Adapting Kerberos for a Browser-based Environment

by

David Benjamin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 22, 2013

Certified by .....
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by .....
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Adapting Kerberos for a Browser-based Environment

by

David Benjamin

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents *Webathena*, a browser-centric implementation of the Kerberos network authentication protocol. It consists of a JavaScript Kerberos client, paired with a simple, untrusted, server-side proxy to wrap the protocol in HTTP. This is used to implement a trusted credential manager with a cross-origin protocol to delegate credentials to untrusted Web applications.

To evaluate *Webathena*, we present *Roost*, a Web-based client for the Zephyr messaging and notification in use at MIT, along with a host of proof-of-concept applications. We find that it is possible to build Web-based clients for Kerberized services similar to or better than existing native ones with no modifications to either the Kerberos KDCs or the services themselves. Finally, we discuss possible modifications to Kerberos to better support this kind of credential delegation.

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I thank my advisor, Nickolai Zeldovich, for supporting this work and for help on thesis direction and organization.

Thanks to Alan Huang for discussions on design and his help in implementing the front-end for the initial versions of *Webathena*. With his help, I was able to focus on the Kerberos portions, and *Webathena* could acquire service tickets and forward to a functional proof-of-concept application within two weeks of beginning implementation.

This work was deployed with the help of the Student Information Processing Board for providing the scripts.mit.edu, XVM, and sql.mit.edu services which host *Webathena* and *Roost*. In particular, I thank Alex Chernyakhovsky for his prompt support with use of these services and for extensive discussions on design and implementation. I also thank him for being an early adopter of *Webathena* in prototyping the new sql.mit.edu website. Thanks also to Anders Kaseorg and Tim Abbott for their discussions on Kerberos and credential delegation.

I thank members of the MIT Kerberos Consortium, including Thomas Hardjono, Greg Hudson, Ben Kaduk, Zhanna Tsitkova, and Tom Yu, for meetings and discussion on the Kerberos protocol.

Finally, I thank Leonid Grinberg for coming up with the name *Webathena*, thus providing the final motivation to actually bring this idea to fruition.

Contents

1 Introduction **13**

2 Kerberos **15**

2.1 Tickets 15

2.2 GSSAPI 16

3 Webathena **19**

3.1 Challenges 19

3.2 Design 20

3.3 Implementation 21

3.3.1 HTTP-to-Kerberos proxy 21

3.3.2 JavaScript Kerberos client 22

3.3.3 Credential delegation 23

3.3.4 Summary 24

3.4 Sample applications 25

3.4.1 zwrite.js 25

3.4.2 Shell In A Box integration 26

3.4.3 ctlfish 26

4 Roost **29**

4.1 Zephyr 29

4.1.1 Protocol 29

4.1.2 Current use 32

4.1.3 Web-based clients . . . 32
4.2 Design . . . 33
4.3 Implementation . . . 34
4.3.1 Subscriber . . . 34
4.3.2 Changes to *libzephyr* . . . 35
4.3.3 Public API . . . 37
4.3.4 Clients . . . 38
4.3.5 Summary . . . 39

5 Related Work **41**
5.1 OAuth . . . 41
5.2 HTTP Negotiate . . . 41
5.3 scripts.mit.edu . . . 42
5.4 Zulip . . . 42

6 Discussion and Future Work **43**
6.1 Client code integrity . . . 43
6.2 Credential delegation . . . 44
6.2.1 Android . . . 45
6.2.2 Finer-grained delegation . . . 45
6.3 Native Kerberos integration . . . 46
6.4 Persistent access and revocation . . . 46
6.4.1 Ticket renewal . . . 47
6.4.2 Ticket revocation . . . 48
6.5 Constrained ticket-granting-tickets . . . 48

7 Conclusions **51**

A Webathena API **53**

B Sample Code **57**

C Testimonials **59**

List of Figures

2-1 A sample Kerberos interaction. 17

3-1 An ticket request in *Webathena* 20
3-2 An application requesting a service ticket from *Webathena* 21
3-3 A sample *Webathena* permissions prompt. 24
3-4 A sample ticket-granting ticket request. 27

4-1 Overview of the Zephyr protocol 31
4-2 Messages being delivered to *Roost*. 34

List of Tables

3.1 *Webathena* components 24
3.2 *Webathena* sample applications 25

4.1 *Roost* components 39

Chapter 1

Introduction

In 1983, the Massachusetts Institute of Technology, along with Digital Equipment Corporation and IBM, began Project Athena [1] to provide a campus-wide computing environment for the Institute. At the center of Athena was Kerberos [41, 33], an authentication protocol used to authenticate users to various services on MIT's network. These include AFS, a networked filesystem, Moira, a mailing list and group manager, Zephyr, a notification and messaging system, and dialup machines for remote login. In addition, MIT community members can register their own Kerberos-based services. These services tend to be accessed with command-line clients using native Kerberos libraries.

Today, Kerberos and other parts of Project Athena live on, but the computing environment has changed. Applications are increasingly moving to the Web and accessed through a browser. On the Web, low-level system access, such as access to raw sockets, is limited. More fundamentally, browsers treat application code as untrusted, isolating them from the system and from each other. Newer platforms have similar models, including Android, iOS, and Windows 8. Moreover, with the advent of cloud computing, remote servers are increasingly used to perform actions on behalf of a user. In this environment, it is important to not only provide applications and remote servers access to a user's account, but also to scope and limit that access.

In parallel with Kerberos, MIT deploys Web-based services on its network. These services primarily authenticate using TLS client certificates [10] and are separate from

Kerberos. A native client cannot use Kerberos to authenticate to such a website, and there is no natural way to access a Kerberos-based service from the browser. Some Kerberos services do have Web clients, but, unlike native ones, they require special provisions in the services themselves. For instance, WebMoira, the Web interface to Moira, authenticates to Moira itself using a trusted superuser principal on the backend and then reimplement authentication checks. This has historically been a source of problems as these checks did not always match those of Moira.

Our work is motivated by the disparity between authentication on Web-based services and that of native Kerberos ones. We believe that Web applications and native applications should eventually converge and services should be able to treat them equivalently. To that end, we present *Webathena*, an adaptation of Kerberos to the Web. It allows Web-based access Kerberos-based services at MIT with no modifications to the existing Kerberos infrastructure.

The Web presents several challenges to adapting a protocol like Kerberos. Low-level networking APIs such as UDP and TCP sockets are unavailable on the Web, so we cannot implement Kerberos directly. In addition, since applications on the Web are expected to be isolated from each other, we must design appropriate boundaries and security policies to decide which applications may access which services.

Webathena handles the constraints of a browser-based environment primarily by deploying proxies which wrap existing protocols in ones available to the browser, namely HTTP [13] and WebSockets [12]. It then provides APIs for applications to request credentials at the service granularity, prompting the user for permission to forward access. We implement several applications using this system, including *Roost*, a fully-featured client for Zephyr, a Kerberos-based messaging system used at MIT.

Chapter 2 gives an overview of Kerberos as used today. We then discuss *Webathena* in detail in Chapter 3. Chapter 4 describes our primary case study, *Roost*. We discuss related work in Chapter 5. Chapter 6 evaluates our overall system as compared to a traditional Kerberos ecosystem and discusses future work. This includes proposed changes to the Kerberos protocol to better serve the needs of the Web and modern platforms. Finally, Chapter 7 concludes.

Chapter 2

Kerberos

As background for *Webathena*, this chapter presents a brief overview of Kerberos. The Kerberos network authentication protocol [41] provides unified access to services across all of Athena. It allows users to securely authenticate to servers using their Athena credentials without sending their password over the network.

At the center of a Kerberos deployment is a central authentication server, the *Key Distribution Center* (KDC). The KDC includes a database with pre-arranged shared secrets for each user and service in the system. In the case of a user, this secret is the user's password. Users and services, however, do not share secrets. The goal of the Kerberos protocol is to arrange a shared session secret between the user and the service. This secret can then be used as part of a secure protocol between the two.

2.1 Tickets

The Kerberos protocol works using *tickets*. To access a particular service, the user requests a ticket for that service from the KDC. This ticket can be used to authenticate the user to that service. We describe tickets in more detail below.

The user begins by requesting access to a service from the KDC. The KDC responds with a session key, freshly generated for the user and the service, as well as a ticket. Tickets contain a copy of the session key and some metadata, all encrypted with the service's secret. Likewise, the user's copy of the session key is encrypted

with the user's password. This encryption allows the the user and the service to trust that this session key was generated by the KDC for them because no one else could have encrypted it with their respective secrets. Tickets are timestamped and last for a limited amount of time, usually on the order of a day.

After receiving the ticket, the client constructs an *authenticator* which contains a copy of the ticket and some timestamps encrypted with the session key. This is presented to the service. The service decrypts the ticket to learn the session key and uses this to verify the authenticator. From there, the client and service may communicate over some application-specific protocol.

A user may access several different services during their login session. To avoid require they re-enter their password for each service, Kerberos introduces a special service, the *ticket-granting service* (TGS). The TGS is usually a component of the KDC. Tickets for the TGS are known as *ticket-granting tickets* or TGTs. The TGS is a special service which has access to the KDC's database can issue tickets for other services. When a user logs in to their machine, they request a TGT and store it in a *credential cache*. Then, as they need to access services, they use this TGT to acquire service tickets without re-entering their password. Figure 2-1 shows a typical flow for a user accessing a service with a TGT.

