University of Tennessee, Knoxville
Trace: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

5-2012

A Framework for Federated Two-Factor Authentication Enabling Cost-Effective Secure Access to Distributed Cyberinfrastructure

Matthew Allan Ezell
matt@mattezell.com

Recommended Citation

Ezell, Matthew Allan, "A Framework for Federated Two-Factor Authentication Enabling Cost-Effective Secure Access to Distributed Cyberinfrastructure. " Master's Thesis, University of Tennessee, 2012.
http://trace.tennessee.edu/utk_gradthes/1151

This Thesis is brought to you for free and open access by the Graduate School at Trace: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of Trace: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Matthew Allan Ezell entitled "A Framework for Federated Two-Factor Authentication Enabling Cost-Effective Secure Access to Distributed Cyberinfrastructure." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Jinyuan Sun, Mark R. Fahey

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

A Framework for Federated Two-Factor Authentication Enabling Cost-Effective Secure Access to Distributed Cyberinfrastructure

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Matthew Allan Ezell

May 2012

© by Matthew Allan Ezell, 2012

All Rights Reserved.

This work is dedicated to Dianne Bull.

Thank you for your love and support.

Acknowledgements

Many individuals have had large influences on my work, and I would like to thank them for their contributions.

First, Dr. Gregory D. Peterson has been instrumental in assisting me in preparing this thesis. I thank him for his advice and support.

I would also like to acknowledge my other committee members, Dr. Jinyuan "Stella" Sun and Dr. Mark R. Fahey. Their feedback has improved this thesis.

I am grateful to Patricia Kovatch for initially interesting me in this topic.

The National Science Foundation XSEDE Security Operations group has provided valuable feedback.

Finally, I would like to thank my family and friends for their love and support.

Abstract

As cyber attacks become increasingly sophisticated, the security measures used to mitigate the risks must also increase in sophistication. One time password (OTP) systems provide strong authentication because security credentials are not reusable, thus thwarting credential replay attacks. The credential changes regularly, making brute-force attacks significantly more difficult. In high performance computing, end users may require access to resources housed at several different service provider locations. The ability to share a strong token between multiple computing resources reduces cost and complexity.

The National Science Foundation (NSF) Extreme Science and Engineering Discovery Environment (XSEDE) provides access to digital resources, including supercomputers, data resources, and software tools. XSEDE will offer centralized strong authentication for services amongst service providers that leverage their own user databases and security profiles. This work implements a scalable framework built on standards to provide federated secure access to distributed cyberinfrastructure.

Contents

1 Introduction 1

2 Technology and Literature Review 3

2.1 Extreme Science and Engineering Discovery Environment 3

2.2 Pluggable Authentication Modules 4

2.3 Remote Authentication Dial In User Service 5

2.3.1 Protocol 5

2.3.2 Packet Format 6

2.3.3 Shared secrets 8

2.3.4 Realms and Proxying 8

2.3.5 Security Extensions for RADIUS 9

2.4 Security Assertion Markup Language 10

2.5 One Time Password Technology 10

2.5.1 RSA SecurID 11

2.5.2 The Initiative for Open Authentication 12

2.6 Similar Authentication Federation Projects 16

2.6.1 eduroam 16
2.6.2 Project Moonshot 17
2.6.3 Energy Sciences Network RADIUS Authentication Fabric . . . 18
2.7 Background Summary 19

3 Authentication Router Design Approach 20
3.1 Overview 20
3.2 Authentication Router 22
3.3 Username Mapping Service 24
3.4 Password Service 24
3.5 OTP Service 25
3.6 Client Support 25

4 Authentication Router Implementation 27
4.1 XSEDE Central Database 27
4.2 RADIUS Framework 29
4.2.1 Modules 31
4.2.2 Virtual and Home Servers 32
4.2.3 Logging 32
4.3 Managing Clients 33
4.4 XSEDE FreeRADIUS Module 34
4.5 Operational Issues 36
4.6 Cost 38

5 Authentication Router Testing 40

5.1 Test Setup . . . 40

5.2 Test Tools . . . 41

5.2.1 Radtest . . . 41

5.2.2 Radclient . . . 42

5.2.3 Pamtester . . . 42

5.3 Test Plan . . . 43

5.4 Problems Experienced . . . 44

5.5 Test Results . . . 45

5.5.1 Authenticating Each User Once . . . 45

5.5.2 Authenticating One User Many Times . . . 45

5.6 Testing Summary . . . 48

6 Conclusions 50

6.1 Thesis Summary . . . 50

6.2 Future Work . . . 51

Bibliography 53

A FreeRADIUS Configuration Listing 59

A.1 radiusd.conf . . . 59

A.2 Sample clients.conf . . . 61

B Source Code Listing 62

B.1 Main Integration Script 62

B.2 Common Settings and Functions 65

B.3 RADIUS Clients Functions 68

B.4 RADIUS Users Functions 71

B.5 Multi-Threaded Caching Functions 74

Vita **76**

List of Tables

2.1 RADIUS Codes 6

2.2 Variables for the HOTP Algorithm 13

2.3 Variables for the TOTP Algorithm 14

4.1 FreeRADIUS Logging Codes 33

4.2 XSEDE Module Command Line Arguments 35

5.1 Test Results Authenticating Each User Once 46

5.2 Test Results Authenticating One User Many Times 47

List of Figures

2.1 RADIUS Packet 7

3.1 System Overview 21

3.2 RADIUS Hierarchy 23

4.1 Database Schema 30

5.1 Test Results Authenticating Each User Once 48

5.2 Test Results Authenticating One User Many Times 49

Chapter 1

Introduction

To keep pace with evolving cyber attacks, security professionals must deploy new innovative techniques to protect digital resources. One method for protecting resources that has been gaining popularity is the use of strong authentication technologies, such as two-factor authentication through the use of one time password devices. Hazlewood et al. [6] describes a successful deployment of one time password devices at the University of Tennessee's National Institute for Computational Sciences.

The National Science Foundation's Extreme Science and Engineering Discovery Environment (XSEDE [24]) consortium has plans to leverage existing investments in strong authentication and make strategic centralized investments in security infrastructure to optimally serve the user community. A set of services and software will be assembled and developed to enable federated two-factor authentication. The solution must be scalable, reliable, and secure while interoperating with existing

XSEDE resources and cyberinfrastructure. All XSEDE users will be issued a single credential that will be usable across all XSEDE services. This provides cost savings by reducing credential management overhead and simplifies the user experience by removing the need to manage multiple passwords or install special software on the client.

This thesis describes an "Authentication Router" system that was developed to take end-user authentication requests from resources at service providers and map them to their unique authentication token. The router authenticates to the backend one-time password (OTP) providers and returns the result to the original resource. The Authentication Router provides flexibility and the ability to enforce security policies across multiple authentication technologies. To the author's knowledge, a system like this has never been put into production. This project has the potential to greatly improve the security of the XSEDE cyberinfrastructure while minimizing costs.

Chapter 2 of this thesis provides the background information necessary for fully understanding the work presented here. Chapter 3 describes the approach taken in designing the federated authentication system. Specific implementation details are given in Chapter 4, and information about testing is available in Chapter 5. Finally, Chapter 6 summarizes and concludes the discussed research.

Chapter 2

Technology and Literature Review

2.1 Extreme Science and Engineering Discovery Environment

The National Science Foundation (NSF [15]) funded the Extreme Science and Engineering Discovery Environment (XSEDE [24]) in 2011 as a replacement for the TeraGrid. XSEDE is a collection of integrated advanced digital resources designed to allow scientists and researchers to collaborate and share scientific information. As of 2012, XSEDE provides access to sixteen supercomputers and visualization resources distributed throughout the United States. The XSEDE project includes seventeen partner institutions with expertise in high performance computing, software development, and scientific discovery. The Authentication Router service described in this thesis is intended to be deployed within XSEDE's infrastructure.

2.2 Pluggable Authentication Modules

Authentication is the process of confirming one's identity, and it is a central component of computer security. Many different forms of authentication technology are deployed in the field, and their complexity and effectiveness have a broad range. This poses a challenge for application developers; how can software be designed to support a seemingly endless collection of authentication methods? How do system administrators ensure that all of their applications support consistent authentication technologies? The Pluggable Authentication Modules (PAM [21]) framework abstracts these details and provides a common application programming interface (API) for authentication. Application developers simply need to implement the PAM API and authentication technologies can be "plugged in" to the application. System administrators can configure PAM from a central location and all PAM-enabled applications can inherit this configuration.

The authentication logic of the PAM framework is implemented as a set of "stackable" modules. The PAM configuration lists the modules as well as the mode for each module. Each module is queried in the order it appears in the configuration. All "required" modules must succeed for the overall authentication to be considered successful. If a required module fails, the other modules are still queried to avoid revealing information about where the failure occurred. "Optional" modules may pass or fail without affecting the overall authentication process. If a module listed as "sufficient" succeeds, PAM will stop processing and immediately return success. A

failure of a sufficient module is treated like a failure of an optional module. The PAM framework exposes four interfaces that modules should implement: authentication management, account management, session management, and password management. PAM enables very complex authentication scenarios on a single system.

2.3 Remote Authentication Dial In User Service

The Remote Authentication Dial In User Service (RADIUS) protocol is a common industry standard that handles Authentication, Authorization, and Accounting (AAA) functions for network services. The standard is described in RFC2865 [18] and described briefly below.

2.3.1 Protocol

RADIUS is a client/server protocol that uses the user datagram protocol (UDP) as a transport mechanism. Packet delivery is unreliable because there is no guarantee of arrival, so clients and servers must implement retransmission logic in case of network faults. UDP was chosen for several technical reasons, including simplicity of implementation and better control of retransmissions in case of network failure. A RADIUS client is often called a Network Access Server (NAS). RADIUS servers listen on port 1812 for RADIUS Authentication requests and port 1813 for RADIUS Accounting requests.

Table 2.1: RADIUS Codes

Code	Assignment
1	Access-Request
2	Access-Accept
3	Access-Reject
4	Accounting-Request
5	Accounting-Response
11	Access-Challenge
12	Status-Server (experimental)
13	Status-Client (experimental)
255	Reserved

When an end user desires to access a resource, he will provide authentication credentials to the NAS. The NAS will take this data and construct an Access-Request packet to send to the RADIUS server. If no response is received in a specified amount of time, the NAS can either retransmit the request or try sending the request to an alternate server. When the RADIUS server receives the Access-Request packet, it must first authenticate the client. It then authenticates the contents of the packet by querying authentication databases or proxying the request to another RADIUS server. The server must then reply to the client with Access-Accept or Access-Reject. The client uses this response to determine if access should be granted or denied.

2.3.2 Packet Format

RADIUS packets are stored in the data field of the UDP datagram. The layout of a RADIUS packet is shown in Figure 2.1. The data is transmitted left-to-right. The packet consists of five fields.

