results were obtained from a local deployment of the system, in a controlled environment.

5.2.1 Logs

In Section 2.3, the importance of logs for application in security as well as their requirements are discussed and their requirement in this research as a criteria for success is stipulated in Section 3.2.2. The logging output of the system can be seen in Appendix A.

Bunyan and Log Requirements

As discussed in Section 4.3.4, Bunyan provides most of the functionality required in Section 2.3.2. By using Bunyan, the following requirements are fulfilled:

- Bunyan logs are able to be related to any distinct incident in question.
- Bunyan logs can be presented in a human readable manner.

Log Integrity

The simplest way of demonstrating log integrity is to store logs on a bespoke system with strict access control. For the purpose of the proof of concept system described in this document, logs were saved locally to the host of the barrier. Logs saved on the local device are trustworthy in the same way as a bespoke system so long as access control to the host device is similarly strict, however the security of the host device is not within the scope of this research. Threats to this include possible bugs in the barrier library that would allow the barrier's host to become compromised.

Complete Incident Logging

In the authorization process of access control, the incidents of interest are authentication and access attempts — regardless of whether they are successful or not. As can be seen in Listing A.5, user authentication and access attempts are logged at the "INFO" level. Logs at this level or above are what is displayed and stored when the system operates in a production environment, as the information they display is considered important from an operational or security standpoint.

Having satisfied all of the requirements for logs in Section 2.3.2, the logging component, as required for application security, is also satisfied.

5.2.2 Tests

Testing initially was incorporated into the research to provide a quantifiable mechanism to strengthen the assertion that the system described in this document adhered to the

Table 5.1: Summary of Test Output

Library	Total	Passing	Pending	Failing
Store	20	20	0	0
Authorization Engine	10	10	0	0
Barrier	207	113	91	0
	237	143	91	0

security requirements in Section 3.2.2. Unit and integration testing and TDD are tools for decreasing programmer error in code. Testing was used in this research to test adherence to the design of the system, including security sensitive matters and logging. The test output can be seen in Appendix B.

Output

Tests shown in the output are divided into passing, pending and failing tests. Passing tests are defined as tests that produce the expected behaviour or results. Pending tests are tests that have been specified ("stubbed"), but not written or tests that are skipped. Failing tests are tests that produce unexpected behaviour or results. None of the tests observed in the test output are failing, although there are a number of pending tests for one of the following reasons:

- Time constraints relative to the benefit of writing the test. Some tests were stubbed but not completed as the code was largely tested already in other tests or would for some other reason not increase code coverage to warrant time spent on them.
- Difficulties were experienced writing tests where interaction with external libraries was necessary. Sinon stubs were used to emulate and test for API calls to other libraries without invoking their actual functions; however, some libraries design did not easily allow for this testing method to be used. In cases that were not resolved, the tests were marked with "skip" and display in the output as pending.

The test output of each subsystem is summarised in Table 5.1.

Coverage

The test coverage analysis utility: Istanbul, was used to analyse tests and calculate the total test coverage in various categories as a percentage. The Istanbul test coverage analysis output, as can be seen in Figure B.1, is summarised by Table 5.2.

Table 5.2: Istanbul Coverage Summary

Library	File	Statements	Branches	Functions	Lines
jacl-auth-engine	AttributeExtractor.js	82.22%	70.00%	100.00%	84.09%
	JACL.js	85.71%	64.29%	80.00%	85.71%
	Rule.js	94.74%	70.00%	100.00%	100.00%
		85.87%	**68.18%**	**93.33%**	**87.50%**
jacl-store	Store.js	**86.32%**	**78.26%**	**100.00%**	**86.32%**
jacl-barrier	Barrier.js	93.75%	66.67%	100.00%	93.75%
	Config.js	100.00%	100.00%	100.00%	100.00%
	ConfigSchema.js	100.00%	100.00%	100.00%	100.00%
	Logger.js	92.86%	88.24%	100.00%	92.86%
	OIDCHandler.js	90.76%	66.67%	94.74%	90.76%
	ProviderSchema.js	100.00%	100.00%	100.00%	100.00%
		92.94%	**77.78%**	**97.22%**	**92.94%**

The following observations can be made from this:

- Test coverage is high across the board, with the exception of some files' branching logic. This asserts that the system is functioning as designed.
- Branching logic test scores, particularly in key files such as *JACL.js*, *Barrier.js* and *OIDCHandler.js*, are relatively low. This could indicate that some branching logic intended for handling edge cases or malformed input are not tested and so the behaviour thereof is unknown. Due to security implications, further testing needs to be done before using this code for a production deployment; however, while the score is low, it is sufficient for a proof of concept.

Despite some low coverage scores, the overall score of each library is high enough to assert that the system as a whole is functioning as designed.

5.3 Analysis

Section 3.2 discusses three main criteria for success. This section analyses the system and qualitatively evaluates it against these criteria.

Table 5.3: Comparison of XACML and JSON Schema Rules

Feature	**XACML**	**JSON Schema**
Functionally complete	✓	✓
Data agnostic (can operate on any data set)	✓	✓
Extensible	✓	✓
Referrable (can used as a component in another rule without being redefined)	✓	✓

5.3.1 Comparison to XACML

XACML is an existing attribute-based access control standard, the functionality of which is emulated and certain aspects are improved on by this research. The existing functionality can be split into two categories: rules and architecture; the first is XACML, which is an attribute-based access control standard.

Attribute-based Access Control Rules

This system demonstrates that it implements attribute-based access control in the following ways:

- JSON Schema is used by this system as the rule language for attribute based access control. JSON Schema operates directly on attributes or raw data from various sources and is functionally complete as it has the capability to complete *AND*, *OR* and *NOT* boolean operations. As all boolean operations can be created from *AND* and *NOT*; JSON Schema is therefore functionally complete and meets the requirements for attribute-based access control.
- JSON Schema operates directly on data or attributes. Any data can be used when validating a JSON Schema and therefore any and all data can be used as an attribute for which rules can be written.
- The system provisions for attributes relating to the subject, object and environment which are provided by an identity provider and stores respectively. Stores can be configured to provide attributes using generator functions that are injected when the process starts. The generator functions are user defined and can therefore provide any serializable data as a named environment attribute which is evaluated at

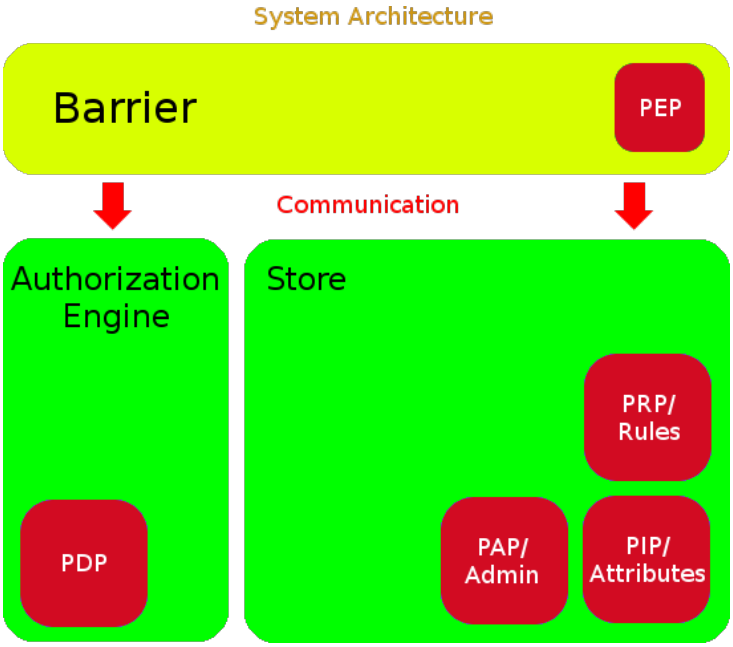


Figure 5.1: Final System Architecture

the time that it is requested. The same can be done for object related attributes. Subject related attributes are fetched from an identity provider which takes responsibility for providing domain specific subject related attributes.