Kerberos also provides a ticket renewal mechanism. Tickets may be marked as renewable and have a second *renew-till* expiration date. Any time before a ticket expires, clients may request the TGT issue a new one. This new ticket has the same renew-till time, and has the same lifetime (but newer start time) as the old ticket or ends at the renew-till time, whichever comes earlier.

2.2 GSSAPI

Most modern protocols using Kerberos use the *Generic Security Services API* [29] or GSSAPI. GSSAPI is a generalized API that may be implemented by different authentication providers, called *mechanisms*. It is primarily used with the Kerberos mechanism [44]. GSSAPI mechanisms allow applications to establish an authen-

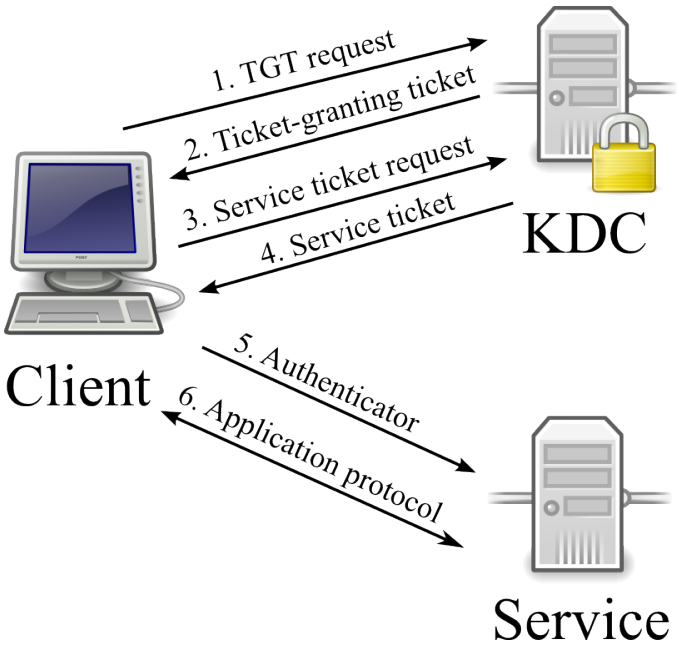

Figure 2-1: A sample Kerberos interaction.

ticated security context and optionally a messaging layer providing confidentiality and/or integrity-checking.

The two primary entry points in GSSAPI are `GSS_Init_sec_context` and `GSS_Accept_sec_context`. These are called by the *context initiator* (usually the client) and the *context acceptor* (usually the service), respectively. The initiator first calls `GSS_Init_sec_context` to generate an opaque context-establishment token and sends it to the acceptor. The acceptor passes it to `GSS_Accept_sec_context` which may return another token to be sent back to another call to `GSS_Init_sec_context`. This process repeats until the context has been established. This context may then be used with the messaging layer.

In the case of Kerberos, the initiator's token consists of an authenticator. If mutual authentication has been requested, the acceptor's response contains a re-encrypted version of the authenticator's timestamp. Otherwise, there is no response. After this, both sides have an established context with a session key to be used with GSSAPI's messaging layer. The GSSAPI implementation abstracts away service ticket requests and does so as-needed on calls to `GSS_Init_sec_context`.

Chapter 3

Webathena

This section details *Webathena*, a Kerberos client tailored for Web browsers. The name is a play on *Debathena* [8], a port of MIT's Athena computing environment to Debian-based systems. In that vein, *Webathena* is a "port" of Athena to the Web.

3.1 Challenges

Adapting Kerberos to a browser environment presents several challenges. The first is that Web content is very restricted in its access to the outside world. JavaScript code running in a browser cannot open UDP or TCP sockets. This means that a Kerberos implementation in the browser cannot directly speak to the KDC. We could simply send the password to a trusted server which speaks Kerberos on the client's behalf, but this runs counter to Kerberos' design goals. The user's password should not be stored in memory long-term or leave the network. A native Kerberos client has no need of a trusted proxy.

The second challenge is related. Since a user may run a Web application as easily as clicking a link in an email, the Web was forced to evolve a security model with *untrusted applications*. Web applications are not only isolated from the user's system, but also from each other. A native application typically links in the MIT Kerberos library which interacts directly with the user's credentials. This would not be acceptable for the Web, as that would allow any site the user visits to steal their

MIT credentials.

3.2 Design

To work around networking constraints, we deploy a server-side proxy. The Kerberos protocol uses a stateless protocol over UDP[1]. All interactions involve the client sending a single request and receiving a single response from the KDC. This allows us to easily wrap it in HTTP. We pair the proxy with a custom JavaScript Kerberos client. Although this proxy handles all of *Webathena*'s Kerberos traffic, it never sees the user's password. It is only as powerful as an active network attacker, and we inherit the Kerberos protocol's security properties under that threat model. Note that this is only true of the proxy itself, and not the server hosting *Webathena*'s JavaScript code. We discuss this in more detail in Chapter 6. This design is summarized in figure 3-1.

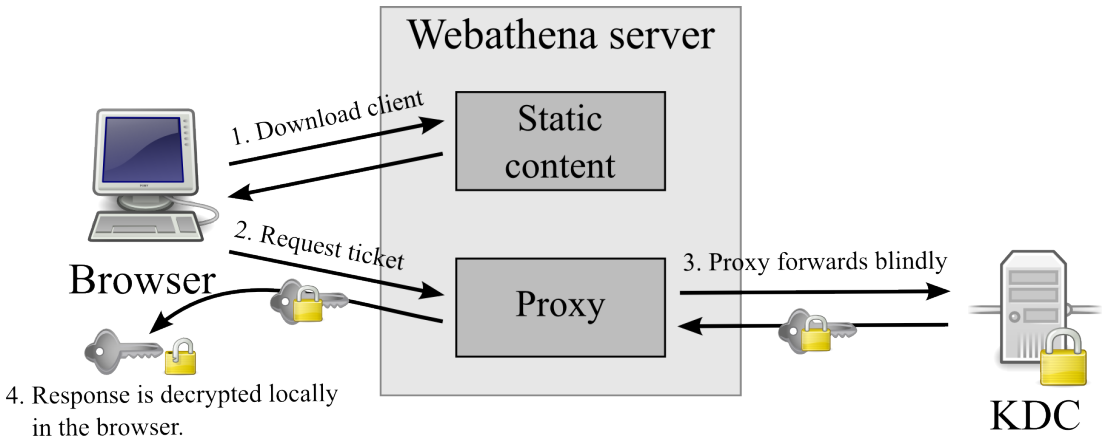

Figure 3-1: An ticket request in *Webathena*

To adapt Kerberos to an environment with untrusted applications, we add a ticket delegation API for *Webathena*. *Webathena* itself acts as a trusted credentials cache to manage the user's Kerberos credentials. This is analogous to how `https://accounts.google.com/` manages a user's Google login for other sites. We expose an API based on `postMessage` [20], a cross-window communications mecha-

[1]There is also a TCP transport, but it is not deployed at MIT.

nism, for other websites to request service tickets. It prompts the user for permission before giving access, much like those in systems based on OAuth [19]. If the user approves, *Webathena* forwards a service ticket for the application to use. Everything is scoped to a page's origin [3] (protocol, host, and port), the security principal used in client-side browser security. This is summarized in figure 3-2.

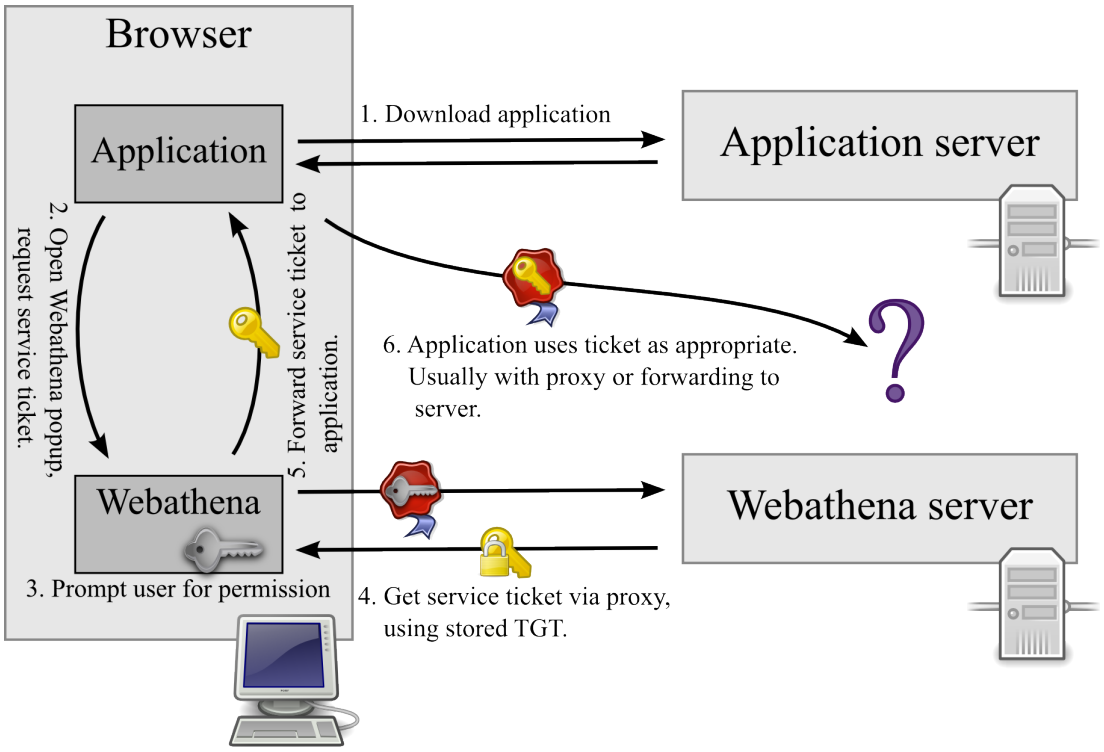

Figure 3-2: An application requesting a service ticket from *Webathena*

3.3 Implementation

3.3.1 HTTP-to-Kerberos proxy

The proxy is implemented in Python and hosted on the scripts.mit.edu [38] service, run by the MIT Student Information Processing Board (SIPB). It takes base64-encoded Kerberos requests as `POST` data, decodes them, and sends them to the KDC. When the KDC responds, it encodes as base64 and returns it as the HTTP response. The proxy also handles retransmitting lost packets and cycling between different KDC

instances using a clone of the algorithm in MIT Kerberos.