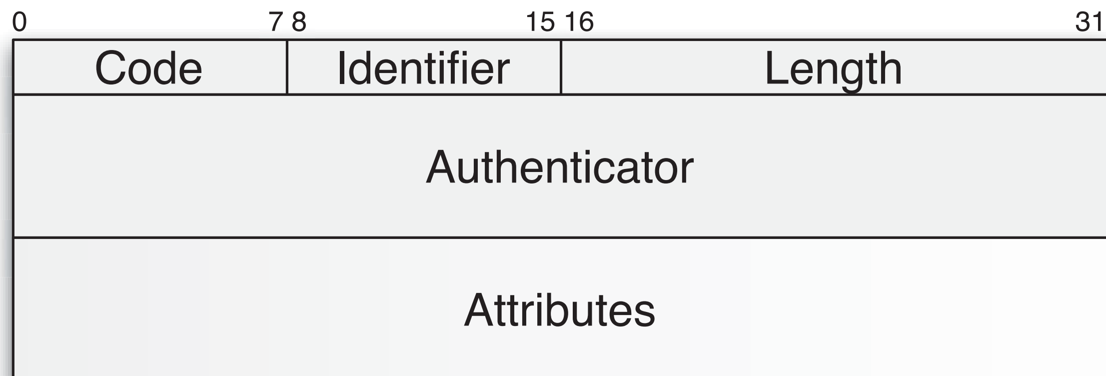

Figure 2.1: RADIUS Packet

Code

The code field occupies the first eight bits of the packet. Possible codes are listed in Table 2.1. If an invalid code is specified, the packet is simply dropped.

Identifier

The identifier is an 8-bit number that is used to match requests and replies. Packets can be recognized as duplicates if they contain identical IP addresses, ports, and identifiers within a short timespan.

Length

Length is a 16-bit field that describes how many bits the entire packet contains. The packet can be a maximum of 4096 bits.

Authenticator

The authenticator field is a 128-bit value used to authenticate the reply from a RADIUS server. For a request packet, the authenticator should be a unique

random value. Protocol security mandates that an authenticator should not be reused because replay attacks are possible. Additionally, the message authenticator is used to obfuscate passwords in Access-Request messages. A response packet contains an authenticator that is a hash of several fields.

Attributes

A RADIUS request can contain zero or more attributes. Some RADIUS codes have both required and optional attributes.

2.3.3 Shared secrets

Every RADIUS client/server pair must share a secret password used to sign messages and obfuscate user passwords sent in RADIUS packets. The shared secret is never sent over the network, and it should be distributed in an out-of-band secure manner. The Internet Engineering Task Force (IETF) Request for Comments (RFC) memorandum recommends at least 16 characters for the secret, but additional length provides better security. Uppercase and lowercase letters, numbers, and symbols should all be used. If a RADIUS request originates from a client that does not have a shared secret configured, or if the shared secret is incorrect, the packed should be dropped.

2.3.4 Realms and Proxying

The RADIUS protocol supports the idea of roaming using "realms." A realm typically corresponds to an administrative domain, defining a user base and set of

policies. Changing from one realm to another may indicate that the request should be authenticated in a different way. A RADIUS server may accept requests from clients that it cannot verify itself. It would then proxy the request to an authoritative server for the given realm and then resend the response to the client. This allows for complicated setups to support heterogeneous user bases, identity federation, and network sharing.

2.3.5 Security Extensions for RADIUS

Additional security for RADIUS has been proposed by the RADIUS Extensions Working Group in an experimental internet draft entitled "Transport Layer Security (TLS) encryption for RADIUS" [23]. This protocol extension, referred to as RADSEC, attempts to overcome two major criticisms of the original RADIUS protocol. The first issue is the use of the unreliable UDP transport for packet transmission. RADSEC instead transmits RADIUS datagrams over the transmission control protocol (TCP). A single port, 2083, is used for both authentication and accounting requests. The second issue concerns the integrity and privacy of the RADIUS datagram. Only certain fields are encrypted, and the encryption is based on a hashing algorithm with known weaknesses. RADSEC overcomes this limitation by support Transport Layer Security, or TLS.

Although RADSEC is just an experimental draft, it does have significant use in the community. The *eduroam* consortium, described later in Section 2.6.1, uses RADIUS over TLS in its production environment to improve security across untrusted internet

connections. Additionally, the commercial “Radiator” RADIUS server and the open source “radsecproxy” implement the RADSEC protocol [23].

2.4 Security Assertion Markup Language

Security Assertion Markup Language, or SAML, is a framework for authentication and authorization built upon the Extensible Markup Language (XML). It aims to allow the exchange of authentication and authorization details across security and platform boundaries. SAML messages, called assertions, contain identity information related to an authentication that has already taken place [14]. SAML itself does not provide trust; the assertions must be signed or verified by another layer in the software stack. Additionally, few non-web-based applications, like the bulk of those used in XSEDE, support the use of SAML.

2.5 One Time Password Technology

Traditional passwords are static strings of varying length and complexity. If a more secure password is required, the only option is to increase the length and add additional characters to the character set. Additionally, static passwords are vulnerable to interception at several layers. A hardware or software keylogger may obtain the password as the user enters it into his terminal. A network sniffer or man-in-the-middle attack may intercept the password as it moves across the network. Faulty or compromised server software may also allow the password to be obtained by

unauthorized users. Nothing exists to prevent unauthorized access once the attacker has access to the static password.

RFC 2289 [5] describes the design for one system that uses a One Time Password (OTP). The key difference here is that instead of a single static password, the user provides a different password each time. If the authentication server is configured to only accept each password once, this effectively eliminates replay attacks and password sniffing vulnerabilities. OTP uses a secret passphrase along with a sequence number that is run through a secure one-way hashing function to generate a set of single-use passwords. Several commercial and non-commercial solutions exist to implement two-factor authentication through the use of OTP technology.

2.5.1 RSA SecurID

RSA Laboratories, the security division of EMC Corporation, offers a commercial product called RSA SecurID. SecurID uses a proprietary algorithm to calculate OTP values. Several form factors are available depending on specific needs [20].

In March of 2011, RSA Laboratories was broken into as part of an advanced persistent threat. Some of RSA's proprietary information related to their SecurID product was exfiltrated [19]. RSA subsequently offered free token replacement for their clients. Despite this, RSA's product is still the most widely deployed form of two-factor authentication. For this reason, it is important for federated OTP solutions to be interoperable with RSA SecurID.

2.5.2 The Initiative for Open Authentication

The Initiative for Open Authentication (OATH) is a collaboration between several security companies that aims to promote the use of universal strong authentication for users and devices [16]. OATH members believe that strong authentication is central to open, federated networks that can be safely and securely used. To realize the goal of more pervasive strong authentication, both complexity and cost must be lowered to provide a low barrier to entry. Thus, an open and royalty-free specification for strong authentication is vital to realizing the goals of OATH's members. To date, OATH has proposed several open algorithms including HOTP, TOTP, and OCRA (defined and described later).

Hash-based Message Authentication Code One Time Password Algorithm

Hash-based Message Authentication Code OTP (HOTP) is an open OTP algorithm intended to be used for two-factor authentication that is described in M'Raihi et al. [9]. It was designed to be economical, easy to implement, and easy to use. Thus, it requires minimal processing power, battery capabilities, and display screen space. The algorithm requires OTP values of at minimum six digits, although longer sequences are also supported.

The HOTP algorithm is based on a shared secret as well as a monotonically increasing sequence value. Equation 2.1 shows the basics of the HOTP operation using the variables shown in Table 2.2. The secret key and counter are input to the hash-based message authentication code with secure hash algorithm (HMAC-SHA-1)

Table 2.2: Variables for the HOTP Algorithm

Variable	Description
K	Secret Key
C	Counter
HMAC()	A SHA-1 Hash-based Message Authentication Code Function
Truncate()	A function that selects 4 bytes from the result
&	A bitwise AND function
%	The modulus operator
d	The number of digits in the OTP

defined in [8]. The 160-bit output is truncated and the leftmost bit is masked out to avoid confusion for signed versus unsigned calculations of the modulus function. The value is then converted into a numeric string with the desired number of digits.

$$\text{HOTP}(K, C, d) = (\text{Truncate}(\text{HMAC}(K, C))\&\text{0x7FFFFFFF})\%10^{d} \qquad (2.1)$$

For proper authentication, the client token and the validation server must share the same counter value. Every time the client token generates an OTP value, it must increment its counter. The validation server will increment its counter with every successful authentication. During a successful authentication process, both the client token and the validation server will calculate the same value. If their counters have diverged due to accidental OTP generation or brute-force attacks, the validation server will typically initiate a resynchronization protocol to update the server's counter to the same value present on the client token.

Table 2.3: Variables for the TOTP Algorithm

Variable	Description
HOTP	The HOTP algorithm
T	The time step sequence
T_0	The Unix epoch
T_c	The current time
X	The time step
floor	Function to round down

Time-Based One Time Password Algorithm

Time-Based One Time Password (TOTP) is a time-based extension of the HOTP algorithm described earlier. The RFC implementation [10] uses a time-based counter instead of an event-based counter. Both the client token and the validation server must know the current Unix time (the number of seconds since midnight on January 1, 1970). Equation 2.2 and Equation 2.3 show the basics of the TOTP operation using the variables shown in Table 2.3.

$$\text{TOTP}(K, d) = \text{HOTP}(K, T, d) \tag{2.2}$$

$$T = \text{floor}\left(\frac{T_c - T_0}{X}\right) \tag{2.3}$$

TOTP only generates a single OTP during every time step. Choosing an appropriate time step size is important both for usability and security reasons. End users may become frustrated if they have to wait a long time between being able to generate OTP values. Conversely, values that change too frequently may be difficult

to use. A successful brute force attack must be able to complete within the time window, which is highly unlikely.

Because the counter is time-based, it is less likely for the client token and validation server to get out of sync. However, network latency or inaccurate drifting clocks may cause the client token and validation server to believe the current time is different. Thus, sliding time windows and resynchronization protocols must be implemented at the validation server.

OATH Challenge-Response Algorithm

The OATH Challenge-Response Algorithm (OCRA [11]) is a generalization of the HOTP algorithm. Instead of a shared counter value, a challenge is issued from the validation server to the user. The user would then input this challenge into the token to receive the OTP value. Equation 2.4 shows the basics of the algorithm.

$$\text{OCRA} = \text{CryptoFunction}(K, DataInput) \tag{2.4}$$

As with the other OATH algorithms, K is a key shared between the user token and the validation server. The *DataInput* is a concatenation of several input values. The *CryptoFunction* is a variation of HOTP, typically a six-digit HMAC-SHA1 function.

2.6 Similar Authentication Federation Projects

2.6.1 eduroam

The *eduroam* project was originally created in Europe to develop an infrastructure to allow roaming access to national research and educational networks. It provides world-wide roaming access to wireless networks through participating research and education institutions. It uses 801.2X for authentication and RADIUS servers arranged in a hierarchal manner. Users authenticate with a user name in the form of 'user@InstitutionA.edu'. If an authentication request for a user from InstitutionA arrives at the servers for InstitutionB, it will proxy the request to the national RADIUS proxy server. The national proxy server forwards the request to the home institution where the password is checked for accuracy. The result is returned to the original endpoint and access is granted or denied. Additional details are available in Florio and Wierenga [3].

eduroam is an elegant solution for the problem the designers were attempting to solve: provide network access to roaming users. As such, it is limited because it only authorizes network access based on authentication performed at a user's home institution to prove affiliation; it provides no identity information. The institution granting access to a guest may not even know the name of the user accessing the network. To provide access to XSEDE cyberinfrastructure, each end user must be mapped to a specific user account that is connected to an allocation. Thus, the ideas provided by *eduroam* are valuable, but they must be extended.