In these regards, the two systems are equivalent in capabilities and the standard of attribute-based access control is maintained. Table 5.3 shows a qualitative comparison of the attribute-based access control rules of this system and XACML.

Architecture

XACML notably provides descriptions of five subsystems in the specification, these are the: policy enforcement point, policy decision point, policy retrieval point, policy information point and policy administration point. Each of these subsystems has a bespoke purpose that is needed to support attribute-based access control.

Figure 5.1 shows which subsystem performs the equivalent task in the system compared to XACML (where the XACML subsystems are represented in red boxes). The barrier and authorization engine complete tasks done by the PEP and PDP respectively and the store handles the remaining tasks of the PIP and PRP. As this system was built to accommodate a small deployment of a single barrier, the PAP functionality was moved to future work; although this functionality would also be implemented on the Store.

It is worth noting that XACML is designed to run its subsystems on dedicated, monolithic servers in order to be able to service many access requests from multiple PEP's. Implementations of XACML are optimised for this and the control access control transaction lies with the PDP, leaving little for the PEP to do other than send access requests and enforce decisions from the PDP. This system was designed to attempt to balance the work load by inverting the control to the barrier. As each individual barrier serves only itself, it is reasonable to use the dedicated computational resources for the task and use less shared resources as this increases overall capacity of the system.

That being said, the system described by this document would not currently be able to replace XACML at the moment. This is a proof of concept implementation to demonstrate the possibility of an alternative and is unlikely to be comparable in efficiency and scalability to XACML at this point in time.

It should be noted that the authorization engine and store libraries were initially planned to be deployed individually — if required — as this is how XACML operates to achieve scalability, however due to this being a proof of concept and not needing that functionality, individual deployment was moved to future work.

Feature Comparison

Table 5.4 shows a comparison between the features of the Sun XACML implementation, as listed on their website (Sun Microsystems, Inc., 2006), and the features offered by JSON Schema as an attribute-based access control language. Due to Sun XACML being closed source and proprietary software, a more detailed comparison between the two cannot be completed at this time and has been moved to future work.

Tests show that the system operates as designed and all of the functionality of XACML is accounted for, therefore the criteria of maintaining the existing standard is complete.

5.3.2 Security

Access control is a mechanism for security within a particular domain. Security mechanisms themselves need to be secure as they cannot exist in isolation. Section 3.2.2 discusses a number of security requirements for the system. Two of these requirements, logging and testing, are addressed in Section 5.2 in this chapter. The remaining security requirements were addressed in the following manner:

Table 5.4: Feature comparison between XACML and this system

Sun's XACML Implementation	**This System**	**Superior**
One standard access control policy language	JSON Schema is language agnostic.	Equal
Good tools for writing and managing XACML policies will be developed, since they can be used with many applications	Easy tools for writing rules dont exist for JSON Schema but its possible.	XACML
XACML is functionally complete and extensible so that new requirements can be supported.	JSON Schema is functionally complete and can be extended or added to.	Equal
One XACML policy can cover many resources. This helps avoid inconsistent policies on different resources.	Rules can be used for multiple resources.	Equal
XACML allows one policy to refer to another. This is important for large organizations. For instance, a site-specific policy may refer to a company-wide policy and a country-specific policy.	JSON Schema supports JSON Pointers which can reference other schemas and subschemas.	This System

- This system communicates via HTTP to various REST endpoints on the identity provider. As this is the only network communication that happens and the OpenID Connect specification demands that all implementations of their standard use only HTTPS, the requirement for encrypted communication is satisfied for this research.
- This system defers authentication (the process of verifying the identity of a subject) to the OpenID Connect identity provider (IdP). In return, the IdP issues a JSON web token (JWT) to the barrier which is signed by the IdP as the guarantor of the subject's identity.
- Verifying the IdP's identity relies on the certificate and DNS name of the IdP.
- The only user input the system receives is the input of the subject's identifier. The system immediately logs and rejects the access attempt if anything other than a numeric identifier is provided in this process.

The system has a very small attack surface as it is the client in the interaction between it and the IdP. There's a single point where the subject provides information and that point

is protected against digital attack. Where this system is not secure is with regard to the physical hardware of the barrier. Anyone who can gain physical access to the barrier's hardware can circumvent the barrier, however security considerations in this regard are considered out of scope for this research.

Tests, logs and the design of the system satisfy all of the security requirements detailed in Section 3.2.2.

5.3.3 Ease of Use

The success criteria for ease of use was defined in Section 3.2.3 as being the successful implementation, testing and deployment of a single barrier attribute-based access control system to protect a single resource. This has been successfully completed, the logs and tests for which can be seen in the appendices of this document.

Other ease of use evaluations can be made in terms of legibility of the JSON Schema rules compared to the XACML XML/RDF rule language as can be seen in Figure 5.2. The evaluation of ease of use in this manner is currently subjective and an objective evaluation of this has been added to future work.

```
[a14]    <Rule
[a15]      RuleId= "urn:oasis:names:tc:xacml:3.0:example:SimpleRule1"
[a16]      Effect="Permit">
[a17]      <Description>
[a18]        Any subject with an e-mail name in the med.example.com domain
[a19]        can perform any action on any resource.
[a20]      </Description>
[a21]      <Target>
[a22]        <AnyOf>
[a23]          <AllOf>
[a24]            <Match
[a25]              MatchId="urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match">
[a26]            <AttributeValue
[a27]              DataType="http://www.w3.org/2001/XMLSchema#string"
[a28]                >med.example.com</AttributeValue>
[a29]            <AttributeDesignator
[a30]              MustBePresent="false"
[a31]              Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
[a32]              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
[a33]              DataType="urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name"/>
[a34]            </Match>
[a35]          </AllOf>
[a36]        </AnyOf>
[a37]      </Target>
[a38]    </Rule>
```

Figure 5.2: XACML Rule Example

JSON Schema benefits from JSON's decreased control characters and size when compared to XML/RDF. JSON is typically represented in an appropriately spaced out manner for better legibility and lacks a static type system that can be seen in XML/RDF. This lack of a static typing mechanism results in each value simply being referable by a unique key.