As currently implemented, the proxy itself is not quite a dumb proxy. It decodes and validates requests before forwarding them along. We discovered during implementation that the MIT Kerberos KDC does not always return an error on malformed requests. For some errors, it simply drops the packet. scripts.mit.edu only allows a limited number of concurrent HTTP requests to a single Web application, so *Webathena*'s proxy ensures that the request is well-formed before sending. This is to prevent a denial-of-service attack on the proxy by forcing it to timeout on bad requests. However, it does not and cannot decrypt the encrypted portions of the request.

As a security consideration, the proxy also limits the destination of its traffic. A completely open socket proxy could unwittingly allow outside access to machines which treat traffic coming from, say, MIT's network as privileged in some way. We thus conservatively only allow traffic to Athena's KDC.

3.3.2 JavaScript Kerberos client

On the client, *Webathena* consists entirely of static files which are served by a normal Web server, in our case the Apache instance running on scripts.mit.edu. It displays a login prompt for the user which, when submitted, requests a ticket-granting ticket from the KDC (via our proxy). MIT's KDC requires the `PA-ENC-TIMESTAMP` preauthentication method, so we implement it as well. Tickets are persisted in `localStorage`.

As in a normal Kerberos implementation, the client does all the cryptography. We two different libraries for cryptography. Kerberos' modern encryption profiles [35] are based on AES and SHA-1. We use the Stanford Javascript Crypto Library [40] (SJCL) implementations of these primitives. MIT is still transitioning away from the older profiles [36] based on single-DES and MD5, so we use the implementations in CryptoJS [32]. Long-term, we plan to remove support for single-DES in *Webathena* once MIT has completed its transition. As the Web Cryptography API [6] is finalized and implemented, we also plan to use it where available. This will allow us to leverage

the browser's native cryptography implementation.

In addition to cryptographic ciphers, a Kerberos client needs access to a cryptographic random number generator. Recent versions of modern browsers provide a `window.crypto.getRandomValues` API [6] to provide a source of randomness, but not all do. So instead, we use SJCL's random number generator. This uses the native API where available, but also incorporates entropy from as many sources as possible, such as mouse movements and load times. I normally waits for sufficient entropy before returning bytes. The resulting user experience is poor, so we seed the generated with entropy from the proxy.

Otherwise, the *Webathena* client is a normal from-scratch implementation of Kerberos, but in JavaScript. One component of note is our ASN.1 and DER implementation, located at `web_scripts/js/asn1.js` in the source repository. It implementants a JavaScript domain-specific language for transcribing ASN.1 types. The transcriptions of Kerberos structures are located at `web_scripts/js/krb_proto.js`.

As proactive security measures, we configure several HTTP headers which protect against many standard Web security vulnerabilities. We enable HTTP Strict Transport Security [22] to force SSL usage and defeat SSL-stripping attacks. We also enable a strict Content Security Policy [4] header to mitigate any cross-site scripting attacks. Finally, we enable the `X-Frame-Options` [37] header to prevent some forms of click-jacking.

3.3.3 Credential delegation

We use Mozilla's WinChan [21] library to implement the `postMessage` API. WinChan provides an RPC interface on top of `postMessage` and also works around some quirks of Internet Explorer's implementation. To request credentials, an application opens a window to *Webathena* and sends a message with a list of service principals. *Webathena* then prompts the user, and, if the user allows, service tickets are sent back to the application.

Instead of showing the full service principal name, *Webathena* interprets certain well known services and displays human-friendly names for them. For instance, re-

questing a ticket for `zephyr/zephyr@ATHENA.MIT.EDU` results in a prompt for

> Send and receive zephyr notices as you

while `host/xvm-remote.mit.edu@ATHENA.MIT.EDU` gives

> Access `xvm-remote.mit.edu` on your behalf

In addition, every prompt always includes "Learn your email address" as the user's Athena principal is included in the response. Figure 3-3 shows a sample permissions prompt which requests two service principals.

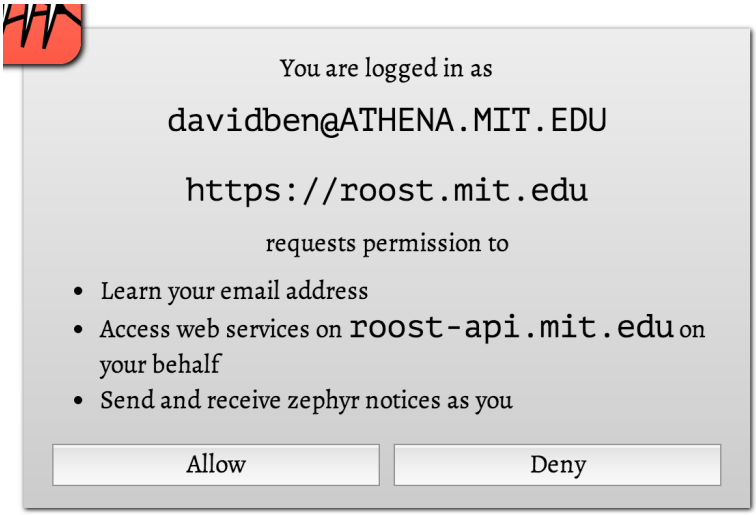

Figure 3-3: A sample *Webathena* permissions prompt.

3.3.4 Summary

Table 3.1 summarizes the various components of *Webathena*. The source code is available on GitHub at `https://github.com/davidben/webathena`.

Component	Environment	Language	Lines of Code
Proxy	Server	Python	786
Client	Browser	JavaScript	4338

Table 3.1: *Webathena* components. Line counts do not include third-party code.

3.4 Sample applications

We implement several proof-of-concept applications to demonstrate how *Webathena* can be used to provide Web interfaces to existing Kerberos services. These are a simplified Zephyr [9] notice sender, integration for Shell In A Box [18], and a remctl [30] implementation. The source for each of these may be found in the `samples` directory of the *Webathena* code repository. Table 3.2 summarizes these samples.

Application	**Language**	**Lines of Code**
zwrite.js	JavaScript, some Python	287
Shell In A Box integration	Python, some JavaScript	312
ctlfish	JavaScript	1330

Table 3.2: *Webathena* sample applications

3.4.1 zwrite.js

In advance of beginning work on *Roost*, described in Chapter 4, we implemented *zwrite.js*, a version of the `zwrite` utility from the Zephyr distribution which uses *Webathena*.

We describe the Zephyr protocol in more detail with *Roost* in Chapter 4. As *zwrite.js* need only send messages, it only concerns itself with a subset. Sending a Zephyr notice consists of sending a single UDP packet to a Zephyr host manager (`zhm`) running on the client's machine. The `zhm` then forwards this notice to the Zephyr servers, handling retransmits and acknowledgments. Once the server has acknowledged the message, it is forwarded back to the client. The packet contains a Kerberos authenticator and checksum used to authenticate the packet.

Although this was not the ultimate implementation strategy used in *Roost*, we implemented *zwrite.js* using a variant of the proxy strategy used in *Webathena* itself. We implement a dumb server-side proxy to the `zhm` running on scripts.mit.edu. This proxy receives a base64-encoded packet and forwards it to the `zhm`. The client-side component requests a ticket for `zephyr/zephyr@ATHENA.MIT.EDU`, the service principal used for Zephyr, and then assembles a packet to be sent out.

There is one complication in adapting this scheme for Zephyr. One of the fields in a Zephyr notice contains the IP address the notice was sent from. Before assembling the packet, we query a second URL which gives the server's IP address. That is then assembled into the packet. Scripts is load-balanced across several IPs, but we rely on the fact that their load-balancer pins to a particular IP until the client becomes inactive.

3.4.2 Shell In A Box integration

We implement a second proof-of-concept application for use with Shell In A Box [18]. Shell In A Box is a Web-based terminal emulator deployed on several Athena dialup servers to provide `ssh` access in the browser. It runs a command, usually `ssh` to `localhost`, on the remote server and connects it to a JavaScript terminal emulator running in the user's browser.

Webathena's Shell In A Box integration allows a user to login using their Kerberos credentials. Since Shell In A Box runs commands remotely on the server, this requires a different implementation strategy. We request a ticket from *Webathena*, as in *zwrite.js*, but serialize it and send it to the server. We then configure the server to run a wrapper script instead of `ssh` directly. This script writes the ticket into a Kerberos credentials cache and then runs `ssh` as before.

At MIT, users usually login to remote machines with credential forwarding. This sends a TGT to the remote machine, allowing the user to access their networked filesystem and other services from there. To support this use, we modify *Webathena* to allow services to request a TGT instead of a service ticket. The permissions prompt displays this request as "Full access to your Athena account" and annotates it with a caution icon. We show a sample such prompt in figure 3-4.