2.6.2 Project Moonshot

Project Moonshot is an effort spearheaded by JANET(UK), the United Kingdom's research and education network consortium that runs the UK Access Federation. JANET(UK) currently provides a web-based SAML single-sign-on federation service for its users. Project Moonshot aims to extend that service and standardize federated identity among non-web applications. Their approach builds on the principles of *eduroam* with a RADIUS fabric for authentication, authorization, and accounting. Moonshot extends this by adding the Security Assertion Markup Language (SAML), the Generic Security Services Application Program Interface (GSS-API), and the Extensible Authentication Protocol (EAP). Many more details about the Project Moonshot architecture are available in Howlett and Hartman [7].

The project is still fairly new and has several technological challenges before widespread use would be feasible. Project Moonshot's architecture requires source-code changes for many of the client and server applications to interoperate with the authentication mechanisms. Specifically, the clients would need to support the GSS-API; even those that already support the GSS-API might need modification to support Moonshot [7]. At this stage, significant client modification is not feasible for the XSEDE cyberinfrastructure.

2.6.3 Energy Sciences Network RADIUS Authentication Fabric

Muruganantham et al. [13] describe a RADIUS Authentication Fabric (RAF) prototype developed by ESnet, the Energy Sciences Network that supports Department of Energy national labs. ESnet proposed the RAF reduce vulnerability to external hacking attempts by spreading the use of OTP technology. Several laboratories had already implemented non-interoperable OTP systems, so a system to allow federated use of existing tokens was desired.

The proposed architecture stationed RADIUS servers at each site and also centrally. The "edge" RADIUS servers were able to service local requests and forward remote requests to the central servers for proper routing. Additional topologies are possible, where every edge RADIUS server talks directly with the edge RADIUS servers at the other sites. This removes the central routers, but makes configuration less straightforward.

One issue that the project group encountered was poor client support for realm-based usernames. Similar to *eduroam*, the RAF supports usernames in the form of 'user@InstitutionA.gov'. Client applications, like the OpenSSH web server, cannot handle usernames of this form.

Although this project seemed quite mature and was also mentioned in another paper about ESnet's authentication services [12], the website listed (`www.es.net/raf`) is no longer accessible. Attempts to contact the authors were unsuccessful, although

communication with staff at Oak Ridge National Laboratory led the author to believe that the project was never brought into production.

2.7 Background Summary

This chapter described several existing technologies related to authentication, OTP systems, security, and federated identity management. While these technologies are valuable solutions for the problems they were designed to solve, none alone are capable of solving XSEDE's federated two-factor authentication needs. Thus, the "Authentication Router" system in this thesis was designed to manage strong authentication requests for XSEDE.

Chapter 3

Authentication Router Design Approach

3.1 Overview

This project is designed to leverage existing software where possible to take advantage of existing security and bug testing. Custom software has been developed to implement the username mapping service and other routing services as necessary. This technology is intended to be deployed throughout the XSEDE network, bringing strong authentication to participating cyberinfrastructure resources. All technologies developed in the project are intended to be open and shared with other similar consortia, encouraging strong authentication across the internet.

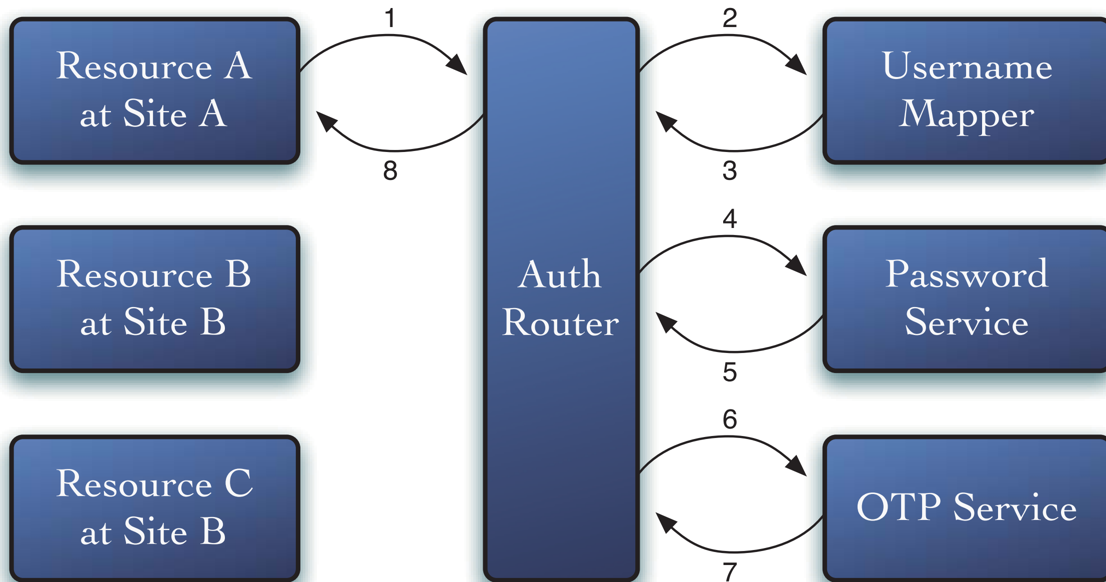

Figure 3.1: System Overview

Figure 3.1 shows the general framework for the interactions between the Authentication Router and the clients, username mapping service, password service, and OTP service. The basic request flow can be described with eight steps for authentication.

1. An authentication request originates at a resource. The original username is sent to the Authentication Router along with the password and OTP code.

2. The original username is sent to the username mapper.

3. The XSEDE canonical username is returned to the Authentication Router along with details about the password service and OTP service to which the user has been assigned.

4. The XSEDE canonical username and supplied password is sent to the password service. For most users in XSEDE, this will likely be the XSEDE Kerberos realm used to authenticate to the XSEDE User Portal.

5. The password service returns success or failure for the authentication request.

6. The XSEDE canonical username and supplied OTP code is sent to an OTP service.

7. The OTP service returns success or failure for the authentication request.

8. A success or failure message is returned to the original client based on the results of the previous steps.

3.2 Authentication Router

The Authentication Router service is the central hub that integrates the individual pieces of this project. Its job is to accept authentication requests from client endpoints and coordinate the rest of the authentication process. It must integrate with a username mapping service, a password service, and an OTP service. These individual pieces are meant to be pluggable and interchangeable, allowing both service providers and end users to take advantage of different authentication technologies.

The RADIUS protocol described in Section 2.3 appears to be best-suited to handle the requirements of the Authentication Router. Many existing authentication technologies are already interoperable with RADIUS, making integration more

straightforward. Additionally, the built-in ability to proxy and forward requests maps well to the requirements of this project.

Service providers have multiple choices for integrating with the Authentication Routers. Figure 3.2 shows two options. A site may choose to deploy local RADIUS servers if it desires to enforce local policy or have more control over the configuration. Local clients would talk to the local RADIUS servers that would proxy the authentication requests to the central RADIUS servers. For simplicity or cost savings, other sites may not want to deploy local RADIUS servers. The clients can talk directly to the central RADIUS servers. RADIUS software typically allows multiple servers to be defined, enabling high-availability setups.

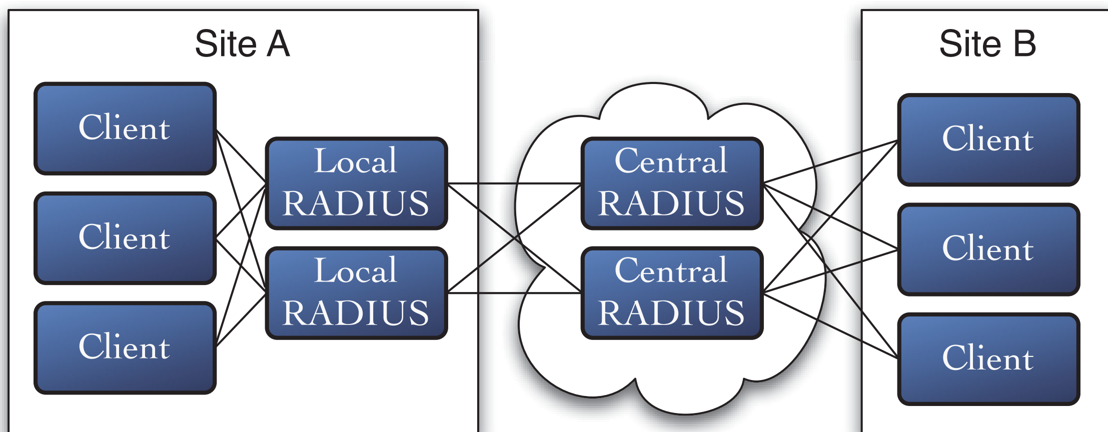

Figure 3.2: RADIUS Hierarchy

3.3 Username Mapping Service

Every service provider maintains its own unique user accounting systems. They integrate with the XSEDE central user accounting system through a protocol called Account Management Information Exchange (AMIE). When a new account is created in XSEDE, an AMIE packet is distributed to all the appropriate XSEDE service providers that contains all the information necessary to create the account locally. Username suggestions are transmitted, and the service providers will choose a username that does not already exist in their local accounting database. The central XSEDE accounting system does not know what "local" accounts exist at each site, and thus cannot ensure that username conflicts will not emerge. It is possible, and even likely, that a user will end up with different usernames at each service provider. Once the account is created, an AMIE packet is generated at the service provider and sent back to the XSEDE central database.

The purpose of the username mapping service is to translate the endpoint usernames into the canonical XSEDE username and then into the authentication target username (if a non-central token is used). The XSEDE central database contains all the information required to do the lookup.

3.4 Password Service

Passwords are typically used as the authentication credential for single-factor authentication, but passwords can also be used as the "first factor" in two-factor

authentication. All XSEDE users have a password credential stored in the XSEDE Kerberos Realm that is used to login to the XSEDE user portal. It makes sense to leverage this password as part of the strong authentication project, assuming the OTP service does not already require its own static password element.

3.5 OTP Service

The OTP service is meant to be pluggable, to allow integration with existing and future OTP service deployments. Existing systems include the NICS RSA servers and the Blue Waters OTP servers. Future work will deploy a central XSEDE OTP server. Additionally, service providers may choose to deploy their own OTP service that can be integrated into the Authentication Router. Each OTP authentication system may need custom interface code that allows it to integrate with the Authentication Router, although systems that implement the RADIUS protocol or support PAM would be trivial to integrate.

3.6 Client Support

Since the Authentication Router is based on the standard RADIUS protocol, many clients already exist that should seamlessly interoperate without any modification. Server programs such as the secure shell daemon (sshd) can use PAM (see Section 2.2) to connect to the Authentication Router. Additionally, RADIUS client code exists in Perl, Python, Java, and many other common programming languages.

With these design principles in place, a system can be implemented by extending open source software and building XSEDE-specific management code.

Chapter 4

Authentication Router Implementation

4.1 XSEDE Central Database

The XSEDE Central Database is a pre-existing data store that contains information about all the users, resources, organizations, and other information related to XSEDE. It has quite a few tables that hold all the information that XSEDE knows about its users and resources. It is served by a redundant PostgreSQL database designed for high reliability.