The result is that JSON contains less control characters as it only needs to distinguish between four types of data at the cost of not having a more complex type system, which lends legibility to its serialized output.

JSON Schema is a standard for applying schemas to schemaless data. The definition of a schema — and therefore attribute-based access control rules using JSON Schema — adds complexity to JSON and decreases its legibility in return for a complex type system. JSON Schema can be used to specify the form (or schema) of JSON objects; customised data types, formats, regular expressions and data attributes can be used to create a complex structure that data must adhere to. The JSON Schema method of applying complex structure to unstructured data replaces the functionality provided by XML/RDF for this purpose; albeit still without a static type mechanism, although this kind of type mechanism is not required within the context of attribute-based access control. JSON Schema still looks and reads better than XACML XML/RDF rules for use in attribute-based access control.

5.3.4 Efficiency

In Section 2.6.1, an investigation into the efficiency of JSON compared to XACML in terms of size — and therefore bandwidth consumption — and computational cost for deserialization concludes that JSON is superior in both. This conclusion has been accepted by this research and no further attempt has been made to confirm them.

5.4 Chapter Summary

This chapter began with a report of the results and observation of the logs and test output from the system in a controlled deployment and continued into an evaluation of the system implementation against the success criteria provided in Chapter 3. Qualitative data was used in the evaluation, so as to justify success criteria being labeled as met or complete and the flaws and points of failure was discussed. The system was found to be functional and comparable to the core functionality found in XACML implementations in accordance with the criteria for success.

Chapter 6

Conclusion

6.1 Introduction

This chapter serves to draw key conclusions from the analysis, discussion and results of Chapter 5. In addition, this chapter addresses the research objectives in Chapter 1 and discusses possible future work to address the limitations, weaknesses and extensions of this research.

6.2 Summary

The aim of this research was to create an attribute-based access control system that improved on the existing standard (XACML) by using modern web technologies and languages to improve ease of use and decrease the barrier to entry for attribute-based access control for small deployments.

The improvements made included simplifying the architecture and using an alternative rule language that is more understandable. The architecture was simplified and reduced from five subsystems to three — albeit lacking some of the initial planned features — and an alternative rule language: JSON Schema, was found to be a suitable replacement for XACML's XML/RDF rules.

A summary of obstacles and observations that were encountered during the execution of this research, together with their resolutions are as follows:

1. The initial specification of the research was overly ambitious and not achievable within the resources allocated to the completion of this research. This was resolved by the scope of the research being narrowed significantly.

2. Evaluation of the system proved to be difficult due to implementations of XACML being proprietary and closed source. In order to combat this testing and logs were introduced as performance indicators for the system and a qualitative feature comparison was done against the features listed by Sun Microsystem's XACML implementation website. The inclusion of testing and logs do not mean that the assertion of complete functionality or security can be made, instead all that can be asserted is that the code written is behaving as expected. A test coverage utility was added to the research in an attempt to quantify this.

3. Evaluating the "improved ease of use" proved difficult to execute objectively. "Ease of deployment" was evaluated by successfully deploying the system in a controlled environment to protect a single resource; hence showing that the system was simple enough to be used for a single resource without a disproportionate amount of hardware required to support it. Although this deployment was successful and the result positive, the system was slow and can be considered a "naive" implementation.

4. Evaluating the "improved ease of use" of the system with regard to the rule structure could not be completed objectively. Instead a subjective analysis was settled on.

6.3 Conclusion

Conclusions that can be drawn from this research are:

- The complex operation of attribute-based access control and supporting software can be simplified to operate on a single host in a small deployment.
- Attribute-based access control using JSON and related technologies is possible and is more bandwidth efficient due to size difference between JSON and XML/RDF.
- Attribute-based access control can be successfully executed with fewer subsystems than those defined by XACML.

Although the system is not perfect and full functionality and security have no way of being guaranteed; the above answers to the questions posed in Section 1.5 allow us to conclude

that it is possible to build a simple, modern attribute-based access control system for use in small deployments.

6.4 Future Work

Throughout the research, various "nice-to-have" features were relegated to this section of the document, as their function was not necessary for the critical operation of the system defined by this document; others were moved here due to a variety of resource constraints such as time. A few of these recommendations are outlined below:

1. **Improved end-user ABAC rule creation**: as discussed in Section 3.5.3, JSON Schema works well as an intermediate access control language, however it is not ideal as an end-user ABAC language. This work would benefit from the creation of a tool to ease the writing of ABAC rules or a new access control language that would compile to JSON Schema.

2. **Data storage abstraction**: From the perspective of scalability; this system would benefit from being able to access store data (rules, attributes and configuration) from a database. This would allow the store subsystem to be scaled when deployed separately.

3. **Separate deployment wrapper implementations**: Currently each of the libraries defined in this document are deployed locally on the barrier as the test case for the system was a standalone barrier. In order for the store and authorization engine to be able to service multiple barriers, they must be deployed individually. Creating a wrapper for these libraries to expose their functionality via REST endpoints would allow the system to be scalable.

4. **Neutral, objective evaluation**: This research would benefit from objective evaluations of the system's relative ease of use compared to XACML as well as the system's performance and possible ways to improve it.

5. **Remote administration**: This feature was not required by the research as it consisted of a single barrier. If work was done to scale the system, it would benefit from having a remote administration mechanism. Potentially a web, desktop or mobile application, or any combination thereof.

6. **Automated deployment**: Automating the deployment of barriers in a large deployment using an OpenID Connect-like discovery mechanism could allow for the rapid deployment of barriers within a large organisation or domain.

6.5 Concluding Remarks

This work has provided an invaluable educational experience for the researcher. Despite an overly ambitious start, delving into the topic area from the broad to the specific has provided a unique insight into the function and design of access control and security services in general. This experience will serve as a firm foundation for creating a production ready attribute-based access control solution and future research into distributed identity.

References

Aloul, Fadi, Zahidi, Syed, & El-Hajj, Wasim. 2009. Multi factor authentication using mobile phones. *International Journal of Mathematics and Computer Science*, **4**(2), 65–80.

Anantheswaran, Krishnan. 2016. Istanbul - Yet another JS code coverage tool that computes statement, line, function and branch coverage with module loader hooks to transparently add coverage when running tests. Github Repository. Available from: https://github.com/gotwarlost/istanbul.

Armando, Alessandro, Grasso, Matteo, Oudkerk, Sander, Ranise, Silvio, & Wrona, Konrad. 2013. Content-based information protection and release in NATO operations. *Pages 261–264 of: Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*. ACM.

Association for Computing Machinery, Inc. 2012. ACM Computing Classification System. Retrieved from http://www.acm.org/about/class/2012/.

Bhargav-Spantzel, Abhilasha, Squicciarini, Anna C, Modi, Shimon, Young, Matthew, Bertino, Elisa, & Elliott, Stephen J. 2007. Privacy preserving multi-factor authentication with biometrics. *Journal of Computer Security*, **15**(5), 529–560.

Bryan, P, & Nottingham, Mark. 2013. JavaScript Object Notation (JSON) Patch Specification. *IETF Standard*.