3.4.3 ctlfish

As a final proof-of-concept, we implement *ctlfish*, a Web-based client for remctl [30]. Remctl allows a server to configure a set of commands that can be executed remotely

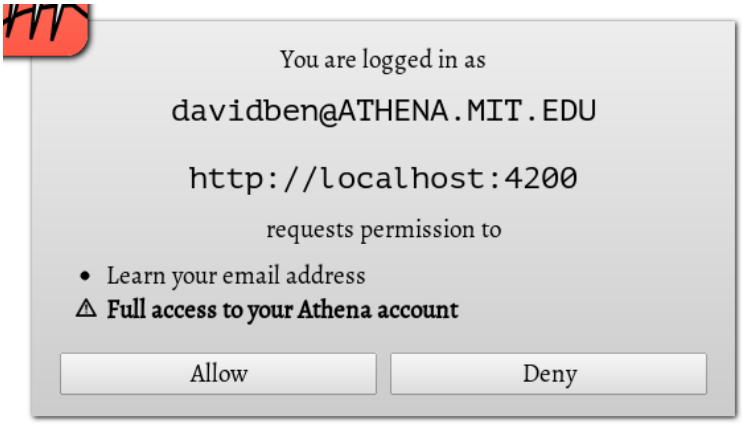

Figure 3-4: A sample ticket-granting ticket request.

by an authenticated user. XVM [43], SIPB's virtual machine service for the MIT community, exposes a number of remctl commands to query and manipulate a user's virtual machines.

Remctl's wire protocol is a straight-forward use of GSSAPI, making it an ideal GSSAPI sample for *Webathena*. The protocol uses a simple framing layer to send GSSAPI messages over a TCP socket. It first establishes a context to authenticate the user and then uses the security layer to encrypt and authenticate all messages.

ctlfish's design is similar to that of *Webathena*. Unlike Kerberos, remctl's protocol does not map as directly HTTP's request/response semantics. Instead we deploy a TCP-to-WebSockets proxy written in Node.js [34]. We use SockJS [39] to fall back to HTTP-based long-polling techniques on browsers where WebSockets are not available. Like the *Webathena* proxy, the *ctlfish* TCP proxy conservatively limits destinations to a whitelist of hostnames and ports for security reasons.

We then implement a reusable GSSAPI library in JavaScript on top of the Kerberos libraries already written for *Webathena*. This library is used in conjunction with the proxy to build a JavaScript implementation of the remctl protocol. We have deployed it on Red Hat's OpenShift platform at `https://ctlfish-davidben.rhcloud.com/`. OpenShift was chosen because scripts.mit.edu does not support Node.js, and XVM was having difficulties creating new virtual machines at the time.

Although *ctlfish*'s own uses are currently limited, we note that it is being used to

build the new interface to SIPB's SQL service, sql.mit.edu [31]. All operations for managing a user's databases will be exposed as remctl commands, natively accessible via the remctl client. The Web interface will simply request tickets and act as any other client. This achieves our ultimate goal of unified authentication for Web and native applications to a service.

Chapter 4

Roost

We now present *Roost*, our primary *Webathena* case study and the use *Webathena* was originally conceived for. *Roost* is a client for Zephyr[9], a notification and messaging service used at MIT. It receives messages on the user's behalf and logs them to a database to be retrieved later when the user is online. This makes *Roost* a particularly interesting case for adapting to *Webathena*. Not only must the user delegate Zephyr access to *Roost*, but this access must be persistent. We wish to receive messages for the user while they are offline. In addition, the user must authenticate to *Roost* itself for access to their logged messages, so we have the opportunity to design a Kerberos-based authentication scheme over HTTP for a brand-new service and avoid the proxies from our sample applications. Our solutions to these problems are very specific to Zephyr, so we begin with an overview of Zephyr and how it is used today.

4.1 Zephyr

4.1.1 Protocol

Zephyr was originally designed as a notification system for Project Athena. Services could broadcast status information to all subscribers or to a particular user. For instance, a file server might notify everyone of a pending shutdown, or a print server might inform a user their print job completed. It also served as a messaging service.

Every notice is sent to a particular subscription *triple*: a class, instance, and recipient. A class originally denoted the type of notice, such as a message, file server notification, mail notification, etc. The instance would further categorize, such as which file server's status is being reported. The recipient may either be all users or a particular user. Every client subscribes to a set of these triples. The Zephyr servers distribute each notice to all subscribers of the relevant triple.

A notice is sent in a single UDP packet. The client assembles the message with a set of headers followed by a body. If the body is too long, messages may be fragmented across several notices and reassembled by the receiver. Outgoing notices may optionally be authenticated using Kerberos. An authenticated notice contains a Kerberos authenticator for the Zephyr service as well as a checksum keyed by the session key in the authenticator. The Zephyr servers verify the authenticator and checksum to verify the notice itself.

Largely as an artifact of its time, notices are not sent directly to the Zephyr servers. Instead, every client participating in the protocol runs a *Zephyr HostManager* or zhm. The zhm receives messages from clients running on that host and handles retransmission to the Zephyr servers, instead of leaving that to the client. When the zhm receives a notice to forward, it acknowledges with an *HMACK* (HostManager ACK). Once it has received an acknowledgement from the server, it forwards a second acknowledgement to the client, a *SERVACK* (server ACK).

Subscribing to a triple is implemented by sending notices to a special class, class `ZEPHYR_CTL`. Instead of delivering such notices, the Zephyr servers interpret them as various control messages. These control messages, in particular, include subscription and unsubscription requests. On a subscription request, the server extracts the session key from the authenticator and stores it along with the subscriptions. These are associated with the client's host and port. (On a new subscription request from that endpoint, it overwrites the previous key.)

On receipt of a notice, the server looks up all clients which are subscribed to the triple and delivers the notice to each. Notices are delivered directly to clients rather than through a zhm. On receipt of a message, the client acknowledges with a

CLIENTACK. If the client does not acknowledge a packet, the server times out and retransmits. After sufficiently many timeouts, the server assumes the client is offline and cancels the client's session.

When delivering authenticated messages to a client, the Zephyr server will look up the stored session key and include a checksum, keyed by the session key. This checksum allows the client to verify the notice came from the Zephyr servers. The client trusts that the servers correctly verified the sender's authenticator and trusts it as authentic. Note that the client cannot verify the original notice's authenticator directly; Kerberos relies entirely on symmetric cryptography, and authenticators can only be decrypted by the service they correspond to.

Figure 4-1 shows the path a notice takes when being sent from one host to another.

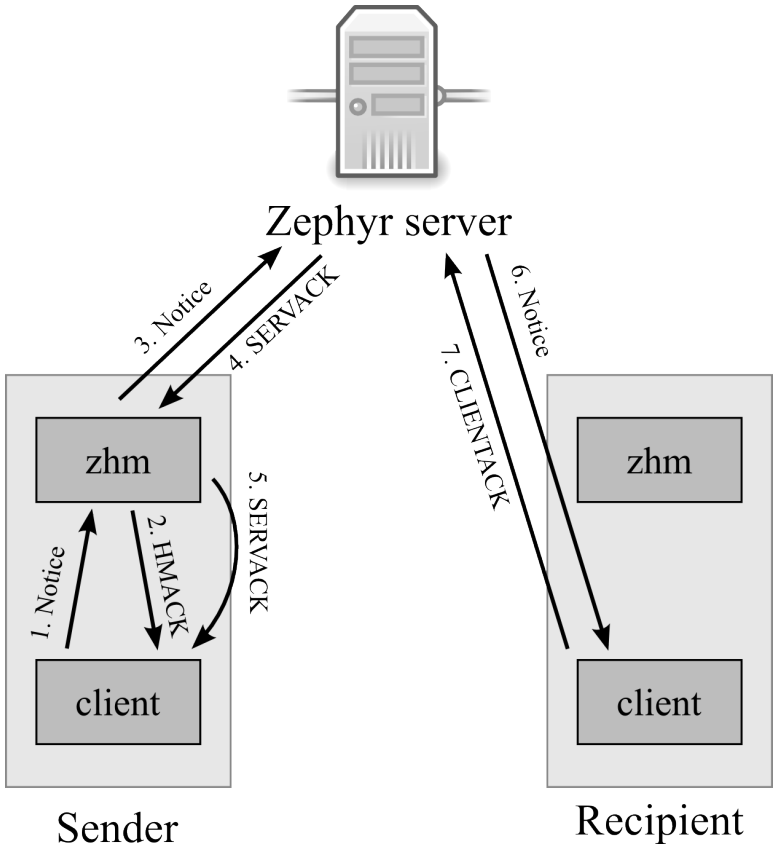

Figure 4-1: Overview of the Zephyr protocol

4.1.2 Current use

Today, Zephyr has been largely repurposed as a chat system within some social groups at MIT [42]. Classes have become analogous to chatrooms and instances topics of conversation. Most users access Zephyr using a terminal-based client called BarnOwl [2]. To continue receiving messages while offline, users run their BarnOwl sessions inside a GNU Screen session [14] on a server of their choice. This is paired with a script to regularly renew Kerberos tickets, but tickets can only be renewed for seven days, so users must still enter their password weekly.

4.1.3 Web-based clients

When *Roost* was designed, there were two Web-based Zephyr clients users could use instead of the more traditional UNIX clients. There were webzephyr [7] and ZephyrPlus [25]. However, because neither had access to the user's Kerberos credentials from within the client, they were forced to introduce alternate authentications schemes instead of being compatible with the native Zephyr one.