For this project to integrate with the XSEDE database, a few new tables are required for information storage. The relevant existing tables, along with the proposed new tables, are described below and also shown graphically in Figure 4.1.

acct.people

Each person known to XSEDE has an entry in the "people" table. This stores basic information, such as name, organization, and title.

acct.system_accounts

Every time an account on a resource is created, an entry is added to the "system_accounts" table. This ties a person to a resource, giving their specific username at that resource.

acct.resources

Each computing resource in XSEDE is described in the "resources" table. A resource has a name as well as an organization that indicates the XSEDE partner site that manages the resource.

acct.organizations

XSEDE partner institutions are described in the "organizations" table. The organization ID is most important for SQL "join" statements.

acct.otp_tokens

The "otp_tokens" table is a new table used to describe each token deployed in the field. The table ties a specific token to a specific person. It also stores a username (in case the specific token type uses different usernames) and a reference to the token type table.

acct.otp_token_types

The "otp_tokens_types" table is a new table used to describe each type of OTP token deployed within XSEDE. It points to the XSEDE partner organization that manages the OTP resource. Currently, it does not store information about the specific OTP implementation, such as digit length, algorithm, or authentication server hostnames. This could be added in the future if desired.

acct.radius_clients

The "radius_clients" table is a new table used to describe client hosts that would use the Authentication Router service. It stores the client's hostname, IP address, shared secret (described in 2.3.3), and the XSEDE partner organization where the client lives. The organization code helps map usernames in case an organization code is not supplied in the authentication request.

4.2 RADIUS Framework

Several open source RADIUS implementations were evaluated for their maturity, stability, scalability, extensibility, and ease of use. Initially, a prototype RADIUS framework was built using pyrad [1], a RADIUS framework coded in the Python scripting language. It was highly configurable and customizable, but it lacked stability and some of the features present in a more mature product. At the time of writing, the last commit to the pyrad project was over eight months ago. Thus, a more full-featured and mature RADIUS framework was chosen to move this project

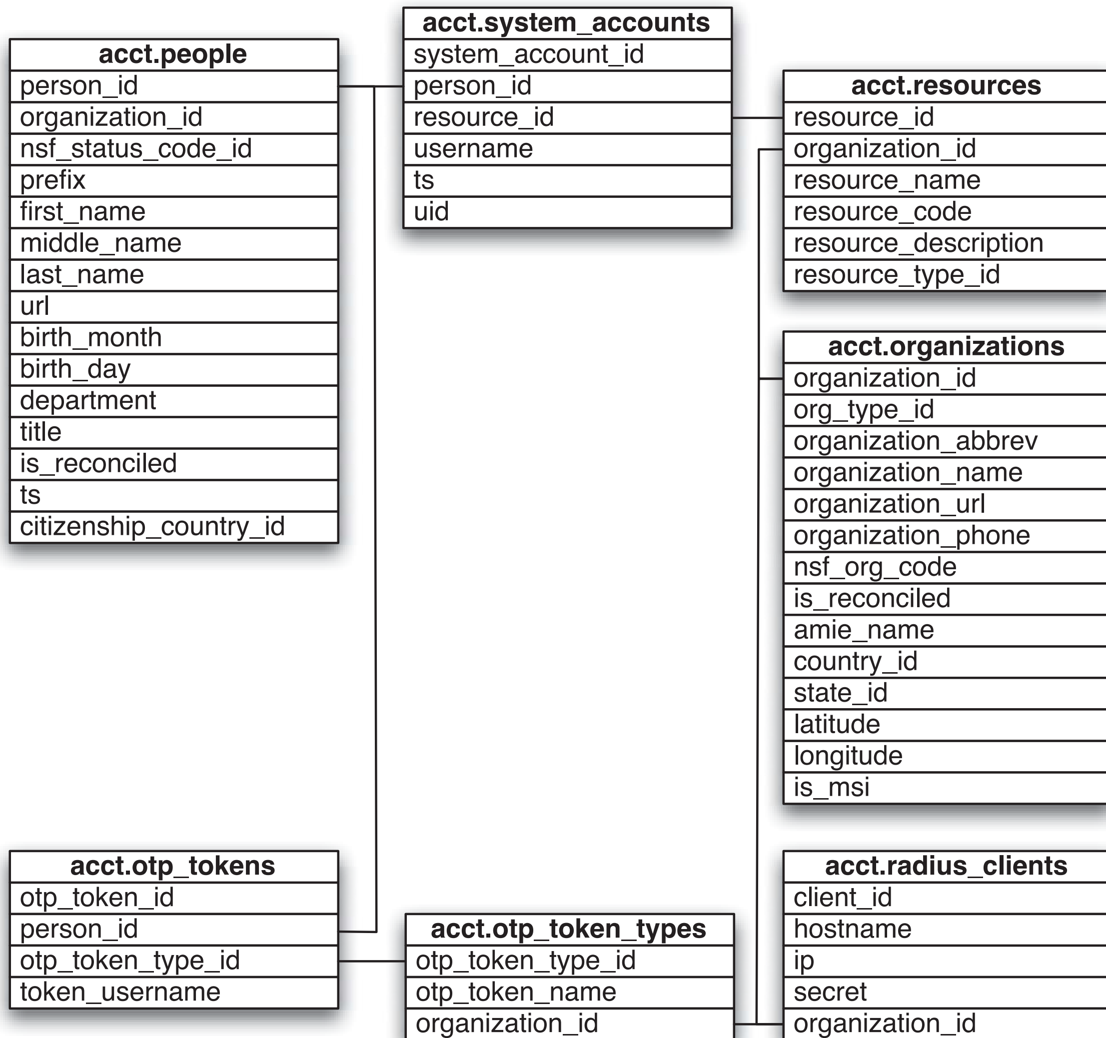

Figure 4.1: Database Schema

forward. The current iteration of the Authentication Router leverages the open source FreeRADIUS package [4] as a base on which the rest of the service is built.

FreeRADIUS' own website [4] claims that it is "the most widely deployed RADIUS server in the world" and asserts that it is "fast, feature-rich, modular, and scalable." Indeed, initial testing showed that the server was immediately functional with minimal configuration and was relatively easy to customize to the specific needs of this project.

The FreeRADIUS server supports both simple and complex configurations. The configuration used in the Authentication Router is included in Appendix A.

4.2.1 Modules

The FreeRADIUS server is built with support for many modules that can be plugged into the framework to extend the base functionality. The base distribution comes with modules to implement various authentication types as well as modules that allow easy integration with external systems.

A module called "rlm_perl" allows developers and server administrators to write extensions in the Perl scripting language. FreeRADIUS and rlm_perl use a threaded embedded Perl interpreter to execute the code. This means that the Perl code is compiled once at startup and is persistent for the life of the FreeRADIUS server process. Additionally, each FreeRADIUS thread houses its own copy of the Perl interpreter, allowing for both thread-local and global thread-shared variables. The majority of the code for the Authentication Router was implemented in Perl. The code is available in Appendix B.

4.2.2 Virtual and Home Servers

The concepts of realms and proxying was discussed in Section 2.3.4. FreeRADIUS supports these concepts through virtual servers and home servers. A virtual server acts like a completely separate RADIUS server that runs within the main server process. This enables complex configurations to be split up into logical pieces maintained separately within the configuration. Home servers allow FreeRADIUS to proxy requests to external RADIUS servers. It supports the concept of defining multiple remote servers for a given realm and load balancing between them for high availability and performance reasons.

In this project, both virtual servers and home servers are used extensively. The Authentication Router lives within the main FreeRADIUS server and accepts all incoming authentication requests. Once the request has been processed, the server knows if the authentication should be handled locally or passed off to a remote server. If the server can handle the authentication request, it is proxied to a virtual server. If the authentication request must be handled remotely, it is proxied to one of the home servers defined for that realm.

4.2.3 Logging

The FreeRADIUS server can be setup to log information in various ways depending on the verbosity required. By default, important events are written to a central file located on the file system at `/var/log/radius/radiusd.log`. Modules can also add

Table 4.1: FreeRADIUS Logging Codes

Code	Constant	Log Level Description
1	L_DBG	Debug
2	L_AUTH	Authentication
3	L_INFO	Informational
4	L_ERR	Error
5	L_PROXY	Proxy
6	L_ACCT	Accounting

events to the system log. In “rlm_perl”, messages can be added to the log with the following code:

```
&radiusd::radlog($loglevel, 'message');
```

The possible log levels, shown in Table 4.1, are available to flag messages with the appropriate category and severity. The message field is a string that includes the relevant information.

The server can also be configured to store authentication requests in a structured query language (SQL) database. This would allow easy report generation to answer questions like how many successful and failed authentication attempts occurred over a given timeframe. Currently, this project is not utilizing this feature due to scalability concerns.

4.3 Managing Clients

Every client that should interact with an Authentication Router must be setup in the configuration files. The server must be aware of the hostname, IP address, and shared secret (see Section 2.3.3) to be able to properly interpret the incoming RADIUS

packets. This configuration is stored in a file called `clients.conf` that is generated from the XSEDE database. A scheduled task could be setup to automatically generate this configuration file at a regular interval. A sample `clients.conf` is available in Appendix A.2.

FreeRADIUS only reads and processes its configuration at startup; this poses a problem when system administrators wish to add an additional client authorized to talk to the server. Instead of requiring all the Authentication Routers to be restarted each time a new client is added, the concept of dynamic clients was implemented.

When authentication requests come in from clients statically defined in the configuration, they are immediately processed. Unrecognized clients cause an internal request to be generated that gets handled by the XSEDE Perl module. The module attempts to query the XSEDE database for a match and authorizes the request if one is found. This client is then added into the server's running configuration so future requests can be served without querying the database. In case the client is not found in the database, the request is denied and the client is added to a time-decaying "bad cache" so that future requests from unauthorized clients can be denied without having to consult the database.

4.4 XSEDE FreeRADIUS Module

As mentioned in Section 4.2.1, the bulk of the code that implements the XSEDE-specific functions is located in a series of Perl files that are loaded into persistent

Table 4.2: XSEDE Module Command Line Arguments

Argument	Type	Function
conf	Boolean	Create clients.conf
verify	Boolean	Verify a client
client_site	Boolean	Map the default site for a client
client	String	IP address of the client to lookup
token	Boolean	Map a user/site to a token
portal	Boolean	Map a user/site to a portal user
user	String	The username to map
site	String	The site to map

Perl interpreters housed within the FreeRADIUS process. Multiple threads of the interpreter are started, allowing concurrent processing of incoming requests. Each thread has its own connection to the XSEDE database, as Perl's database interface is not thread-safe. Each thread uses the "prepare" method to preload queries within the database for faster searches later.

The main integration script named `xsede.pl` is the central entry point for the XSEDE module. If run from the command line, it will process the command line options to determine what function to perform. This gives system administrators the ability to query the same information that the Authentication Router has access to. Manually running commands can also help diagnose problems. The accepted command line options are shown in Table 4.2. If the code is instantiated from within FreeRADIUS, it will setup several data structures and connect to the database in preparation to service authentication requests. The code for `xsede.pl` is available in Appendix B.1.

A Perl file named `XSEDE::Common.pm` contains constant values, configuration information, logging functions, and common code. It includes functions for connecting

to and interacting with the XSEDE database. The source for this is included in Appendix B.2.

`XSEDE::Clients.pm` contains the code related to managing clients. It implements the functions described above in Section 4.3, including creating the clients.conf file, mapping a client to a default site with cache, and authorizing new client connections with a cache of "bad" clients. This source for this is included in Appendix B.3.

The code handling users is stored in `XSEDE::Users.pm` and also included in Appendix B.4. The most commonly used function maps a user and site to an OTP token type and a token username. The module will have to do a database query to determine this information, but caching has been implemented to speed this up for common users. The module also contains a function to map a given username at a site to the canonical portal username.

`XSEDE::Cache.pm` is a custom module derived from Jesse Vincent's Perl module *Cache::Simple::TimedExpiry* [22]. The XSEDE version, available in Appendix B.5, adds the ability to share cache objects between threads.