Bryan, Paul, & Zyp, Kris. 2011. JavaScript Object Notation (JSON) Pointer Specification. *IETF Standard*.

Casey, Donal. 2008. Turning log files into a security asset. *Network Security*, **2008**(2), 4–7.

Dalton, John-David. 2016. Lodash - A modern JavaScript utility library delivering modularity, performance, & extras. Github Repository. Available from: https://github.com/lodash/lodash/.

Denicola, Domenic. 2016. Sinon Chai - Extends Chai with assertions for the Sinon.JS mocking framework. Github Repository. Available from: https://github.com/domenic/sinon-chai.

Farrell, Stephen, & Housley, Russell. 2002. An internet attribute certificate profile for authorization. *IETF Standard*.

Ferraiolo, David, Cugini, Janet, & Kuhn, D Richard. 1995. Role-based access control (RBAC): Features and motivations. *Pages 241–48 of: Proceedings of 11th Annual Computer Security Application Conference*.

Forte, Dario. 2005. Log management for effective incident response. *Network Security*, **2005**(9), 4–7.

Forte, Dario. 2009. The importance of log files in security incident prevention. *Network Security*, **2009**(7), 18–20.

Galiegue, Francis, & Zyp, Kris. 2013. JSON Schema: Core definitions and terminology. *IETF Standard*.

Gorge, Mathieu. 2007. Making sense of log management for security purposes–an approach to best practice log collection, analysis and management. *Computer Fraud & Security*, **2007**(5), 5–10.

Gunson, Nancie, Marshall, Diarmid, Morton, Hazel, & Jack, Mervyn. 2011. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. *Computers & Security*, **30**(4), 208–220.

Hallam-Baker, Phillip. 2001. Security Assertions Markup Language. *OASIS Standard*, **14**, 1–24.

Hameseder, Katrin, Fowler, Scott, & Peterson, Anders. 2011. Performance analysis of ubiquitous web systems for smartphones. *Pages 84–89 of: Performance Evaluation of Computer & Telecommunication Systems (SPECTS), 2011 International Symposium on*. IEEE.

Hardt, D. 2012. RFC6749 The OAuth 2.0 Authorization Framework. *IETF Standard*, October.

Holowaychuk, TJ. 2016. Mocha - simple, flexible, fun javascript test framework for Node.JS & the browser. Github Repository. Available from: https://github.com/mochajs/mocha.

Hu, Vincent C, Ferraiolo, David, Kuhn, Rick, Schnitzer, Adam, Sandlin, Kenneth, Miller, Robert, & Scarfone, Karen. 2014. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special Publication*, **800**, 162.

Jin, Xin. 2014. *Attribute-based access control models and implementation in cloud infrastructure as a service*. University of Texas.

Jin, Xin, Krishnan, Ram, & Sandhu, Ravi S. 2012. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. *DBSec*, **12**, 41–55.

Johansen, Christian. 2016. Sinon - Test spies, stubs and mocks for JavaScript. Github Repository. Available from: https://github.com/sinonjs/sinon.

Jones, M, & Hildebrand, J. 2015. JSON Web Encryption (JWE) Specification. *IETF Standard*.

Jones, Michael. 2015a. JSON Web Algorithms (JWA) Specification. *IETF Standard*.

Jones, Michael. 2015b. JSON Web Key (JWK) Specification. *IETF Standard*.

Jones, Michael, Tarjan, Paul, Bradley, J, Goland, Yaron, Sakimura, Nat, Panzer, John, & Balfanz, Dirk. 2011. JSON Web Signature (JWS) Specification. *IETF Standard*.

Jones, Michael, Tarjan, Paul, Bradley, J, Goland, Yaron, Sakimura, Nat, Panzer, John, & Balfanz, Dirk. 2012. JSON Web Token (JWT) Specification. *IETF Standard*.

Kapke, William. 2016. Node.js ES2015 Support. Online Report. Available from: http://node.green/.

Kent, Karen, & Souppaya, Murugiah. 2006. *Guide to computer security log management: recommendations of the National Institute of Standards and Technology (NIST)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology.

Klyne, Graham, & Carroll, Jeremy J. 2006. Resource description framework (RDF): Concepts and abstract syntax. *W3C Standard*.

Luer, Jake. 2016. Chai - BDD / TDD assertion framework for node.js and the browser that can be paired with any testing framework. Github Repository. Available from: https://github.com/chaijs/chai.

Ma, Qingxiong, & Pearson, J Michael. 2005. ISO 17799: "Best Practices" in Information Security Management? *Communications of the Association for Information Systems*, **15**(1), 32.

Matyáš, Václav, & Říha, Zdeněk. 2002. Biometric authenticationsecurity and usability. *Pages 227–239 of: Advanced Communications and Multimedia Security*. Springer.

Mick, Trent. 2016. Bunyan - a simple and fast JSON logging module for Node.JS services. Github Repository. Available from: https://github.com/trentm/node-bunyan/.

Mohan, Apurva, & Blough, Douglas M. 2010. An attribute-based authorization policy framework with dynamic conflict resolution. *Pages 37–50 of: Proceedings of the 9th Symposium on Identity and Trust on the Internet*. ACM.

M'Raihi, D., Machani, S., Pei, M, , & Rydell, J. 2011. TOTP: Time-Based One-Time Password Algorithm. *Internet Engineering Task Force (IETF)*, May.

Nair, Srijith. 2013. XACML Reference Architecture. Axiomatics Blog. Available from: https://www.axiomatics.com/blog/entry/xacml-reference-architecture.html.

Nottingham, Mark. 2012. JSON or XML: Just Decide. Personal Blog. Available from: https://www.mnot.net/blog/2012/04/13/json_or_xml_just_decide. Accessed: 2016-10-30.

Nurseitov, Nurzhan, Paulson, Michael, Reynolds, Randall, & Izurieta, Clemente. 2009. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Caine*, **2009**, 157–162.

Priebe, Torsten, Dobmeier, Wolfgang, Schläger, Christian, & Kamprath, Nora. 2007. Supporting attribute-based access control in authorization and authentication infrastructures with ontologies. *Journal of Software*, **2**(1), 27–38.

Ratha, Nalini K., Connell, Jonathan H., & Bolle, Ruud M. 2001. Enhancing security and privacy in biometrics-based authentication systems. *IBM systems Journal*, **40**(3), 614–634.

Richardson, Robert. 2008. CSI computer crime and security survey. *Computer Security Institute*, **1**, 1–30.

Rissanen, Erik. 2013. Extensible access control markup language (XACML) version 3.0. *OASIS Standard*.

Saint-Germain, René. 2005. Information security management best practice based on ISO/IEC 17799. *Information Management*, **39**(4), 60.

Sakimura, Natsuhiko, Bradley, J, Jones, M, de Medeiros, B, & Mortimore, C. 2014. OpenID Connect Core Specification. *The OpenID Foundation.*

Sandhu, Ravi. 1996. Role hierarchies and constraints for lattice-based access controls. *Pages 65–79 of: Computer Security – ESORICS 96.* Springer.