The older client is webzephyr. Webzephyr allows users to send personal messages to each other over Zephyr. However, it cannot subscribe to normal personal messages on users behalf. Instead, the backend subscribes to class `webzephyr`, recipient `daemon/webzephyr.mit.edu`. It interprets the instance as the intended recipient and delivers notices to that user via the Web interface. To integrate with normal Zephyr clients, webzephyr also forwards notices as native personals to the user. However, webzephyr cannot receive native personals sent by other clients.

ZephyrPlus is a more recent Web-based client. Instead of introducing an alternate notion of personal message, it just has no support for personals and only delivers public messages. The backend unions the subscription list of each user and subscribes to those triples. It then delivers the relevant subset of incoming messages to each user. However, not supporting personals still leaves some authentication issues. Messages sent from ZephyrPlus cannot be authenticated with the user's credentials. Instead, they are all sent unauthenticated with the signature field set to "via ZephyrPlus".

There is an ad-hoc mechanism where, whenever the ZephyrPlus backend sees an unauthenticated message with that signature that it did not send, it responds claiming

> The previous zephyr,
> *Message here*
> was FORGED (not sent from ZephyrPlus).

But this does not work for classes the backend does not subscribe to, and is far from Zephyr's native authentication.

The lack of Web-based clients which use native Zephyr authentication motivated us to build *Roost* and, by extension, *Webathena* as such a client would need access to Kerberos.

4.2 Design

Roost consists of a server-side component which subscribes to messages and logs them into a database. We treat public and personal subscriptions separately. Public subscriptions are deduplicated and then redistributed to users as in ZephyrPlus. This allows us to avoid storing multiple copies of a single message. We subscribe to these via a dedicated Kerberos principal on the backend. It is the handling of personal subscriptions that makes *Roost* unique.

For each user, *Roost* runs an *inner daemon*, a small process which handles personal subscriptions and sending messages for that user. All operations on inner daemons require the user to staple Zephyr credentials. Inner daemons forward any received messages back to the main server which then writes them to the database. Outgoing messages also go through the inner daemon for authentication. Figure 4-2 shows the possible paths for a message being delivered to *Roost*.

The server spawns inner daemons on-demand as it receives zephyr credentials from users. Users may be offline for long periods of time, so inner daemons must continue functioning on expired tickets. To preserve Zephyr sessions across server restarts and code pushes, we serialize all inner daemon session state and restore it on startup. It

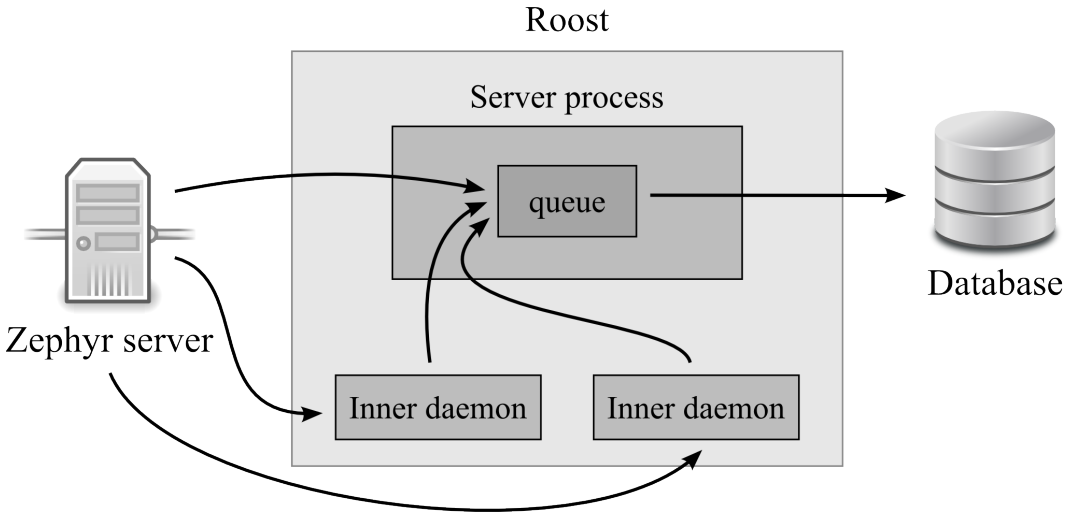

Figure 4-2: Messages being delivered to *Roost*. Arrows show paths of messages traveling between components.

takes around 20 minutes after missing a *CLIENTACK* for the Zephyr servers to time out a session, so this preserves inner daemon sessions in most cases.

Clients communicate with *Roost* using an HTTP and WebSockets API. This allows Web content to access the server directly instead of requiring the proxy schemes of our sample applications. Other platforms do not have these restrictions, but HTTP libraries are readily available, so implementing non-Web clients should be equally natural.

GSSAPI is used to authenticate to the server itself. The client passes an initial GSSAPI context token in a `POST` request. The server, if it accepts the authenticator, returns an access token. To reuse existing standards, we use OAuth's Bearer authentication scheme [24] to present this access token in requests. This token lasts much longer than the user's ticket so users need not to re-enter their password as frequently.

4.3 Implementation

4.3.1 Subscriber

We implement *Roost*'s backend using Node.js [34]. This allows us to reuse *Webathena*'s ASN.1 implementation, but primarily it seems fitting when the rest of the

work in this thesis is also in JavaScript. It is hosted on XVM [43] and sql.mit.edu [31], SIPB's virtual machine and database services, respectively.

At the core, *Roost* consists of a queue for messages. This queue globally orders all received messages and inserts them into the database. The queue receives messages from public subscriptions as well as any running inner daemons. At the other end of the queue, we insert messages into the database in order. At insertion time, we compute which users should see a given message by querying the subscriptions table and log that as well. We take care to use the same algorithm for matching subscriptions as the real Zephyr servers, including normalization of class names.

Messages are globally ordered in a single messages table on the database. We maintain a many-to-many relation between it and the users table. Although this does require inserting many rows per message for popular classes, it allows MySQL to optimize read queries. We considered complex designs which computed a user's view based on historical subscription data, but we were unable get MySQL to index those queries effectively.

4.3.2 Changes to *libzephyr*

Instead of reimplementing the Zephyr protocol in JavaScript, we wrote Node.js bindings for *libzephyr*, the C implementation. This required several changes to fix bugs and provide new features in the library. The first difficulty was that Node.js uses an asynchronous single-threaded evented architecture while many of *libzephyr*'s functions block on the network. We use an alternate version of the `ZSendNotice` call, `ZSrvSendNotice`, which takes a custom callback for sending packets. We pass a callback which simply returns all packets to JavaScript. We then send them to the zhm asynchronously via Node's UDP implementation and correlate the acknowledgements ourselves. This required fixing a bug in *libzephyr* in the interaction between *HMACK*s and the fragment reassembly code[1].

For subscriptions, we expose a new call in *libzephyr* to provide alternate versions

[1]We pause to note for amusement that this bug and others fixed in Zephyr are older than the author of this paper.

of `ZSubscribeTo` and other subscription functions with a send callback parameter, analogous to `ZSrvSendNotice`. But, as the number of public subscriptions grew, we saw failures. As of writing, the server now subscribes to over 26,000 distinct triples on startup[2]. First, we overflowed the operating system's send buffer and lost packets to the zhm. (Despite using UDP, `libzephyr` assumes messages sent to the local zhm are reliable.) We also hit a bug where the zhm breaks checksums on retransmitted packets. To resolve these, we wait asynchronously for *HMACK*s between packets and run a beta version of the zhm for a fix to the retransmit bug.

Roost required further changes to *libzephyr* to improve the behavior of our inner daemons. The Zephyr servers continue delivering and checksumming notices to a session long after the credentials that created it have expired. But *libzephyr* never remembers the key for the checksums. Rather, it queries the user's credential cache each time it verifies one. First, this means that messages fail to authenticate after the cache changes if subscriptions are not refreshed. Second, when credentials expire, it no longer finds the key and messages again fail to authenticate. We fix this by patching *libzephyr* to save all keys from subscription notices. We then add a conservative heuristic for when to retire them from memory. Among many deficiencies in the Zephyr protocol is that it is not explicit which key is current on the server, so we resort to heuristics based on received messages.

Since we expected to regularly deploy new versions of *Roost*, especially in early development, it was important to maintain inner daemons across code pushes. UDP is connectionless, so we simply never tear down sessions for inner daemons. We patch *libzephyr* to include two functions `ZDumpSession` and `ZLoadSession`. These, respectively, serialize and restore the port number and all keys of the Zephyr session. *Roost* regularly requests the inner daemons to serialize their state. On startup, this state is loaded up again. Provided our uptime is such that the servers never expire our sessions, we can retain inner daemon sessions across code pushes and even reboots. To detect when a session has expired, we regularly ping inner daemons by sending to a special triple and expecting the notice to come back authenticated.

[2]This largely the work of one or two users rather than a reflection of *Roost*'s activity.

4.3.3 Public API

Clients query *Roost* using an HTTP and WebSockets-based API. Authentication is based on GSSAPI. The client *POST*s a GSSAPI token and the server responds with a bearer token good for a month. Note that, although GSSAPI in general may require multiple exchanges to tokens to fully establish a context, the Kerberos mechanism requires only one exchange. This simplifies the server in that we do not need to maintain incomplete contexts as state. To implement this, we wrote bindings to the system GSSAPI libraries. That we can use these bindings with *ctlfish*'s JavaScript version further demonstrates the interoperability between our various Kerberos implementations.