4.5 Operational Issues

Interacting with the the Authentication Router is a mandatory step whenever an end-user desires to connect to a resource that uses the Authentication Router as an authentication source. Thus, performance, scalability, reliability, and security are

important and necessary factors to consider. FreeRADIUS has been developed since 1999, and it has been deployed by thousands of sites worldwide [4].

FreeRADIUS performance depends on many factors, including database usage, server hardware, and the complexity of the configuration. The FreeRADIUS server is designed to be highly-threaded, allowing it to take advantage of modern multi-core processors. Furthermore, the code has been optimized and improved over the years. The FreeRADIUS website [4] claims that the most computationally-intense part of the server is the security hash generation. The XSEDE-specific code has been optimized by implementing cache everywhere possible. The Perl interpreters stay in memory, allowing the Perl code to executed multiple times after being interpreted only once.

The RADIUS protocol supports multiple servers, improving scalability by allowing the load to be spread in a scale-out fashion. Additional hardware can be added whenever bottlenecks are found. Simple domain name service (DNS) round-robin techniques will spread multiple clients to different servers, helping to prevent many clients from overwhelming a single server. Additionally, hierarchies of RADIUS servers, as described in Section 3.2, can be built to “funnel” the requests and allow fair-share quality of service.

Configuring multiple servers not only improves performance, it also adds reliability and fault tolerance in case of failure. Clients can specify multiple servers (or pools of servers) so that they can failover in case a server does not respond within a configurable amount of time.

The code is open for inspection, allowing any interested party to audit the code for security vulnerabilities. The Authentication Router code has been designed to "fail open" such that any error condition would cause the request to fail. It is preferable to fail a legitimate request rather than allow an illegitimate request to succeed.

The security of authentication-related functions is extremely important; close attention must be paid to the Authentication Router. A compromise of the Authentication Router could allow an attacker to intercept authentication credentials. This is somewhat mitigated by the use of OTPs, but an attacker could block the original authentication request and use the acquired credentials to pose as the original user. The servers that hosts the Authentication Routers should be single-purpose and hardened against attacks. Best practices in security should be followed when possible. The list of users allowed to login directly to the machine should be limited to only a few system administrators. The operating system and all installed software should be regularly patched to ensure all the latest security vulnerabilities are fixed. Additionally, a firewall should be configured to further protect the machine.

4.6 Cost

The Authentication Router system is completely built around free and open source software. Thus, the only cost associated with the Authentication Router service is the hardware on which it runs. OTP services typically have hardware or license fees, which may be non-trivial. The flexibility of pluggable OTP services within the

Authentication Router makes it easy to leverage existing OTP solutions where the costs have already been borne.

Chapter 5

Authentication Router Testing

5.1 Test Setup

The Authentication Router is designed to be lightweight and streamlined, requiring very little system resources. Development and initial testing was performed on a simple desktop machine with a dual-core AMD Athlon processor. The machine was installed with the CentOS 6.2 Linux operating system, a free variant of the Red Hat Enterprise Linux operating system. FreeRADIUS 2.1.10 was installed from the operating system repository with the following command:

```
yum install freeradius freeradius-utils freeradius-perl
```

To simulate multiple clients, virtual machines (VMs) were created with the Kernel-based Virtual Machine (KVM) software that is built into the operating system. Each VM had its own IP address so that it would appropriately exercise the code dealing

with authorizing new and existing clients. Several tests were run on the prototype using the open source tools described in Section 5.2.

To facilitate further testing, the National Institute for Computational Sciences at the University of Tennessee setup a virtual machine with client access to their RSA servers. This VM had four virtual processors and four gigabytes of memory. The entire prototype was also implemented in this environment and tested in a similar manner.

5.2 Test Tools

5.2.1 Radtest

Radtest is a simple utility that comes with the `freeradius-utils` package in Red Hat Enterprise Linux. It is designed to directly query RADIUS servers for simple testing. The important parameters are the username, password, RADIUS server hostname, and the shared secret. A "NAS port number" should also be specified, which can be a random integer. Simply call *radtest* with the appropriate parameters:

```
radtest username password radius-server nas-port-number secret
```

Radtest will show the server's reply, which will be an Access-Accept or Access-Reject RADIUS packet.

A simple script can be written to generate a test workload to validate the server's operation. For full test coverage, authentication as both existing and non-existing users should be attempted.

5.2.2 Radclient

Radclient is another utility that comes with `freeradius-utils`. It can read a file with a collection of RADIUS requests, including their attributes, and query the RADIUS server. It also has the capability to send multiple requests in parallel to take advantage of the multi-threaded server. Unfortunately, the tool itself is not multi-threaded. For extremely rapid request rates, the performance is likely limited to the tool's performance on a single core. The syntax for calling *Radclient* is fairly simple:

```
radclient -f input_file -p num_parallel_requests server auth secret
```

This tool can be wrapped with the *time* utility to calculate the number of requests processed per second.

5.2.3 Pamtester

Pamtester [17] is an open source utility designed to test PAM module and configurations. A PAM configuration file should be setup to implement the "auth" method. Optionally, a "client_id" can be supplied to indicate which site the request originates from. A sample PAM configuration file could be as simple as:

```
auth required pam_radius_auth.so client_id=NICS
```

To test that PAM is properly configured, *Pamtester* should be invoked with the name of the PAM configuration file pointing to the RADIUS module, a username, and an operation such as authenticate.

```
pamtester service username operation
```

This will prompt for a password prior to sending the authentication request to the server. The response will indicate if the request was successful or not.

5.3 Test Plan

Extensive testing with both manual and automated test suites has shown that the Authentication Router framework is capable of working quickly, efficiently, and correctly. Setting up database indexes should enable the queries to be served much faster. Enabling the module's cache is expected to significantly improve performance when the same user and site combination attempts repeated authentications. It is believed that the virtual machine overhead may skew the numbers, but the results can be compared relatively to examine the effects of database indexing, caching, and parallelism.

The database was loaded with fake authentication tokens for the 3820 users defined with username mappings at NICS. These tokens were setup to automatically accept any password provided. The first test should iterate through each of these users and attempt an authentication. The second test should choose a single user and attempt to authenticate many times. The *Radclient* software was configured to handle both of these scenarios.

The Authentication Router should be tested with database indexing both on and off and caching both on and off. Then, increasing numbers of parallel requests should be tested to determine the scalability of the Authentication Router.

5.4 Problems Experienced

Testing revealed an issue with the way FreeRADIUS handled server threads. With a spike in traffic, FreeRADIUS would spawn new threads up to a configurable limit. When the increased traffic level subsided, FreeRADIUS failed to reduce the number of active threads. The author reported this issue to the developers along with a proposed patch. The developers addressed the problem, and they committed a fix to the issue that will be present in a future release of FreeRADIUS. Though this issue may result in extra threads remaining in the idle state instead of exiting, it is not believed that this will negatively impact the Authentication Router.

Additionally, a problem was discovered when multiple threads of rlm_perl were spawned in quick succession. An internal FreeRADIUS variable was not properly protected from concurrent access. This could cause the FreeRADIUS process to crash and exit. The fix to this issue is to serialize the access with a mutex, and this fix has been committed to the project's version control system. The patch was extracted and FreeRADIUS was recompiled to include the fix. The problem was not experienced with the patch applied. A request has been opened with RedHat to include this patch in future versions of RedHat Enterprise Linux.

5.5 Test Results

5.5.1 Authenticating Each User Once

The performance results from attempting to authenticate as each user one time is shown in Table 5.1 and graphed in Figure 5.1. With no database indexing or caching, almost fifty requests per second can be achieved with only eight concurrent requests. Enabling database indexes greatly improves performance, up to eight threads where it tops out likely due to constrained resources. With both indexing and caching, over three thousand requests per second can be achieved with eight concurrent requests. The site caching is helpful here because only a single client is tested, but the user caching adds overhead for no gain. Unfortunately, adding additional concurrent requests actually lowers performance due to overhead in adding users to the cache.

5.5.2 Authenticating One User Many Times

Authenticating the same user many times should greatly benefit from server caching. The results are shown in Table 5.2 and Figure 5.2. As was seen in the previous test case, the lack of indexing and caching results in approximately fifty requests per second. Turning on indexing improves the performance up to a point. That point is slightly higher due to caching in the database layer. Turning on application caching greatly improves performance, achieving over five thousand requests per second at eight client threads. Unfortunately, performance significantly drops over eight concurrent requests. Reasons for this performance issue are described in Section 5.6.

Table 5.1: Test Results Authenticating Each User Once

Indexes	Cache	Par. Req.	# Req	Time (sec)	Req/sec
X	X	1	3820	280.53	13.62
X	X	2	3820	139.17	27.45
X	X	4	3820	88.49	43.17
X	X	8	3820	76.48	49.95
X	X	16	3820	71.74	53.25
X	X	32	3820	69.57	54.91
X	✓	1	3820	281.31	13.58
X	✓	2	3820	139.04	27.47
X	✓	4	3820	89.57	42.65
X	✓	8	3820	77.18	49.49
X	✓	16	3820	71.48	53.44
X	✓	32	3820	70.01	54.56
✓	X	1	3820	4.43	862.30
✓	X	2	3820	2.94	1299.32
✓	X	4	3820	1.87	2042.78
✓	X	8	3820	1.36	2808.82
✓	X	16	3820	1.37	2788.32
✓	X	32	3820	1.40	2728.57
✓	✓	1	3820	2.87	1331.01
✓	✓	2	3820	2.23	1713.00
✓	✓	4	3820	1.45	2634.48
✓	✓	8	3820	1.13	3380.53
✓	✓	16	3820	1.30	2938.46
✓	✓	32	3820	1.48	2581.08

Table 5.2: Test Results Authenticating One User Many Times

Indexes	Cache	Par. Req.	# Req	Time (sec)	Req/sec
X	X	1	3820	281.88	13.55
X	X	2	3820	141.59	26.98
X	X	4	3820	87.56	43.63
X	X	8	3820	76.57	49.89
X	X	16	3820	70.97	53.83
X	X	32	3820	69.81	54.72
X	✓	1	3820	2.58	1480.62
X	✓	2	3820	1.53	2496.73
X	✓	4	3820	1.02	3745.10
X	✓	8	3820	0.78	4897.44
X	✓	16	3820	0.98	3897.96
X	✓	32	3820	1.67	2287.43
✓	X	1	3820	3.84	994.79
✓	X	2	3820	2.66	1436.09
✓	X	4	3820	1.87	2042.78
✓	X	8	3820	1.34	2850.75
✓	X	16	3820	1.28	2984.38
✓	X	32	3820	1.43	2671.33
✓	✓	1	3820	2.40	1591.67
✓	✓	2	3820	1.35	2829.63
✓	✓	4	3820	0.87	4390.80
✓	✓	8	3820	0.73	5232.88
✓	✓	16	3820	0.84	4547.62
✓	✓	32	3820	1.12	3410.71