Sandhu, Ravi, Ferraiolo, David, & Kuhn, Richard. 2000. The NIST model for role-based access control: towards a unified standard. *In: ACM Workshop on Role-based Access Control*, vol. 2000.

Sandhu, Ravi S, & Samarati, Pierangela. 1994. Access control: principle and practice. *Communications Magazine, IEEE*, **32**(9), 40–48.

Sandhu, Ravi S, Coyne, Edward J, Feinstein, Hal L, & Youman, Charles E. 1996. Role-based access control models. *Computer*, 38–47.

Smith, Christian, Zagidulin, Dmitri, & Linklater, Gregory. 2016a. JOSE - JSON Object Signing and Encryption for Node.js and the browser. Github Repository. Available from: https://github.com/anvilresearch/jose.

Smith, Christian, Linklater, Gregory, & Zagidulin, Dmitri. 2016b. JSON Document - Fast, declarative, standards-based JavaScript (ES6) data modeling & manipulation. Github Repository. Available from: https://github.com/anvilresearch/json-document.

Sun Microsystems, Inc. 2006. Sun XACML Homepage. Website. Available from: http://sunxacml.sourceforge.net/.

Yuan, Eric, & Tong, Jin. 2005. Attributed based access control (ABAC) for web services. *In: Proceedings of IEEE International Conference on Web Services and ICWS, 2005.* IEEE.

Appendices

Appendix A

Logs

A.1 Setup

A.1.1 Configuration

Listing A.1: Barrier configuration for controlled deployment test: allow access

```
{
  "provider": {
    "name": "Identifier",
    "signin": "https://openid.ict.ru.ac.za/connect/card",
    "issuer": "https://openid.ict.ru.ac.za",
    "client_id": "cbaff460-a4ea-40b1-aee1-d5893e6558f5",
    "client_secret": "53a6326ac4475675867d",
    "redirect_uris": ["https://barrier.redirect"],
    "scope_attributes": {

    }
  },
  "stores": ["./test/store"],
  "access": "allow"
}
```

Listing A.2: Barrier configuration for controlled deployment test: deny access

```
{
  "provider": {
    "name": "Identifier",
    "signin": "https://openid.ict.ru.ac.za/connect/card",
    "issuer": "https://openid.ict.ru.ac.za",
    "client_id": "cbaff460-a4ea-40b1-aee1-d5893e6558f5",
    "client_secret": "53a6326ac4475675867d",
    "redirect_uris": ["https://barrier.redirect"],
    "scope_attributes": {

    }
```

```
  },
  "stores": ["./test/store"],
  "access": "deny"
}
```

A.1.2 Rules

Listing A.3: Rules for controlled deployment test

```
{
  rules: {
    allow: {
      type: 'object',
      required: ['subject', 'environment'],
      properties: {
        subject: {
          type: 'object',
          required: ['staff', 'department'],
          properties: {
            staff: {
              type: 'boolean',
              enum: [true]
            },
            department: {
              type: 'string',
              enum: ['Computer Science', 'Information Systems']
            }
          }
        },
        environment: {
          type: 'object',
          properties: {
            time: {
              type: 'object',
              required: ['hours', 'minutes'],
              anyOf: [
                {
                  properties: {
                    hours: {
                      type: 'number',
                      minimum: 7,
                      maximum: 17
                    },
                    minutes: {
                      type: 'number',
                      minimum: 30
                    }
                  }
                },
                {
                  properties: {
                    hours: {
                      type: 'number',
                      maximum: 17,
                      minimum: 8
                    }
                  }
                }
              ]
            }
          }
        }
      }
    },
    deny: {
      type: 'object',
      required: ['subject', 'environment'],
```

```
      properties: {
        subject: {
          type: 'object',
          required: ['staff', 'department'],
          properties: {
            staff: {
              type: 'boolean',
              enum: [false]
            },
            department: {
              type: 'string',
              enum: ['Computer Science', 'Information Systems']
            }
          }
        },
        environment: {
          type: 'object',
          properties: {
            time: {
              type: 'object',
              required: ['hours', 'minutes'],
              anyOf: [
                {
                  properties: {
                    hours: {
                      type: 'number',
                      minimum: 7,
                      maximum: 17
                    },
                    minutes: {
                      type: 'number',
                      minimum: 30
                    }
                  }
                },
                {
                  properties: {
                    hours: {
                      type: 'number',
                      maximum: 17,
                      minimum: 8
                    }
                  }
                }
              ]
            }
          }
        }
      }
    }
  }
}
```

A.2 Allow

Listing A.4: Log output for Barrier library (default format: JSON) for rule: allow

```
1 {"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"path":"./test/config/dev.json","msg":"importing
      config","time":"2016-10-29T09:22:43.610Z","v":0}
2 {"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"Barrier configured","time":"2016-10-29T09
      :22:43.610Z","v":0}
3 {"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"Stores built","time":"2016-10-29T09:22:43.617Z
      ","v":0}
4 {"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"Engine built","time":"2016-10-29T09:22:43.621Z
      ","v":0}
```

```
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"rule":"allow","msg":"Starting access control","time":"2016-10-29T09:22:43.626Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"attributes":["/subject/staff","/subject/department","/environment/time/hours","/environment/time/minutes"],"msg":"needed attributes","time":"2016-10-29T09:22:43.627Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"fetched store attributes","time":"2016-10-29T09:22:43.628Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"identifier":42,"msg":"Handle OIDC Auth","time":"2016-10-29T09:22:43.628Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"OIDC Discovery for issuer https://openid.ict.ru.ac.za","time":"2016-10-29T09:22:43.629Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Handle config response and fetch JWKs from https://openid.ict.ru.ac.za/jwks","time":"2016-10-29T09:22:44.089Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Fetched 2 JWKs","time":"2016-10-29T09:22:44.577Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Starting session at https://openid.ict.ru.ac.za/connect/card","time":"2016-10-29T09:22:44.697Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"text":{},"msg":"Handle signin response","time":"2016-10-29T09:22:45.087Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"identifier":42,"msg":"authenticate at https://openid.ict.ru.ac.za/connect/card/callback","time":"2016-10-29T09:22:45.088Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"text":{},"msg":"Handle authenticate response","time":"2016-10-29T09:22:45.591Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":30,"identifier":42,"code":"0856f0f2b8258dcb9383","msg":"User authenticated","time":"2016-10-29T09:22:45.592Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Fetching token","time":"2016-10-29T09:22:45.593Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Handle token response","time":"2016-10-29T09:22:45.915Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"awaiting JWK","time":"2016-10-29T09:22:45.915Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"verifying JWT","time":"2016-10-29T09:22:45.917Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Fetching user info","time":"2016-10-29T09:22:45.935Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"userinfo":{"sub":"cd5c5e40-b5aa-4c56-8f46-2af4b2f4fbe2","family_name":"Linklater","given_name":"Greg","updated_at":1477297183230,"email":"g1214025@campus.ru.ac.za","department":"Computer Science","staff":true},"msg":"Validating Userinfo against required fields","time":"2016-10-29T09:22:46.277Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"attributes prepared for Auth Engine","time":"2016-10-29T09:22:46.278Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":30,"msg":"Saving cookies!","time":"2016-10-29T09:22:46.279Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":30,"decision":true,"identifier":42,"msg":"Access control decision","time":"2016-10-29T09:22:46.283Z","v":0}
```