The primary call in the *Roost* API is the message querying call. It takes an anchor point, a count of messages, a direction (forwards or backwards), and an optional filter. This queries the database for the next batch of messages after the anchor, subject to the filter. The query also has a WebSocket-based version, which, as in *ctlfish*, we implement with SockJS [39] for compatibilty with older browsers. The WebSockets variant streams messages in realtime if the anchor point is near the end. The count may be increased as the user scrolls. We calls these queries *tails*. They start from an anchor point and stream messages starting from that anchor.

Although messages are globally ordered, we do not send their indices to the client. This is to avoid leaking information about messages received by other users. Instead, messages are sealed by encrypting the counter a single AES block with some secret key. We chose this scheme just to avoid maintaining a mapping in the database from opaque identifiers to indices. This does have the side effect that clients cannot compare two identifiers, but this has not proved too problematic.

To allow hosting Web-based clients on different origins from the server, we use Cross-Origin Resource Sharing [26] (CORS) and allow any origin to access the API. Access tokens are manually attached via OAuth bearer tokens [24] rather than stored in cookies, so there is no danger of the usual Cross-Site Request Forgery attacks. As a practical measure, we avoid any parts of CORS which require a preflight request. This

is both for performance on browsers which support CORS and to support Internet Explorer 9. Internet Explorer 9 supports a predeceder of CORS, *XDomainRequest* [28]. XDomainRequest is much more limited and cannot use features which would require a preflight in CORS, such as custom headers or methods. We do not support Internet Explorer 8 and earlier.

4.3.4 Clients

We implement two clients: an official Web-based interface to the service and an small script to import subscriptions stored on Athena.

The Web interface is intended to be a full-featured Zephyr client. As of writing, the user interface is still incomplete, but it is functional enough to have largely replaced BarnOwl in the author's usage of Zephyr. The backend and Kerberos portions are fully functional. The interface consists only of static files, hosted on scripts.mit.edu [38]. It is deployed at `https://roost.mit.edu`. We use the tail APIs in the server to implement a bidirectional infinite scroll. As the user scrolls up, we request more messages from the top and retire messages from the bottom. As the user scrolls down, we request more from the bottom and retire from the top. This gives the appearance of scrolling through the user's entire message list while only keeping a constant window in memory.

The import script is implemented in Python. Instead of using custom Kerberos and GSSAPI code, we use the native system Kerberos and GSSAPI libraries. These are accessed with the `ctypes` module. The import script parses the user's subscriptions stored on Athena and uploads them to *Roost*, along with zephyr credentials. We host the import script in our Athena locker, as is the standard practice for software on Athena.

We believe this import script demonstrates an important point about *Roost*. The server is not specific to *Webathena*. It is a Kerberos-based service which authenticates via GSSAPI like many other services and is accessed via the platform's native Kerberos libraries. The Web has none, so we provide one in *Webathena* for the official client, but a Python client using MIT Kerberos libraries can access *Roost* just as

naturally as the Web interface does.

4.3.5 Summary

We summarize *Roost*'s many components in table 4.1.

Component	**Description**	**Language**	**Lines of Code**
node-zephyr	Node.js bindings for libzephyr	C++, JavaScript	1074
node-gss	Node.js bindings for GSSAPI	C++, JavaScript	1011
roost	*Roost* server component	JavaScript	4319
roost-client	Web-based *Roost* client	JavaScript	4843
roost-python	Import tool and support code	Python	1089

Table 4.1: *Roost* components. Line counts do not include third-party code or code pulled from *Webathena*.

Chapter 5

Related Work

5.1 OAuth

Many services deployed natively on the Web today have similar systems for delegating access to other applications. Many of them, such as those run by Google, Facebook, and Dropbox, use the OAuth [19] authorization framework. OAuth provides a mechanism for a client to receive an opaque *access token* for a particular resource server. This token is only given after user permission and is scoped to certain interactions with the resource. OAuth also introduces a *refresh token* which may be exchanged for renewed access tokens when persistent access to a service is needed.

The design of OAuth deeply influenced *Webathena*. However, we opted not to use OAuth itself as we were interested in delegating credentials between purely client-side Web applications. OAuth makes extensive use of HTTP redirects which would inadvertently send information to servers hosting our applications. We plan to investigate adapting Google's `postMessage`-based OAuth flow [16] for a future iteration of *Webathena*.

5.2 HTTP Negotiate

The HTTP Negotiate authentication mechanism [23] provides GSSAPI-based authentication to HTTP servers. It uses the SPNEGO [45] GSSAPI mechanism which

negotiates the use of another GSSAPI mechanism. HTTP Negotiate is deployed at MIT for access to some services, but browser support is imperfect.

We considered adopting it for *Roost*, but it is not suitable for use in JavaScript. Using SPNEGO requires additional token exchanges to establish a context, but Negotiate simply associates the incomplete GSSAPI contexts with the TCP or TLS connection. Browser APIs do not provide control over how requests map to connections, so any scheme which associates state with the connection itself is problematic.

5.3 scripts.mit.edu

SIPB's scripts.mit.edu [38] service allows members of the MIT community to host dynamic websites out of their Athena file locker. Like our work, it needs limited access to the user's services on Athena, namely their files. Athena uses the Andrew File System (AFS) which provides mechanisms for access controls. Rather than delegate tickets, scripts.mit.edu provides sign-up utilities which gives the service access to certain directories in the user's locker.

This mechanism is specific to AFS, one of many different services accessible by Kerberos, but demonstrates another means of access delegation on Athena.

5.4 Zulip

While *Roost* was being conceived and implemented, several MIT alumni began a startup, Zulip, to build a Zephyr-like messaging system for internal use at companies. To test their product, they provide Zephyr integration for MIT use. Zulip handles public subscriptions similarly to ZephyrPlus and *Roost*. For personal subscriptions and authenticating outgoing messages, it requires users to run a *mirroring script* to forward personals and send authenticated notices. It must be run on a long-running dialup server and supplied up-to-date Kerberos tickets. This mirroring script is analogous to *Roost*'s inner daemons. But because we can delegate credentials, our service can run the mirroring script equivalent internally.

Chapter 6

Discussion and Future Work

To evaluate *Webathena* itself, we demonstrated how to use it to build applications which access existing Kerberos services at MIT. These have been detailed extensively in chapters 3 and 4. But more generally, by adapting Kerberos to the Web, we introduced a new model for Kerberos for untrusted applications, giving it security properties different from a traditional Kerberos deployment. We now discuss our overall Web-based Kerberos ecosystem as well as possible directions for future improvement.

6.1 Client code integrity

Deploying *Webathena* as normal Web application allowed for maximum compatibility with existing Web browsers, but we now inherit the security properties of a Web application. The browser regularly downloads JavaScript from the Internet, so we must trust our server, namely scripts.mit.edu, as well as SSL and the CA system to be sure we are running the correct JavaScript code. This is unavoidable in the Web today, although schemes such as public key pinning [27, 11] mitigate this somewhat.

In contrast, a native Kerberos implementation is less reliant on SSL and online servers. While the initial download will anchor its trust eventually on SSL or a similar public key infrastructure, this trust is limited to the initial download. From there, updates can be signed by offline keys specific to Kerberos. The Web lacks this

statefulness and must trust SSL on every visit.

Instead of a Web application, we could deploy *Webathena* as a browser extension, like a Chrome extension [17] to augment the existing deployment. (The existing deployment should be kept for maximum compatibility.) Browser extensions are installed locally and updated by the browser vendor. In the case of Chrome extensions, updates are also signed by an extension-specific key. One difficulty in deploying an extension would be allowing other applications to discover the installed extension. Unlike other applications which take this approach, such as Cryptocat [5], *Webathena* provides an external API for other websites.

Alternatively, instead of abandoning Web applications, browsers could be modified to provide stronger code signing. *Webathena*'s client consists entirely of static content. We could sign this content offline with some dedicated keypair. The browser would be given the public half using a stateful mechanism, similar to public key pinning [11]. The server could include our key in a header which the browser would remember for the origin. Subsequent accesses would require a signature from this key. This limits our reliance on the CA system and even the live server to the first visit. However, as in key pinning, a misplaced code-signing header would disable *Webathena* in all supporting browsers.

6.2 Credential delegation

Although *Webathena* loses some security by being a Web application, we *gain* in security with our credential delegation scheme. Unlike their native counterparts, *Webathena* applications are not trusted with the user's full credentials. Programs like the official remctl client, BarnOwl, etc., have access to not only the host system, but also the user's TGT. A vulnerability in BarnOwl can be exploited to gain access to the user's files, the groups they administer, their email, and all other services on Athena. The *Webathena* counterparts are much more limited. *Roost* is, by necessity, trusted with zephyr credentials, but has no access to files or other services. Likewise, *ctlfish* requires user permission to access each remote machine individually, so the

user can limit its access.

6.2.1 Android

Webathena's delegation scheme could be extended to other platforms, such as Android. Android applications may access low-level networking functions, so we have no need of our proxies, but, like the Web, Android applications run in a sandboxed environment. The MIT Kerberos Consortium has begun a port of Kerberos to Android, but it is not yet usable as a complete solution for applications.

We propose that the Kerberos for Android implement a similar trusted credential manager as *Webathena*. The Android platform provides an `AccountManager` [15] framework which allows applications to implement custom account types. Account managers expose an interface similar to OAuth and may present the user with permissions prompts. *Webathena*'s semantics could be mapped onto these, thus providing users with access to Kerberos-authenticated services on their phone with the security benefits of our delegation scheme.