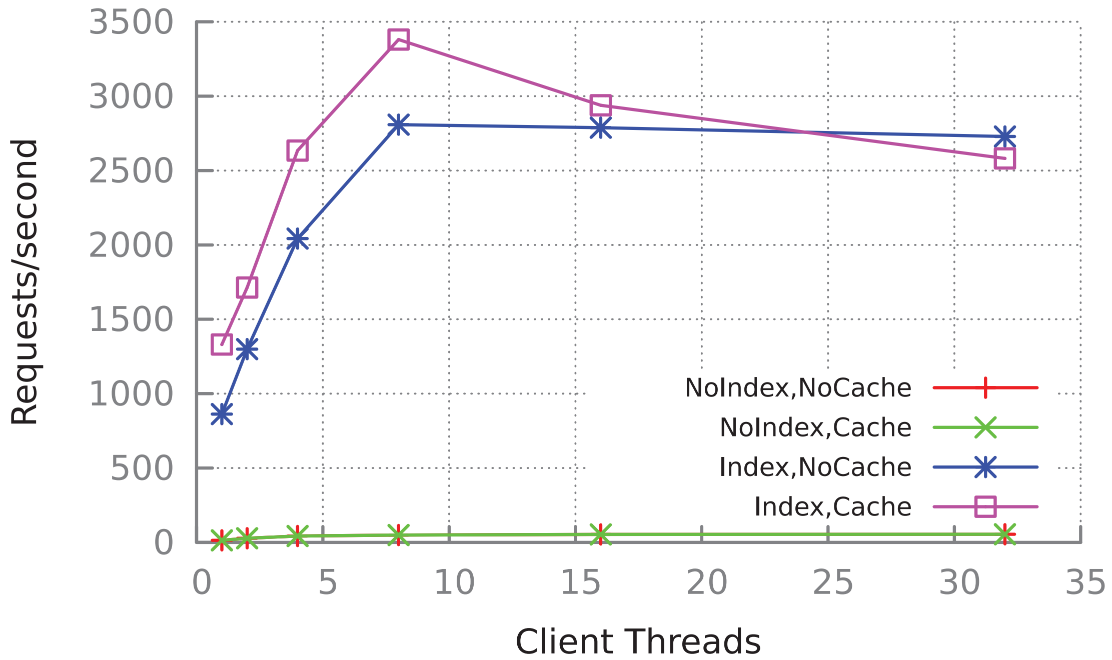

Figure 5.1: Test Results Authenticating Each User Once

5.6 Testing Summary

Two main factors contribute to decreased performance when many threads are used concurrently. The first is an artifact of the test setup. FreeRADIUS does not spawn additional threads until a load spike warrants additional threads. Each time a new thread spawns, it must instantiate a new Perl interpreter, copy the non-shared data from another thread, and connect to the database. For short-running tests, the time required to spawn threads is non-trivial. The second factor that appears to decrease performance is Perl's implementation of thread-shared variables. A single thread "owns" each variable and access to it from other threads requires cooperation

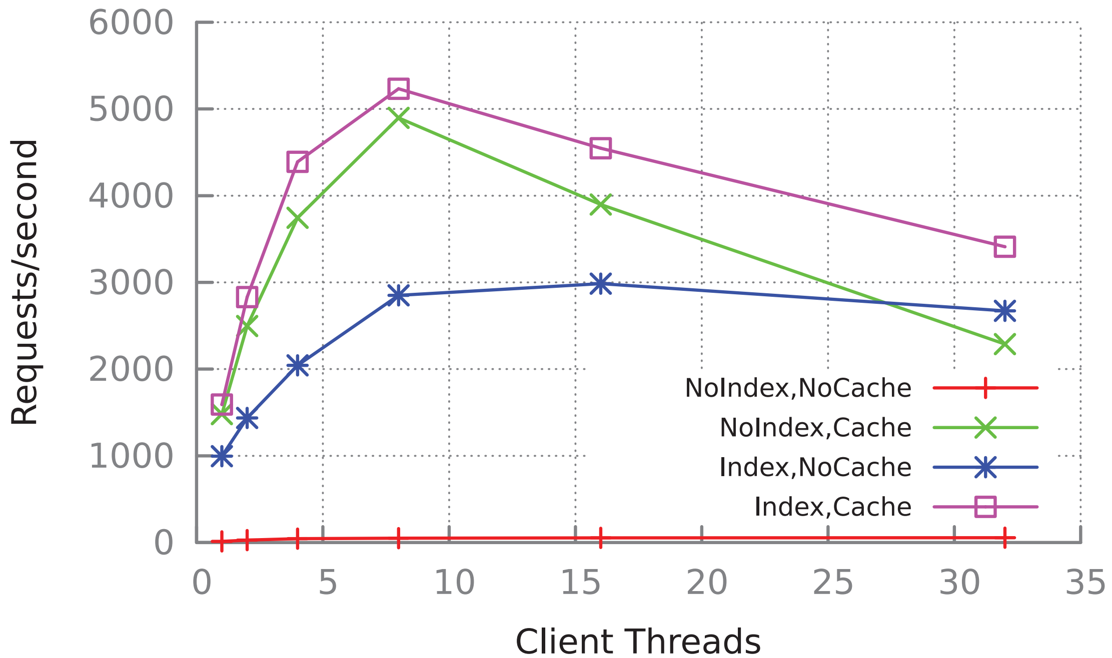

Figure 5.2: Test Results Authenticating One User Many Times

from this master thread. With lots of threads, contention around access to these variables slows performance. Though cache appears to improve performance, there are tradeoffs.

In June of 2011, NICS recorded logins from 851 distinct users throughout the entire month[6]. Even if every user attempted to login at exactly the same time, a single Authentication Router could easily handle the load. The performance achieved in testing greatly exceeds the requirements expected in a production deployment for XSEDE, especially if multiple Authentication Routers are deployed.

Chapter 6

Conclusions

6.1 Thesis Summary

Many standards exist for authentication, ranging from simple static passwords stored on a single machine to complicated distributed systems. Organizations concerned about protecting their digital assets from sophisticated cyber attacks have begun relying on two-factor authentication as a defense against unauthorized access.

The concepts presented in this thesis provide an extensible framework that allows multiple institutions to federate their strong two-factor authentication services across distributed cyberinfrastructure, providing potentially enormous cost savings. A fully-functional prototype was created for the National Science Foundation's XSEDE cyberinfrastructure consortium. This Authentication Router service, built upon industry standards and open source software, is designed to be high performance, scalable, and secure.

6.2 Future Work

The obvious next step in this research area is to work with XSEDE staff to implement the Authentication Router as a production resource within the XSEDE infrastructure. The author is in close contact with several key members of the XSEDE operations team, and implementation seems likely in the future. XSEDE staff would want to audit the service to verify the findings presented here and test at a larger scale. Publications describing the specific implementation details would be appropriate for the XSEDE conference as well as various security and grid-related conferences.

For the Authentication Router concept to achieve more adoption within the community, the details presented here should be generalized and made available to other consortia with similar organizational structures as XSEDE. Specifically, it would be useful to reach out to the original authors of the RADIUS Authentication Fabric papers [12] [13] to determine if this concept could be implemented across the Department of Energy National Laboratories. To further extend this framework, it would be beneficial to evaluate additional features to extend the functionality.

In National Science Foundation supercomputing, some users write scientific workflows based on the Globus grid toolkit. Globus uses X.509 certificates for authentication. Direct integration with MyProxy [2], an X.509 certificate credential repository, could help extend strong authentication to grid software. Hazlewood et al. [6] present an example of using an OTP backend with MyProxy; it would be trivial to substitute the Authentication Router. Additionally, it should be feasible to modify

the PAM RADIUS module to fetch an X.509 certificate when a user authenticates. This would allow an end user to obtain a grid certificate "for free" upon login. This could allow "single sign-on" or more complex scenarios where a single login corresponds to a single job.

In distributed cyberinfrastructure where a single user can have accounts at many service providers, it can be difficult to remember which username corresponds to which site. Currently, most of the generic client tools that would use the Authentication Router require end users to know their username at the site they are attempting to login to. Contrasted with systems like *eduroam* that only care about a user's association, most of the systems in XSEDE require that connections correspond to a specific username. This situation could be improved by modifying client tools to accept realm-specific usernames, such as "user1@site1." The Authentication Router would then handle mapping the remote realm username to the site username and sending this back to the client. This has been achieved in a prototype by modifying the code for the RADIUS PAM module as well as the OpenSSH secure shell daemon. Having these patches pushed upstream would make this idea feasible.

For XSEDE's initial implementation of the Authentication Router, all authentication traffic should occur between trusted hosts over XSEDE's private network, a trusted network link. Additional security could be added to the Authentication Router by implementing RADSEC (described in Section 2.3.5). Additional work would be required to replace shared secret management with client certificate management.

Bibliography

Bibliography

[1] W. Akkerman. pyrad: Python RADIUS Implementation. URL https://github.com/wichert/pyrad. 29

[2] Jim Basney, Marty Humphrey, and Von Welch. The myproxy online credential repository. *Software: Practice and Experience*, 35(9):801–816, 2005. ISSN 1097-024X. doi: 10.1002/spe.688. URL http://dx.doi.org/10.1002/spe.688. 51

[3] L. Florio and K. Wierenga. Eduroam, providing mobility for roaming users. In *Proceedings of the EUNIS 2005 Conference, Manchester*, 2005. 16

[4] FreeRADIUS. FreeRADIUS: The world's most popular RADIUS Server. URL http://www.freeradius.org. 31, 37

[5] N. Haller, C. Metz, P. Nesser, and M. Straw. A One-Time Password System. RFC 2289 (Standard), February 1998. URL http://www.ietf.org/rfc/rfc2289.txt. 11

[6] Victor Hazlewood, Patricia Kovatch, Matthew Ezell, Matthew Johnson, and Patti Redd. Improved grid security posture through multi-factor authentication.

Grid Computing, IEEE/ACM International Workshop on, 0:106–113, 2011. ISSN 1550-5510. doi: http://doi.ieeecomputersociety.org/10.1109/Grid.2011.41. 1, 49, 51

[7] J. Howlett and S. Hartman. Project Moonshot, June 2010. URL http://www.project-moonshot.org/sites/default/files/moonshot-tnc-2010.pdf. 17

[8] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. URL http://www.ietf.org/rfc/rfc2104.txt. Updated by RFC 6151. 13

[9] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226 (Informational), December 2005. URL http://www.ietf.org/rfc/rfc4226.txt. 12

[10] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-Based One-Time Password Algorithm. RFC 6238 (Informational), May 2011. URL http://www.ietf.org/rfc/rfc6238.txt. 14

[11] D. M'Raihi, J. Rydell, S. Bajaj, S. Machani, and D. Naccache. OCRA: OATH Challenge-Response Algorithm. RFC 6287 (Informational), June 2011. URL http://www.ietf.org/rfc/rfc6287.txt. 15

[12] D. Muruganantham, M. Helm, and T. Genovese. ESnet authentication services and trust federations. In *Journal of Physics: Conference Series*, volume 16, page 591. IOP Publishing, 2005. 18, 51

[13] D. Muruganantham, M. Helm, T. Genovese, R. Morelli, and J. Webster. ESnet RAF Progress Report. *Work in progress*, May 2005. 18, 51

[14] M. Naedele. Standards for XML and Web services security. *Computer*, 36(4): 96–98, 2003. 10

[15] National Science Foundation. National Science Foundation. URL http://www.nsf.gov/. 3

[16] OATH. An Industry Roadmap for Open Strong Authentication, 2006. URL http://www.openauthentication.org/webfm_send/14. 12

[17] Pamtester. pamtester - test pluggable authentication modules (PAM) facility. URL http://pamtester.sourceforge.net/. 42

[18] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote Authentication Dial In User Service (RADIUS). RFC 2865 (Draft Standard), June 2000. URL http://www.ietf.org/rfc/rfc2865.txt. Updated by RFCs 2868, 3575, 5080. 5