Listing A.5: Log output for Barrier library (pretty printed) for rule: allow

```
09:20:26.705Z DEBUG barrier: importing config (path=./test/config/dev.json)
09:20:26.705Z DEBUG barrier: Barrier configured
09:20:26.714Z DEBUG barrier: Stores built
09:20:26.718Z DEBUG barrier: Engine built
09:20:26.723Z DEBUG barrier: Starting access control (rule=allow)
09:20:26.724Z DEBUG barrier: needed attributes
    attributes: [
      "/subject/staff",
      "/subject/department",
      "/environment/time/hours",
      "/environment/time/minutes"
    ]
09:20:26.725Z DEBUG barrier: fetched store attributes
09:20:26.726Z DEBUG oidc: Handle OIDC Auth (identifier=42)
09:20:26.727Z DEBUG oidc: OIDC Discovery for issuer https://openid.ict.ru.ac.za
09:20:27.191Z DEBUG oidc: Handle config response and fetch JWKs from https://openid.ict.ru.ac.za/jwks
09:20:27.619Z DEBUG oidc: Fetched 2 JWKs
09:20:27.739Z DEBUG oidc: Starting session at https://openid.ict.ru.ac.za/connect/card
09:20:28.193Z DEBUG oidc: Handle signin response (text={})
09:20:28.194Z DEBUG oidc: authenticate at https://openid.ict.ru.ac.za/connect/card/callback (identifier=42)
09:20:28.643Z DEBUG oidc: Handle authenticate response (text={})
09:20:28.644Z  INFO oidc: User authenticated (identifier=42, code=d7f8fff26e0677a77e40)
09:20:28.645Z DEBUG oidc: Fetching token
```

```
09:20:29.261Z DEBUG oidc: Handle token response
09:20:29.262Z DEBUG oidc: awaiting JWK
09:20:29.264Z DEBUG oidc: verifying JWT
09:20:29.283Z DEBUG oidc: Fetching user info
09:20:29.924Z DEBUG oidc: Validating Userinfo against required fields
    userinfo: {
      "sub": "cd5c5e40-b5aa-4c56-8f46-2af4b2f4fbe2",
      "family_name": "Linklater",
      "given_name": "Greg",
      "updated_at": 1477297183230,
      "email": "g12l4025@campus.ru.ac.za",
      "department": "Computer Science",
      "staff": true
    }
09:20:29.925Z DEBUG oidc: attributes prepared for Auth Engine
09:20:29.926Z  INFO oidc: Saving cookies!
09:20:29.929Z  INFO barrier: Access control decision (decision=true, identifier=42)
```

A.3 Deny

Listing A.6: Log output for Barrier library (default format: JSON) for rule: deny


```
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"path":"./test/config/dev_deny.json","msg":"importing config","time":"2016-10-29T09:22:46.287Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"Barrier configured","time":"2016-10-29T09:22:46.288Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"Stores built","time":"2016-10-29T09:22:46.290Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"Engine built","time":"2016-10-29T09:22:46.293Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"rule":"deny","msg":"Starting access control","time":"2016-10-29T09:22:46.297Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"attributes":["/subject/staff","/subject/department","/environment/time/hours","/environment/time/minutes"],"msg":"needed attributes","time":"2016-10-29T09:22:46.298Z","v":0}
{"name":"barrier","hostname":"barrier_host","pid":8577,"level":20,"msg":"fetched store attributes","time":"2016-10-29T09:22:46.299Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"identifier":42,"msg":"Handle OIDC Auth","time":"2016-10-29T09:22:46.300Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"OIDC Discovery for issuer https://openid.ict.ru.ac.za","time":"2016-10-29T09:22:46.301Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Handle config response and fetch JWKs from https://openid.ict.ru.ac.za/jwks","time":"2016-10-29T09:22:46.697Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Fetched 2 JWKs","time":"2016-10-29T09:22:47.143Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Starting session at https://openid.ict.ru.ac.za/connect/card","time":"2016-10-29T09:22:47.148Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"text":{},"msg":"Handle signin response","time":"2016-10-29T09:22:47.541Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"identifier":42,"msg":"authenticate at https://openid.ict.ru.ac.za/connect/card/callback","time":"2016-10-29T09:22:47.541Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"text":{},"msg":"Handle authenticate response","time":"2016-10-29T09:22:47.862Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":30,"identifier":42,"code":"faaea03ebca86ead6e1e","msg":"User authenticated","time":"2016-10-29T09:22:47.862Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Fetching token","time":"2016-10-29T09:22:47.863Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Handle token response","time":"2016-10-29T09:22:48.309Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"awaiting JWK","time":"2016-10-29T09:22:48.310Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"verifying JWT","time":"2016-10-29T09:22:48.311Z","v":0}
{"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"Fetching user info","time":"2016-10-29T09:22:48.316Z","v":0}
```

```
22 {"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"userinfo":{"sub":"cd5c5e40-b5aa-4c56-8f46-2af4b2f4fbe2"
       ,"family_name":"Linklater","given_name":"Greg","updated_at":1477297183230,"email":"g1214025@campus.ru.ac.za","
       department":"Computer Science","staff":true},"msg":"Validating Userinfo against required fields","time":"
       2016-10-29T09:22:48.791Z","v":0}
23 {"name":"oidc","hostname":"barrier_host","pid":8577,"level":20,"msg":"attributes prepared for Auth Engine","time":"
       2016-10-29T09:22:48.791Z","v":0}
24 {"name":"oidc","hostname":"barrier_host","pid":8577,"level":30,"msg":"Saving cookies!","time":"2016-10-29T09:22:48.792Z
       ","v":0}
25 {"name":"barrier","hostname":"barrier_host","pid":8577,"level":30,"decision":false,"identifier":42,"msg":"Access
       control decision","time":"2016-10-29T09:22:48.793Z","v":0}
```