6.2.2 Finer-grained delegation

While our scheme is an improvement over a traditional Kerberos library, we can only delegate access at a service-level granularity. This is appropriate for Zephyr, but more complex services may benefit from finer-grained delegation. For instance, delegating tickets to AFS gives access to all of a user's files on Athena, while an application may only need access to a single file or directory or perhaps only read access without write access.

We could solve this by using the *AuthorizationData* field in the encrypted portion of a Kerberos ticket. This is an extensible field for adding restrictions to a ticket. When requesting a service ticket, clients may request restrictions be appended to those in their TGT. We would introduce a new AuthorizationData type to act as an equivalent to the scope parameter in OAuth. Services would interpret this field and limit access accordingly. For instance, Moira could interpret it as whether the client

may administer or only query a user's mailing lists, depending on this field.

One difficulty would be in displaying this scope to the user. *Webathena* could be hard-coded to interpret the service-specific scopes for services it knows about. Unknown restrictions would not be displayed and prompt for stronger access than will actually be given. Alternatively, we could standardize a scheme for *Webathena* to communicate with the services about how to present these to the user.

6.3 Native Kerberos integration

Although *Webathena* can acquire Kerberos tickets directly in the browser, many users may access services using native Kerberos libraries outside the browser as well. For instance, a user on an Athena cluster machine already has Kerberos tickets in their login session. In addition, some security-conscious users expressed discomfort at entering passwords into a browser window. Allowing *Webathena* to integrate with the host system's credential store would thus be valuable.

As Web content cannot access the host system's files, this would require exposing the host credential cache to the browser. Some possibilities include a unsandboxed native-code browser plugin, browser extensions, or a local HTTP server which somehow communicates securely with *Webathena*'s web content. The last option may be problematic on multi-user machines as other users may communicate with the server as well.

Alternatively, we could also provide a script which serializes the user's TGT in a format suitable for pasting into a form on *Webathena*. However, this is unlikely to be usable by anyone other than advanced users.

6.4 Persistent access and revocation

We believe ticket lifetimes in Kerberos are not ideal for today's computing environment. As deployed at MIT, Kerberos limits ticket lifetimes to just under a day (specifically, 21 hours and 15 minutes) and may be renewed up to one week. This

causes *Webathena* some difficulties.

Several users of *Roost* commented that *Webathena* requests the user's password too frequently. We currently do not implement ticket renewal, so users must enter their passwords daily to send messages. When Kerberos was designed, users were expected to use various cluster machines, log in for a short session, and log back out. A lifetime of one day was perfectly adequate. Today, people have personal machines and month-long login sessions are feasible.

Delegating persistent access to a service can also be difficult in *Webathena*. We achieve it in *Roost* by serializing Zephyr sessions, but by using what is arguably a flaw in the protocol. The Zephyr servers should expire sessions when the tickets that create them expire. It is also impossible to revoke access if a user decides they no longer trust *Roost* with their Zephyr credentials.

6.4.1 Ticket renewal

Renewing tickets would increase decrease password prompts to weekly, matching BarnOwl usage, but this is still frequent compared to login forms on the Web. Moreover, tickets may only be renewed while they are unexpired, so the user must visit *Webathena* daily so that the renewal code can run. As it is only a login portal, this is unrealistic. One possibility is to include a page for *Roost* to embed in an invisible `iframe` which repeatedly renews tickets, however this still assumes the user visits *Roost*, or some other *Webathena* application, regularly. We could also investigate using Kerberos' postdated ticket mechanism and request enough to cover the entire window. This does, however, defeat the purpose of ticket renewal.

Moreover, even with ticket renewal, we could not delegate persistent access to services to applications. Although a long-running server like *Roost* can easily renew tickets regularly, this still only lasts for a week. Requiring the user to visit *Roost* weekly or lose subscriptions, in a hypothetical Zephyr which more readily expired sessions, would also be unacceptable.

6.4.2 Ticket revocation

The simplest solution would be to significantly increase the lifetime of Kerberos tickets, or at least their renewal time, but this lifetime is too closely tied to revocation in Kerberos. If a user changes their password, the old one can no longer be used to acquire tickets. *However, existing tickets continue to be usable.* Kerberos conflates two parameters: how often must the user enter their password and how long before a password revocation propagates.

For service tickets, it is reasonable that they are difficult to revoke. Using a service ticket does not require either party to communicate with the KDC. This is a convenient property and, more importantly, we cannot change this without breaking existing services. However, exchanging a TGT for a service ticket by necessity involves the KDC. It is very natural to check for revocation here, but tickets do not contain information about which version of the password produced them.

We propose to add a new AuthorizationData type to Kerberos which contains the version number of the client's password which was used to create this TGT. This restriction would mean that the ticket is only valid as long as that version number was up-to-date. The KDC, when acting on a TGT or renewing a service ticket would refuse to honor the request if the password had since been changed.

With this change, TGT lifetimes may safely be increased without affecting revocation. In fact, we would *decrease* the propagation time for revocation. Tickets can be renewed, so it is possible for a compromised password to remain in use for up to a week. In our scheme, because renewals also get revocation checks, the propagation time is only one day. Note that service ticket lifetimes must still be limited as their uses do not check for revocation. We can, however, increase their renewal time and thus provide a way to delegate long-lived credentials for *Roost* and other applications.

6.5 Constrained ticket-granting-tickets

Finally, we propose another scheme for delegating persistent access that we believe fits more naturally into Kerberos. While infinitely renewable service tickets would

allow persistent delegation of credentials, they must be regularly renewed before they expire. Servers can be expected to be online, so this is not a serious constraint, but it does impose liveness requirements on all applications. More fundamentally, there is an asymmetry in cloud applications using renewable service tickets for persistent access and the user using a TGT for persistent access.

We propose instead to reuse the mechanism proposed in 6.2.2 to constrain TGTs as well. We call these *constrained TGTs*. A constrained TGT can only request tickets for certain services, along with an instance of those services' own constraints. We would then implement all persistent delegation in *Webathena* as forwarding TGTs constrained to the services requested.

Constrained TGTs unify *Webathena*'s service-ticket-based credential delegation with Kerberos as used today. Servers acting on behalf of a user now act identically to a user's native login session. It would allow us to seamlessly reuse all the existing Kerberos infrastructure for delegating unconstrained TGTs and provide the security benefits we gain in *Webathena*. A user may run BarnOwl on a remote dialup server, but forward a TGT constrained to Zephyr and any other Athena services needed by BarnOwl. With *no changes to existing services*, we will have transplanted *Roost*'s security properties to BarnOwl while remaining compatible with SSH credential delegation and existing client software for consuming tickets.

Chapter 7

Conclusions

In this thesis, we have presented *Webathena*, an adaptation of the Kerberos network authentication protocol to the Web. Using proxies, *Webathena* implements a native Kerberos client for the browser as a standard Web application without requiring additional capabilities. In doing so, we design a new model for limited credential delegation in Kerberos to better serve today's computing environments.

We demonstrate how *Webathena* can be used with existing Kerberos services at MIT by building simple clients for Zephyr, SSH, and remctl. Then, in *Roost*, we build a new service which both interacts with an existing Kerberos service and, while independent of *Webathena* itself, is designed with an eye for native Web compatibility. These clients improve upon the security of the traditional Kerberos ecosystem by restricting access to the user's credentials. Moreover, we achieve this without modifications to browsers or existing Kerberos infrastructure.

Finally, we propose modifications to the Kerberos protocol to better unify our new model with existing Kerberos infrastructure.

Appendix A

Webathena API

The Webathena API contains a single API call, accessed using Mozilla's WinChan library, at the URL `https://webathena.mit.edu/#!request_ticket_v1`. The `relay_url` parameter to WinChan is `https://webathena.mit.edu/relay.html`. It takes a parameters dictionary with a single key, `services`. This key contains an array of service principals the that the website is requesting. Each service principal is a dictionary containing two keys:

`realm` The realm of the service as a string.

`principal` The principal name of the service. This is specified as an array of strings, each containing a component of the principal name, pre-parsed.

As an example, the principal `HTTP/roost-api.mit.edu@ATHENA.MIT.EDU` would be represented as

```
{
  "realm": "ATHENA.MIT.EDU",
  "principal": [
    "HTTP",
    "roost-api.mit.edu"
  ]
}
```

Webathena responds with a dictionary containing a key `status`. It has three possible values:

OK The user approved the permission grant. The dictionary contains a second key, `sessions`, with Kerberos service sessions corresponding to each of those requested.

DENIED The user did not approve the permissions grant.

ERROR The request was malformed.

Service sessions are included as JSON objects with the following keys, made to mirror the ASN.1 definitions of the `KDC-REP` and `EncKDCRepPart` structures in the Kerberos protocol.

crealm The client realm as a string.

cname The client principal name as a Kerberos `PrincipalName`.

ticket The ticket as a Kerberos `Ticket`.

key The session key as a Kerberos `EncryptionKey`.

flags The ticket flags as an array of ones and zeros.

authtime The ticket authtime as a JavaScript Date.

starttime Optional; the ticket starttime as a JavaScript Date.

endtime The ticket endtime as a JavaScript Date.

renewTill Optional; the ticket renew-till time as a JavaScript Date.

srealm The server realm as a string.

sname The server principal name as a Kerberos `PrincipalName`.

caddr Optional; The ticket's host addresses as a Kerberos `HostAddresses`.

Where reference is made to ASN.1 structures in Kerberos, we use a straightforward serialization of the ASN.1 structure into JSON. Primitive types map to their corresponding JSON type. We encode `OCTET STRING` in base64. A `SEQUENCE` maps to a dictionary, and `SEQUENCE OF` maps to an array. We refer to the `web_scripts/js/krb_proto.js` file in the *Webathena* source tree for our canonical versions of these structures.