[19] RSA. Open Letter to RSA Customers, . URL http://www.rsa.com/node.aspx?id=3872. 11

[20] RSA. RSA SecurID Authenticators: The gold standard in two-factor authentication, . URL http://www.rsa.com/products/securid/datasheets/2305_h9061-sid-ds-0212.pdf. 11

[21] V. Samar. Unified login with pluggable authentication modules (pam). In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10. ACM, 1996. 4

[22] Jesse Vincent. Cache::simple::timedexpiry. URL http://search.cpan.org/~jesse/Cache-Simple-TimedExpiry-0.27/lib/Cache/Simple/TimedExpiry.pm. 36

[23] S. Winter, M. McCauley, S. Venaas, and K. Wierenga. Transport Layer Security (TLS) encryption for RADIUS, February 2012. URL http://tools.ietf.org/html/draft-ietf-radext-radsec. 9, 10

[24] XSEDE. XSEDE Overview, 2012. URL https://www.xsede.org/ca/overview. 1, 3

Appendix

Appendix A

FreeRADIUS Configuration Listing

A.1 radiusd.conf

```
prefix = /usr
exec_prefix = /usr
sysconfdir = /etc
localstatedir = /var
sbindir = /usr/sbin
logdir = ${localstatedir}/log/radius
raddbdir = ${sysconfdir}/raddb
radacctdir = ${logdir}/radacct
name = radiusd
confdir = ${raddbdir}
run_dir = ${localstatedir}/run/${name}
db_dir = ${raddbdir}
libdir = /usr/lib64/freeradius
pidfile = ${run_dir}/${name}.pid
user = radiusd
group = radiusd
max_request_time = 30
cleanup_delay = 5
max_requests = 1024

# Main server
listen {
  type = auth
  ipaddr = *
  port = 0
}

# Control Socket
```

```
29 listen {
30   type = control
31   socket = ${run_dir}/${name}.sock
32   uid = root
33   gid = root
34   mode = rw
35 }
36
37 hostname_lookups = no
38 allow_core_dumps = no
39 regular_expressions = yes
40 extended_expressions  = yes
41
42 log {
43   destination = files
44   file = ${logdir}/radius.log
45   syslog_facility = daemon
46   stripped_names = no
47   auth = no
48   auth_badpass = no
49   auth_goodpass = no
50 }
51
52 checkrad = ${sbindir}/checkrad
53
54 security {
55   max_attributes = 200
56   reject_delay = 1
57   status_server = yes
58 }
59 proxy_requests  = yes
60
61 $INCLUDE clients/
62
63 thread pool {
64   start_servers = 5
65   max_servers = 32
66   min_spare_servers = 3
67   max_spare_servers = 10
68   max_requests_per_server = 0
69 }
70
71 modules {
72   perl xsede {
73     module = ${confdir}/perl/xsede.pl
74   }
75   dynamic_clients {
76     # No config for this
77   }
78 }
79
```

```
80 authorize {
81   xsede
82 }
83 authenticate {
84   #
85 }
86
87 # Dynamic client server
88 server dynamic_client_server {
89   authorize {
90     update control {
91       FreeRADIUS-Client-IP-Address = "%{Packet-Src-IP-Address}"
92     }
93     xsede
94   }
95 }
```

A.2 Sample clients.conf

```
1 client localhost {
2   ipaddr=127.0.0.1
3   secret=testing123
4 }
5 client bert {
6   ipaddr=192.168.1.9
7   secret=testing123
8 }
```

Appendix B

Source Code Listing

B.1 Main Integration Script

```
1 #!/usr/bin/perl
2 # Integration functions for the XSEDE Authentication Routers
3 # Designed to run standalone OR within FreeRADIUS
4 # Originally written by Matt Ezell <matt@mattezell.com>
5
6 use strict;
7 use DBI;
8 use Switch;
9
10 # Pull in common variables and constants
11 use lib '/etc/raddb_xsede/perl';
12 use XSEDE::Common;
13 use XSEDE::Clients;
14 use XSEDE::Users;
15
16 # Check to see if we are within FreeRADIUS
17 if(defined(&radiusd::radlog)){
18   xsede_log(L_INFO,"xsede module initializing...");
19 } else {
20   # Read command line arguments
21   use Getopt::Long;
22   my($dump_client_conf,$verify_client,$get_client_site,$client_ip);
23   my($map_token,$map_portal,$user,$site);
24   my $result = GetOptions("conf"        => \$dump_client_conf,
25          "verify"       => \$verify_client,
26          "client_site"  => \$get_client_site,
27          "client=s"     => \$client_ip,
28          "token"        => \$map_token,
```

```
29          "portal"       => \$map_portal,
30          "user=s"       => \$user,
31          "site=s"       => \$site
32          );
33   if($dump_client_conf) {
34     xsede_db_connect();
35     dump_client_conf();
36   } elsif($verify_client && $client_ip) {
37     xsede_db_connect();
38     prepare_clientquery();
39     if(client_authorize($client_ip)==2) {
40       print "Authorized\n";
41       exit 0;
42     }else{
43       print "Not Authorized\n";
44       exit 1;
45     }
46   } elsif($get_client_site && $client_ip) {
47     xsede_db_connect();
48     prepare_clientquery();
49     my $site=find_default_site($client_ip);
50     if($site) {
51       print "$client_ip is at $site\n";
52     } else {
53       print "Unknown\n";
54     }
55   } elsif($map_token && $user && $site) {
56     xsede_db_connect();
57     prepare_tokenquery();
58     my($token_type,$token_username)=map_user_to_token($user,$site);
59     print "$user at $site has a $token_type token with username
         $token_username\n";
60   } else {
61     print "Unknown options\n";
62   }
63 }
64
65 # Function to handle authorization
66 # This could be either a client or a user
67 sub authorize {
68   if(defined($RAD_CHECK{'FreeRADIUS-Client-IP-Address'})) {
69     # Authorize Client
70     xsede_log(L_INFO,'Authorizing client ' . $RAD_CHECK{'FreeRADIUS-
         Client-IP-Address'});
71     return client_authorize($RAD_CHECK{'FreeRADIUS-Client-IP-Address'});
72   } else {
73     # Authorize User
74     my $user = $RAD_REQUEST{'User-Name'};
75     if(!$user) {
76       xsede_log(L_ERR, "xsede:: Dropping request because no user was
           specified");
```

```
77        return RAD_FAIL;
78      }
79      my $site;
80      if(defined($RAD_REQUEST{'NAS-Identifier'})){
81        $site = $RAD_REQUEST{'NAS-Identifier'};
82      } else {
83        $site = find_default_site($RAD_REQUEST{'FreeRADIUS-Client-IP-
            Address'});
84      }
85      if(!$site) {
86        xsede_log(L_ERR, "xsede:: no site defined");
87        return RAD_FAIL;
88      }
89      xsede_log(L_DBG,"xsede:: site is $site");
90      my($ttype,$tuser)=map_user_to_token($user,$site);
91      if($ttype && $tuser) {
92        xsede_log(L_INFO, "xsede:: sending $user at $site to $ttype as
            $tuser");
93        $RAD_REQUEST{'User-Name'}=$tuser;
94        $RAD_CHECK{'Proxy-to-realm'}=$ttype;
95      } else {
96        xsede_log(L_ERR, "xsede:: Not mapped, user:$user, site:$site,
            token:$ttype tuser:$tuser");
97        return RAD_FAIL;
98      }
99      return RAD_OK;
100   }
101 }
102
103 # Function for authentication
104 # Not currently used, but certain token types may in the future
105 sub authenticate {
106   # Always fail
107   return RAD_FAIL;
108 }
109
110 # This is called when a new thread is spawned
111 sub CLONE {
112   xsede_log(L_INFO, "xsede:: Spawning a new interpreter thread...");
113   xsede_db_connect() or die("Can't connect to XSEDE Central DB");
114   prepare_clientquery();
115   prepare_tokenquery();
116   prepare_portalquery();
117 }
118
119 # This is called when threads exit
120 sub detach {
121   xsede_log(L_INFO,"xsede:: shutting down an interpreter thread");
122   xsede_db_disconnect();
123 }
```

B.2 Common Settings and Functions

```
1 #!/usr/bin/perl
2
3 package XSEDE::Common;
4 use strict;
5 use DBI;
6 use threads;
7 use threads::shared;
8
9 # Sites to make User maps for
10 our @sites=qw(NICS TACC NCSA SDSC);
11
12 # Settings for Database Connection
13 our %DB;
14 $DB{HOST}    = 'localhost';
15 $DB{PORT}    = 5432;
16 $DB{NAME}    = 'xsede';
17 $DB{USER}    = 'xsede';
18 $DB{PASS}    = 'xsede';
19 $DB{HANDLE} = undef;
20 $DB{connected} = 0;
21 $DB{failtime}  = 0;
22
23 # Settings for Cache
24 our %CACHE;
25 $CACHE{sites_enable}      = 1;
26 $CACHE{clients_enable}    = 1;
27 $CACHE{clients_duration} = 60*10;
28 $CACHE{token_enable}      = 1;
29 $CACHE{token_duration}    = 60*10;
30
31 # Return codes, as defined by FreeRADIUS
32 use constant {
33         RAD_REJECT   => 0,# /* Immediately reject          */
34         RAD_FAIL     => 1,# /* Module failed, don't reply */
35         RAD_OK       => 2,# /* Module is OK, continue     */
36         RAD_HANDLED => 3,# /* Handled the request, stop  */
37         RAD_INVALID => 4,# /* Request is invalid          */
38         RAD_USERLOCK=> 5,# /* User is locked, reject     */
39         RAD_NOTFOUND=> 6,# /* User not found, reject     */
40         RAD_NOOP     => 7,# /* Succeeded, but did nothing */
41         RAD_UPDATED => 8,# /* OK (pairs modified)         */
42         RAD_NUMCODES=> 9 # /* Number of return codes     */
43 };
44
45 # Log Levels
46 use constant {
47   L_DBG  => 1, # Debug
48   L_AUTH => 2, # Authentication
49   L_INFO => 3, # Informational
```

```
50   L_ERR   => 4, # Error
51   L_PROXY=> 5, # Proxying
52   L_ACCT  => 6  # Accounting
53 };
54
55 # Pull variables from main FreeRADIUS, if they are there
56 use vars qw(%RAD_REQUEST %RAD_REPLY %RAD_CHECK);
57
58 # Our own logging function to handle inside and outside FR
59 sub xsede_log {
60   my($level,$message) = @_;
61   my @levels = qw(Debug Auth Proxy Info Error);
62    if(defined(&radiusd::radlog)) {
63     &radiusd::radlog($level,$message);
64   } else {
65     print "Message from outside!!\n";
66     print $levels[$level] . $message . "\n";
67   }
68 }
69
70 # Connect to the XSEDE Database
71 # Args:     None
72 # Returns: None
73 sub xsede_db_connect {
74   #my %attr = ( PrintError => 0, RaiseError => 0 );
75   my %attr = ( PrintError => 1, RaiseError => 1 );
76   #my %attr;
77   $DB{HANDLE} = DBI->connect(
78     "dbi:Pg:dbname=$DB{NAME};host=$DB{HOST};port=$DB{PORT}",
79     $DB{USER}, $DB{PASS}, \%attr );
80 }
81
82 # Disconnect from the XSEDE Database
83 # Args:     None
84 # Returns: return code from disconnect()
85 sub xsede_db_disconnect {
86   return $DB{HANDLE}->disconnect();
87 }
88
89 # Allow everything above to be exported
90 require Exporter;
91 our @ISA = 'Exporter';
92 our @EXPORT = qw(RAD_REJECT RAD_FAIL RAD_OK RAD_HANDLED RAD_INVALID
93        RAD_USERLOCK RAD_NOTFOUND RAD_NOOP RAD_UPDATED
94        L_DBG L_AUTH L_INFO L_ERR L_PROXY L_ACCT
95        %RAD_REQUEST %RAD_REPLY %RAD_CHECK
96        @sites %DB %CACHE
97        xsede_log
98        xsede_db_connect
99        xsede_db_disconnect
100      );
```

```
101
102 # Always return true
103 1;
```