Listing A.7: Log output for Barrier library (pretty printed) for rule: deny

```
 1 09:16:00.074Z DEBUG barrier: importing config (path=./test/config/dev_deny.json)
 2 09:16:00.075Z DEBUG barrier: Barrier configured
 3 09:16:00.077Z DEBUG barrier: Stores built
 4 09:16:00.081Z DEBUG barrier: Engine built
 5 09:16:00.084Z DEBUG barrier: Starting access control (rule=deny)
 6 09:16:00.086Z DEBUG barrier: needed attributes
 7     attributes: [
 8       "/subject/staff",
 9       "/subject/department",
10       "/environment/time/hours",
11       "/environment/time/minutes"
12     ]
13 09:16:00.087Z DEBUG barrier: fetched store attributes
14 09:16:00.088Z DEBUG oidc: Handle OIDC Auth (identifier=42)
15 09:16:00.088Z DEBUG oidc: OIDC Discovery for issuer https://openid.ict.ru.ac.za
16 09:16:00.454Z DEBUG oidc: Handle config response and fetch JWKs from https://openid.ict.ru.ac.za/jwks
17 09:16:06.587Z DEBUG oidc: Fetched 2 JWKs
18 09:16:06.594Z DEBUG oidc: Starting session at https://openid.ict.ru.ac.za/connect/card
19 09:16:07.015Z DEBUG oidc: Handle signin response (text={})
20 09:16:07.016Z DEBUG oidc: authenticate at https://openid.ict.ru.ac.za/connect/card/callback (identifier=42)
21 09:16:07.823Z DEBUG oidc: Handle authenticate response (text={})
22 09:16:07.823Z  INFO oidc: User authenticated (identifier=42, code=8cd162d28b97b435bf6f)
23 09:16:07.823Z DEBUG oidc: Fetching token
24 09:16:08.674Z DEBUG oidc: Handle token response
25 09:16:08.674Z DEBUG oidc: awaiting JWK
26 09:16:08.674Z DEBUG oidc: verifying JWT
27 09:16:08.682Z DEBUG oidc: Fetching user info
28 09:16:09.101Z DEBUG oidc: Validating Userinfo against required fields
29     userinfo: {
30       "sub": "cd5c5e40-b5aa-4c56-8f46-2af4b2f4fbe2",
31       "family_name": "Linklater",
32       "given_name": "Greg",
33       "updated_at": 1477297183230,
34       "email": "g1214025@campus.ru.ac.za",
35       "department": "Computer Science",
36       "staff": true
37     }
38 09:16:09.102Z DEBUG oidc: attributes prepared for Auth Engine
39 09:16:09.102Z  INFO oidc: Saving cookies!
40 09:16:09.104Z  INFO barrier: Access control decision (decision=false, identifier=42)
```

Appendix B

Test Output

B.1 Authorization Engine

Listing B.1: Test output for Authorization Engine library

```
> jacl-auth-engine@0.0.1 test /opt/jacl/jacl-auth-engine
> mocha

// Note this error is intentionally logged and is tied to the test:
// "JACL Authorization Engine with invalid rule should reject the authorization request"
Error: Invalid Rule
    at new Rule (/opt/jacl/jacl-auth-engine/src/Rule.js:56:13)
    at JACL.register (/opt/jacl/jacl-auth-engine/src/JACL.js:49:15)
    at JACL.Object.keys.forEach.key (/opt/jacl/jacl-auth-engine/src/JACL.js:34:12)
    at Array.forEach (native)
    at new JACL (/opt/jacl/jacl-auth-engine/src/JACL.js:33:24)
    at Suite.describe (/opt/jacl/jacl-auth-engine/test/JACL.js:82:14)
    at Object.create (/opt/jacl/jacl-auth-engine/node_modules/mocha/lib/interfaces/common.js:114:19)
    at context.describe.context.context (/opt/jacl/jacl-auth-engine/node_modules/mocha/lib/interfaces/bdd.js:42:27)
    at Object.<anonymous> (/opt/jacl/jacl-auth-engine/test/JACL.js:24:1)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.require (module.js:497:17)
    at require (internal/module.js:20:19)
    at /opt/jacl/jacl-auth-engine/node_modules/mocha/lib/mocha.js:220:27
    at Array.forEach (native)
    at Mocha.loadFiles (/opt/jacl/jacl-auth-engine/node_modules/mocha/lib/mocha.js:217:14)
    at Mocha.run (/opt/jacl/jacl-auth-engine/node_modules/mocha/lib/mocha.js:485:10)
    at Object.<anonymous> (/opt/jacl/jacl-auth-engine/node_modules/mocha/bin/_mocha:403:18)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
    at run (bootstrap_node.js:394:7)
    at startup (bootstrap_node.js:149:9)
    at bootstrap_node.js:509:3


  JACL
```

```
    authorization engine
      with incorrect attributes
        ✓ should reject the authorization request
      with missing attributes
        ✓ should reject the authorization request
      with correct attributes
        ✓ should allow the authorization request
      with invalid rule
        ✓ should reject the authorization request
    attribute list
      ✓ subject should have two attributes
      ✓ subject attribute list should have attributes relating to the subject
      ✓ object attribute list should be empty
      ✓ environment should have two attributes
      ✓ environment attribute list should have attributes relating to the environment
      ✓ should have four attributes in total


  10 passing (10ms)
```

B.2 Store

Listing B.2: Test output for Store library

```
> jacl-store@0.0.1 test /opt/jacl/jacl-store
> mocha



  Store
    data directory
      ✓ should glob all files in the data store directory
      ✓ should load and assign attributes and generators to the cache
    getAttribute
      ✓ should fetch attributes given a JSON Pointer string
      ✓ should fetch attributes given an array of JSON Pointer strings
      ✓ should evaluate attributes defined in terms of a function
      ✓ should ignore parameters for attributes defined in terms of a function
      ✓ should return null for invalid JSON Pointer strings
      ✓ should return null if requested attribute is not present
      alias (get)
        ✓ should fetch attributes given a JSON Pointer string
        ✓ should fetch attributes given an array of JSON Pointer strings
        ✓ should evaluate attributes defined in terms of a function
        ✓ should ignore parameters for attributes defined in terms of a function
        ✓ should return null for invalid JSON Pointer strings
        ✓ should return null if requested attribute is not present
    config
      without specifying a JSON Pointer to the config
        ✓ should fetch the default config
      specifying a JSON Pointer to the config
        ✓ should fetch the specified config
      specifying a JSON Pointer to an invalid config
        ✓ should return null
    rule
      ✓ should throw if no rule name is supplied
      ✓ should return null if the rule does not exist
      ✓ should return the rule object descriptor if it exists


  20 passing (42ms)
```