The `web_scripts/js/webathena.js` library in the *Webathena* source tree includes a `krb.Session.fromDict` API which wraps this object in a convenience class and implements various functions for use it with. We recommend client applications needing to interpret the ticket use this library along with its dependencies, also found in *Webathena*'s source tree.

q.min.js The Q promises library

sjcl.js The Stanford Javascript Crypto Library

tripledes.js The `tripledes.js` "rollup" bundle in CryptoJS. We use the MD5 and DES implementations in CryptoJS for some legacy cipher profiles in Kerberos.

For the functions provided in `webathena.js`, we refer to the `web_scripts/js/krb.js` file in the *Webathena* source tree.

Webathena also includes a GSSAPI implementation, `web_scripts/js/gss.js`, which builds on `webathena.js`.

Appendix B

Sample Code

Below is a code sample from `ctlfish`, demonstrating the use of the cross-origin *Webathena* API. It uses the Q promises library to simplify asynchronous programming, but this is not necessary to use *Webathena*.

```
var ccache = { };
function getCredential(peer) {
    var key = peer.principal.toString();
    if (ccache[key])
        return Q.resolve(ccache[key]);

    var deferred = Q.defer();
    WinChan.open({
        url: WEBATHENA_HOST + "/#!request_ticket_v1",
        relay_url: WEBATHENA_HOST + "/relay.html",
        params: {
            services: [{
                realm: peer.principal.realm,
                principal: peer.principal.principalName.nameString
            }]
        }
```

```
    }, function (err, r) {
        if (err) {
            deferred.reject(err);
            return;
        }
        if (r.status !== "OK") {
            deferred.reject(r);
            return;
        }
        var session = krb.Session.fromDict(r.sessions[0]);
        ccache[key] = session;
        deferred.resolve(session);
    });
    return deferred.promise;
}
```

Appendix C

Testimonials

For additional clarity and insight into this work, we present here a number of testimonials gathered during discussions of the design and implementation of *Webathena*, reproduced in their original form.

> “You may have noticed that davidben is above-average crazy even for Zephyr :)”
>
> *Nelson Elhage*

> “Why . . . are you doing this”
>
> *Nelson Elhage*

> “Please tell me you’re not trying to implement kerberos in Javascript”
>
> *Nelson Elhage*

> “You have to admit it’s something davidben would do.”
>
> *Adam Glasgall*

> “You have . . . interesting . . . ideas of ‘fun’.”
>
> *Benjamin Kaduk*

> I'm torn between congratulating you and worrying about your mental health.
>
> *Adam Glasgall*

> it has LEGS and it's LOOKING AT ME
>
> *Alioth Drinkwater*

> Congrats, and you're insane.
>
> *Nelson Elhage*

> okay, less trolling davidben, moar coding
>
> *Geoffrey Thomas*

> Don't give him more ideas, man.
>
> *Adam Glasgall*

> The great thing about davidben is that if you leave him alone long enough, he just starts trolling himself.
>
> *Nelson Elhage*

> What are you... maybe I shouldn't ask.
>
> *Alan Huang*

> As a wise man once said: "Congratulations, and you're insane."
>
> *Alan Huang*

> If I didn't already think you were dangerously insane...
>
> *Adam Glasgall*

> I thought the general consensus was that you were already insane, so.
>
> *Geoffrey Thomas*

> That's an interesting definition of "fun" you have there.
>
> *Adam Glasgall*

> Kerberos in Javascript sounds slightly insane.
>
> *Benjamin Tidor*

> davidben is slightly insane.
>
> *Alex Dehnert*

> As mentioned before, davidben is slightly insane.
>
> *Adam Glasgall*

> . . . did you just implement gssapi in javascript?
>
> *Adam Glasgall*

> . . . david I worry about your sanity
>
> *Di Liu*

> Please don't tell me this is going to end up with kerberized webathena jabber
>
> *Alex Chernyakhovsky*

> please tell me that is made up
>
> *Joshua Pollack*

> davidben is gloriously insane.
>
> *Adam Glasgall*

> David, what are you doing?
>
> *Adam Glasgall*

> David's so insane, he's on there *twice*.
>
> *Alan Huang*

> All right, I am ready to start acknowledging your insanity.
>
> *Victor Vasiliev*

> David is insane.
>
> *Jennifer Wang*

> . . . um
>
> *Geoffrey Thomas*

> hold on, my brain is exploding
>
> *Justin Dove*

> Oh my god, this thing does work.
>
> *Victor Vasiliev*

> i dare you to incldue that.
>
> *Cassandra Xia*

Bibliography

[1] J. M. Arfman and P. Roden. Project Athena: supporting distributed computing at MIT. *IBM Syst. J.*, 31(3):550–563, June 1992.

[2] barnowl. `https://barnowl.mit.edu/`.

[3] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

[4] Adam Barth, Daniel Veditz, and Mike West. Content Security Policy 1.1. Working draft, W3C, June 2013. `http://www.w3.org/TR/2013/WD-CSP11-20130604/`. Latest version available at `http://www.w3.org/TR/CSP11/`.

[5] Cryptocat. `https://crypto.cat/`, 2013.

[6] David Dahl and Ryan Sleevi. Web Cryptography API. Working draft, W3C, June 2013. `http://www.w3.org/TR/2013/WD-WebCryptoAPI-20130625/`. Latest version available at `http://www.w3.org/TR/WebCryptoAPI/`.

[7] Jeremy Daniel. webzephyr.mit.edu. `http://webzephyr.mit.edu/`.

[8] Debathena. `http://debathena.mit.edu/`.

[9] C. Anthony Dellafera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld. The zephyr notification service. In *Proceedings of Winter 1988 Usenix Technical Conference*, pages 213–220, 1988.

[10] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[11] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. `http://tools.ietf.org/html/draft-ietf-websec-key-pinning-08`, July 2013.

[12] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[14] Free Software Foundation, Inc. GNU Screen — GNU Project — Free Software Foundation. http://www.gnu.org/software/screen/, February 2010.

[15] Google, Inc. Creating a Custom Account Type — Android Developers. http://developer.android.com/training/id-auth/custom_auth.html.

[16] Google, Inc. OAuth PostMessage Flow. https://code.google.com/p/oauth2-postmessage-profile/.

[17] Google, Inc. What are extensions? — Google Chrome. http://developer.chrome.com/extensions/, 2013.

[18] Markus Gutschke. Shell In A Box. https://code.google.com/p/shellinabox/.

[19] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[20] Ian Hickson. HTML Standard. Technical report, WHATWG, August 2013. http://www.whatwg.org/specs/web-apps/current-work/multipage/.

[21] Lloyd Hilaiel. WinChan. https://github.com/mozilla/winchan.

[22] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), November 2012.

[23] K. Jaganathan, L. Zhu, and J. Brezak. SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows. RFC 4559 (Informational), June 2006.

[24] M. Jones and D. Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750 (Proposed Standard), October 2012.

[25] Gurtej Kanwar, Gary Wang, Michael Wu, Timothy Yang, and Ernest Zeidman. ZephyrPlus! https://zephyrplus.mit.edu/.

[26] Anne van Kesteren. Cross-Origin Resource Sharing. Candidate recommendation, W3C, January 2013. http://www.w3.org/TR/2013/CR-cors-20130129/. Latest version available at http://www.w3.org/TR/cors/.

[27] Adam Langley. Public key pinning. https://www.imperialviolet.org/2011/05/04/pinning.html, May 2011.

[28] Eric Lawrence. XDomainRequest — Restrictions, Limitations, and Workarounds. http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx, May 2010.

[29] J. Linn. Generic Security Service Application Program Interface Version 2, Update 1. RFC 2743 (Proposed Standard), January 2000. Updated by RFC 5554.

[30] Peter Marshall. remctl. http://www.eyrie.org/~eagle/software/remctl/.

[31] MIT SIPB MySQL Service for Athena. http://sql.mit.edu/.

[32] Jeff Mott. CryptoJS. https://code.google.com/p/crypto-js/.

[33] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806.

[34] node.js. http://nodejs.org/.

[35] K. Raeburn. Advanced Encryption Standard (AES) Encryption for Kerberos 5. RFC 3962 (Proposed Standard), February 2005.

[36] K. Raeburn. Encryption and Checksum Specifications for Kerberos 5. RFC 3961 (Proposed Standard), February 2005.

[37] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. http://tools.ietf.org/html/draft-ietf-websec-x-frame-options-07, July 2013.

[38] The SIPB scripts.mit.edu project. scripts.mit.edu. http://scripts.mit.edu/.

[39] SockJS. http://sockjs.org.

[40] Emily Stark, Mike Hamburg, and Dan Boneh. Stanford Javascript Crypto Library. http://crypto.stanford.edu/sjcl/.

[41] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Usenix Conference Proceedings*, pages 191–202, 1988.

[42] Student Information Processing Board. Using Zephyr (a.k.a. Zephyr For Dummies). http://sipb.mit.edu/doc/zephyr/.

[43] XVM — Virtual Servers for MIT. http://xvm.mit.edu/.

[44] L. Zhu, K. Jaganathan, and S. Hartman. The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2. RFC 4121 (Proposed Standard), July 2005. Updated by RFCs 6112, 6542, 6649.

[45] L. Zhu, P. Leach, K. Jaganathan, and W. Ingersoll. The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism. RFC 4178 (Proposed Standard), October 2005.