B.3 RADIUS Clients Functions

```
1 #!/usr/bin/perl -w
2 # This implements functions and data structures to handle clients
3 # Originally written by Matt Ezell <matt@mattezell.com>
4
5 use strict;
6 use DBI;
7 use Cache::Simple::TimedExpiry;
8 use threads;
9 use threads::shared;
10
11 package XSEDE::Clients;
12
13 # Pull in common variables and constants
14 use XSEDE::Common;
15
16 # Create a failed client cache
17 my $failed_clients = Cache::Simple::TimedExpiry->new;
18 $failed_clients->expire_after($CACHE{clients_duration});
19
20 # Cache "default site" for clients
21 # Only need to cache if client didn't specify a site
22 # Assume that clients don't typically change default sites,
23 # So cache this forever. We can just use a simple hash
24 my %site_map :shared;
25
26 # Prepared statement to look up a single client
27 my $clientquery;
28 sub prepare_clientquery {
29   $clientquery = $DB{HANDLE}->prepare(q{ select hostname,ip,secret,
       amie_name
30     from acct.radius_clients rc,acct.organizations o
31     where rc.organization_id = o.organization_id
32     AND ip=? });
33 }
34
35 # Authorize a new client
36 # Note, existing clients should be in the clients.conf
37 # Args:     client_ip
38 # Sets:     RADIUS_values
39 # Returns:  RAD_return_code
40 sub client_authorize {
41   my $client = shift;
42
43   # Check to see if the client has recently failed
44   if ($CACHE{clients_enable} && $failed_clients->has_key($client)) {
45     xsede_log(L_DBG,"xsede_clients: $client already in failed cache,
         ignoring");
46     return RAD_NOTFOUND;
47   }
```

```
48
49   # Run the query (prepared earlier)
50         $clientquery->execute($client);
51         if ( my($hostname,$ip,$secret,$site) = $clientquery->
             fetchrow_array() ) {
52                 $RAD_CHECK{'FreeRADIUS-Client-IP-Address'} = $ip;
53                 $RAD_CHECK{'FreeRADIUS-Client-Shortname'}  = $hostname;
54                 $RAD_CHECK{'FreeRADIUS-Client-Secret'}     = $secret;
55                 xsede_log(L_INFO, "xsede_clients: Authorizing client
                    $hostname,$ip");
56                 return RAD_OK;
57         } else {
58                 xsede_log(L_INFO, "xsede_clients: Refusing to add client
                     $client");
59     $failed_clients->set($client,1);
60                 return RAD_NOTFOUND;
61   }
62 }
63
64 # Find default site
65 # Args:    client_ip
66 # Returns: default_site
67 sub find_default_site {
68   my $client = shift;
69   if($CACHE{sites_enable} && defined($site_map{$client})){
70     xsede_log(L_DBG,"xsede:: cache hit for default site ($client,
        $site_map{$client})");
71     return $site_map{$client};
72   } else {
73     # Look up site
74     $clientquery->execute($client);
75     my ($hostname,$ip,$secret,$site) = $clientquery->fetchrow_array();
76     $site_map{$client} = $site if ($CACHE{sites_enable});
77     return $site;
78   }
79 }
80
81 # Dump client conf
82 sub dump_client_conf {
83   # Prepare SQL to find all clients
84   my $clientsquery;
85         $clientsquery = $DB{HANDLE}->prepare(q{ select hostname,ip,
             secret,amie_name
86       from acct.radius_clients rc,acct.organizations o
87       where rc.organization_id = o.organization_id});
88   $clientsquery->execute();
89   open CLIENTS, ">", "/etc/raddb_xsede/clients/clients.conf" or die $!;
90   while ( my ($hostname,$ip,$secret,$site) = $clientsquery->
       fetchrow_array() )  {
91     print CLIENTS "client $hostname {\n";
92     print CLIENTS "\tipaddr=$ip\n";
```

```
93      print CLIENTS "\tsecret=$secret\n}\n";
94    }
95    close CLIENTS;
96 }
97
98 # Allow everything above to be exported
99 require Exporter;
100 our @ISA = 'Exporter';
101 our @EXPORT = qw(find_default_site
102        prepare_clientquery
103        client_authorize
104        dump_client_conf
105                    );
106
107 # Always return true
108 1;
```

B.4 RADIUS Users Functions

```
1 #!/usr/bin/perl
2 # This implements functions and data structures to handle Users
3 # Originally written by Matt Ezell <matt@mattezell.com>
4
5 use strict;
6 use Switch;
7 use Cache::Simple::TimedExpiry;
8
9 package XSEDE::Users;
10
11 # Pull in common variables and constants
12 use XSEDE::Common;
13 use XSEDE::Clients;
14 use XSEDE::Cache;
15 use threads::shared;
16
17 # Create a user to token map
18 # This *might* change, so only remember for a configurable period
19 #my $user_token_map = Cache::Simple::TimedExpiry->new;
20 #$user_token_map->expire_after($CACHE{token_duration});
21 my $user_token_map = XSEDE::Cache->new($CACHE{token_duration});
22
23 # Create a portal user map
24 # This is unlikely to change, so we can keep a hash
25 my %user_portal_map;
26
27 # A prepared statement to look up a user token
28 my $tokenquery;
29 sub prepare_tokenquery {
30   $tokenquery = $DB{HANDLE}->prepare(q{
31          select distinct saa.username as localuser,
32          tt.otp_token_name as token_type,
33          t.token_username as token_username
34   from
35          acct.system_accounts saa,
36          acct.resources r,
37          acct.organizations o,
38          acct.otp_tokens t,
39          acct.otp_token_types tt
40          where saa.username=?
41   and o.amie_name=?
42          and saa.person_id = t.person_id
43          and saa.resource_id = r.resource_id
44          and r.organization_id = o.organization_id
45          and t.otp_token_type_id = tt.otp_token_type_id});
46 }
47
48
49 # Map a user/site to a token
```

```
50 # Args:      username
51 #            site_name
52 # Returns: @(token_type,token_username)
53 sub map_user_to_token {
54   my $username = shift;
55   my $site = shift;
56
57   if ($CACHE{token_enable} && $user_token_map->has_key("$site,$username"
       )) {
58     my $cacheref = $user_token_map->get("$site,$username");
59     xsede_log(L_INFO,"xsede:: map_user_to_token: $site,$username already
         in cache");
60     return(split(/,/,$$cacheref));
61   }
62
63   # Run the query (prepared earlier)
64   $tokenquery->execute($username,$site);
65   if (my($user,$token_type,$token_username) = $tokenquery->
       fetchrow_array() ) {
66     if($CACHE{token_enable}) {
67       my $value = "$token_type,$token_username";
68       $user_token_map->set("$site,$username",share($value));
69       xsede_log(L_INFO,"xsede:: map_user_to_token: Adding $site,
           $username to cache");
70     }
71     return ($token_type,$token_username);
72   } else {
73     return undef;
74   }
75 }
76
77 # A prepared statement to look up a portal username
78 my $portalquery;
79 sub prepare_portalquery {
80   $portalquery = $DB{HANDLE}->prepare(q{
81   select distinct saa.username as localuser,
82          sab.username as xduser
83   from
84          acct.system_accounts saa,
85          acct.system_accounts sab,
86          acct.resources r,
87          acct.organizations o
88          where saa.username=?
89   and o.amie_name=?
90          and saa.person_id = sab.person_id
91          and sab.resource_id=2013
92          and saa.resource_id = r.resource_id
93          and r.organization_id = o.organization_id;});
94 }
95
96 # Map a user/site to a portal username
```

```
97 # Args:      username
98 #            site_name
99 # Returns:  portal_username
100 sub map_user_to_portal {
101         my $username = shift;
102         my $site = shift;
103 
104         # Run the query (prepared earlier)
105         $portalquery->execute($username,$site);
106         if (my($user,$portal_username) = $tokenquery->fetchrow_array() )
            {
107                 return $portal_username;
108         } else {
109                 return undef;
110         }
111 }
112 
113 # Allow everything above to be exported
114 require Exporter;
115 our @ISA = 'Exporter';
116 our @EXPORT = qw(prepare_tokenquery
117     map_user_to_token
118     prepare_portalquery
119     map_user_to_portal
120     );
121 # Always return true
122 1;
```

B.5 Multi-Threaded Caching Functions

```perl
#!/usr/bin/perl
# This implements thread-safe caching
# Derived from Cache::Simple::TimedExpiry by Jesse Vincent <
    jesse@bestpractical.com>
use strict;

package XSEDE::Cache;

use threads;
use threads::shared;

sub new {
  my $class = shift;
        share(my @self);
  my $duration = shift;
  $duration = defined($duration) ? $duration : 600;
  $self[0] = $duration;
  $self[1] = &share({});
  $self[2] = &share([]);
  $self[3] = 0;
  return (bless(\@self, $class));
}

sub has_key ($$) {
  my ($self, $key) = @_;
  my $time = time;
  $self->expire($time) if ($time > $self->[3]);
  return 1 if defined $key && exists $self->[1]->{$key};
  return 0;
}

sub get ($$) {
  my ($self,$key) = @_;
  unless ( $self->has_key($key)) {
    return undef;
  }
  return $self->[1]->{$key};
}

*set = \&store;
sub store ($$$) {
  my ($self,$key,$value) = @_;
  return undef unless defined ($key);
  my $time = time;
  $self->expire($time) if ($time > $self->[3]);
  $self->[1]->{$key} = $value;
  my @value :shared = ( time, $key );
  push @{$self->[2]}, \@value;
}
```

```
49
50 sub expire ($$) {
51   my ($self,$time) = @_;
52   $self->[3] = $time;
53   my $oldest_nonexpired_entry = ($time - $self->[0]);
54   return unless defined $self->[2]->[0]; # do we have an element in the
       array?
55   return unless $self->[2]->[0]->[0] < $oldest_nonexpired_entry; # is it
       expired?
56   while ( @{$self->[2]} && $self->[2]->[0]->[0] <
       $oldest_nonexpired_entry ) {
57     my $key =  $self->[2]->[0]->[1];
58     delete $self->[1]->{ $key };
59     shift @{$self->[2]};
60   }
61 }
62
63 1;
```

Vita

Matthew Allan Ezell received a Bachelor of Science in Electrical Engineering from the University of Tennessee. Following his graduation, he became a full-time High Performance Computing System Administrator at the National Institute for Computational Sciences at The University of Tennessee. He specialized in administering large Cray systems operated for the National Science Foundation. Matthew also lead a project to implement two-factor authentication for the NSF's XSEDE cyberinfrastructure. In 2012, he changed employment to the National Center for Computational Sciences at Oak Ridge National Laboratory. Matthew can be reached at matt@mattezell.com.