B.3 Barrier

Listing B.3: Test output for Barrier library

```
> jacl-barrier@0.0.1 test /opt/jacl/jacl-barrier
> NODE_TLS_REJECT_UNAUTHORIZED=0 mocha



  Barrier
    Integration
      positive rule
        ✓ should allow access (5009ms)
      negative rule
        ✓ should deny access (5218ms)
    log
      ✓ should return the logger singleton
    constructor
      ✓ should throw if no descriptor is provided
      ✓ should assign a Store
      ✓ should assign an AuthEngine
      string (config path) parameter
        ✓ should log to debug
        ✓ should throw if file doesn't exist
        ✓ should assign it to the barrier instance if successful
      Config instance parameter
        ✓ should assign it to the barrier instance
      Config descriptor object parameter
        ✓ should assign it to the barrier instance
    makeDecision
      ✓ should defer to the auth engine
      ✓ should log to info
      ✓ should return a boolean
    authenticate
      ✓ should defer to the OIDC handler
      ✓ should pass the barrier and the provider config to the OIDC handler
    enforce
      ✓ should log to debug
      ✓ should not log to info
      ✓ should log to warn if passed an invalid identifier
      ✓ should reject if passed an invalid identifier
      ✓ should request the attribute list from the auth engine
      ✓ should call authenticate
      ✓ should return a Promise
      ✓ should resolve to a boolean

  Config
    schema
      ✓ should be an instance of JSONSchema
      ✓ should return ConfigSchema
    serialize
      with a relative path
        that is valid
          ✓ should log to info
          ✓ should not log to error
          ✓ should return true
          ✓ should call writeFileSync
        that is invalid
          ✓ should log to error
          ✓ should not log to info
          ✓ should return false
          ✓ should call writeFileSync
          ✓ writeFileSync should throw
      with a absolute path
        that is valid
          ✓ should log to info
          ✓ should not log to error
          ✓ should return true
          ✓ should call writeFileSync
        that is invalid
          ✓ should log to error
          ✓ should not log to info
```

```
69          ✓ should return false
70          ✓ should call writeFileSync
71          ✓ writeFileSync should throw
72      static deserialize
73        with a relative path
74          that is valid
75            ✓ should return a Config object
76            ✓ should log to info
77            ✓ should not log to error
78            ✓ should call readFileSync
79          that is invalid
80            ✓ should not log to info
81            ✓ should log to error
82            ✓ should return null
83            ✓ should call readFileSync
84            ✓ readFileSync should throw
85        with an absolute path
86          that is valid
87            ✓ should return a Config object
88            ✓ should log to info
89            ✓ should not log to error
90            ✓ should call readFileSync
91          that is invalid
92            ✓ should not log to info
93            ✓ should log to error
94            ✓ should return null
95            ✓ should call readFileSync
96            ✓ readFileSync should throw
97
98    Logger
99      init
100       ✓ should call initDir
101     loggersSerializer
102       ✓ should produce the expected result
103       ✓ should not alter the loggers container
104     initDir
105       ✓ should create the directory if it doesn't exist
106       ✓ should create the directory if a file with the same name exists
107       ✓ shouldn't create the directory if it already exists
108     defaultLogLevel
109       ✓ should return INFO if ENV is production
110       ✓ should return DEBUG if ENV is anything else
111     defaultLogger
112       ✓ should call defaultLogLevel
113       ✓ should be dynamically named
114       ✓ should use the default log level
115       ✓ should produce the expected output
116     getLogger
117       ✓ should call getLoggerByString when passed a string
118       ✓ should call getLoggerByDescriptor when passed an object
119       - should get the default logger when passed undefined
120       - should log to trace when passed undefined
121       - should log to trace when passed nothing
122       ✓ should throw when passed null
123     getLoggerByString
124       ✓ should call createNamespacedLogger if passed a namespaced name
125       ✓ should call getOrCreateLogger if passed a non-namespaced name
126     createNamespacedLogger
127       - should log on trace
128       ✓ should create a logger for each namespace
129       - should return the final logger
130     getLoggerByDescriptor
131       - should log on warn
132       ✓ should use defaultLogLevel if level is not supplied
133       - should log to error if name is not supplied
134       ✓ should add two default streams
135       ✓ should add an additional specified stream
136       ✓ should add additional specified streams
137       ✓ should return a newly created logger
138
139   OIDCHandler
140     static handle
141       ✓ should call discover
142       ✓ should call getJSONResponse x4
143       ✓ should call handleDiscoveryResponse
```

```
144       ✓ should call handleJWKSetResponse
145       ✓ should call signin
146       ✓ should call handleSignInResponse
147       ✓ should call authenticate
148       ✓ should call handleAuthenticationResponse
149       ✓ should call getJWT
150       ✓ should call handleJWTResponse
151       ✓ should call userinfo
152       ✓ should call verifyAttributes
153       ✓ should call persistCookies
154       ✓ should return a Promise
155       - should log to debug
156     static handle where an inner method rejects
157       ✓ should call discover
158       ✓ should call getJSONResponse x4
159       ✓ should call handleDiscoveryResponse
160       ✓ should call handleJWKSetResponse
161       ✓ should call signin
162       ✓ should call handleSignInResponse
163       ✓ should call authenticate
164       ✓ should call handleAuthenticationResponse
165       ✓ should call getJWT
166       ✓ should not call handleJWTResponse
167       ✓ should not call userinfo
168       ✓ should not call verifyAttributes
169       ✓ should not call persistCookies
170       ✓ should return a Promise
171       - should log to debug
172       - should log to error
173     method
174       discover
175         - should return a Promise
176         - should fetch ${issuer}/.well-known/openid-configuration
177         - should log to debug
178         - should resolve to an array
179         - result should contain a handler reference
180         - result should contain a fetch response
181         - should reject if the fetch fails
182       getJSONResponse
183         - should return a Promise
184         - should log to trace
185         - should resolve to an array
186         - result should contain a handler reference
187         - result should contain an object
188       handleDiscoveryResponse
189         - should log to debug
190         - should fetch jwks
191         - should return a Promise
192         - should resolve to an array
193         - result should contain a handler reference
194         - result should contain a fetch response
195         - should reject if the fetch fails
196       handleJWKSetResponse
197         - should log to debug
198         - should return a Promise
199         - should to a handler reference
200         - should reject jwks is empty
201       signin
202         - should log to debug
203         - should return a Promise
204         - should fetch signin
205         - should resolve to an array
206         - result should contain a handler reference
207         - result should contain a fetch response
208         - should reject if the fetch fails
209       handleDiscoveryResponse
210         - should log to debug
211         - should return a Promise
212         - should to a handler reference
213       authenticate
214         - should log to debug
215         - should return a Promise
216         - should fetch authenticate
217         - should resolve to an array
218         - result should contain a handler reference
```

```
        - result should contain a fetch response
        - should reject if the fetch fails
      handleAuthenticationResponse
        - should log to debug
        - should log to info
        - should return a Promise
        - should resolve to an array
        - result should contain a handler reference
        - result should contain the authorization code
        - should reject if authorization code not present
      getJWT
        - should log to debug
        - should return a Promise
        - should fetch access token jwt
        - should fetch using POST
        - should fetch with authorization header
        - should resolve to an array
        - result should contain a handler reference
        - result should contain a fetch response
        - should reject if the fetch fails
      handleJWTResponse
        - should log to debug
        - should return a Promise
        - should fetch decode access token jwt
        - should call verify on JWT
        - should resolve to an array
        - result should contain a handler reference
        - should reject if the fetch fails
        result should contain an access token
          - that is verified
          - that is decoded
      userinfo
        - should log to debug
        - should return a Promise
        - should fetch userinfo jwt
        - should fetch with authorization header
        - should resolve to an array
        - result should contain a handler reference
        - result should contain a fetch response
        - should reject if the fetch fails
      verifyAttributes
        - should log to debug
        - should return a Promise
        - should assign userinfo to the handler
        - should resolve to an array
        - result should contain a handler reference
        - result should contain userinfo
        - result should contain relevenat subject attributes object
        - should reject if missing any required attributes
      persistCookies
        - should log to info
        - should return a Promise
        - should evaluate to the same as what was passed in


  113 passing (11s)
  94 pending
```

B.4 Test Coverage

Figure B.1: Test Coverage Reports for: (a) Authorization Engine, (b) Store, (c) Barrier