SPLIT AUTH: A DECENTRALIZED SYSTEM FOR WEB AUTHENTICATION

by

Thomas Parisi

A THESIS

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE - CYBERSECURITY

Thomas Parisi, Candidate

ADVISORY COMMITTEE

Antonio Nicolosi, Advisor

Susanne Wetzel, Reader

STEVENS INSTITUTE OF TECHNOLOGY
Castle Point on Hudson
Hoboken, NJ 07030
2012
SPLIT AUTH: A DECENTRALIZED SYSTEM FOR WEB AUTHENTICATION

ABSTRACT

Split Auth is a software distribution for identity management at the Internet scale. To facilitate interoperation with existing Internet protocols, Split Auth is built on top of OpenID, an open standard that describes how users can be authenticated in a decentralized manner. Most OpenID deployments only use a password to authenticate a user. This can be weak when users pick their own passwords. Split Auth seeks to fix the inherent password problems by using public key cryptography and digital signatures. For usability, the system does not completely eliminate password, but reduces the number of passwords that a user needs to remember from \( n \) (the number of different websites that require user authentication) to 2, the number of passwords required by Split Auth. By reducing the number of passwords needed to remember, the complexity of the passwords can be increased. By increasing the password complexity, the security of the overall system is increased.

Author: Thomas Parisi
Advisor: Antonio Nicolosi
Date: November 26, 2010
Department: Computer Science
Degree: Master of Science
Acknowledgments

I would like to acknowledge my family for supporting me and my advisors Professor Antonio Nicolosi and Professor Susanne Wetzel for their guidance.
Table of Contents

Abstract iii

Acknowledgments iv

List of Figures ix

1 Introduction 1
   1.1 Problem Statement 1
   1.2 Problem Scope 5
   1.3 Thesis Approach 6
   1.4 Outline of Thesis Work 7

2 Background 10
   2.1 OpenID 10
      2.1.1 OpenID Security 13
   2.2 Identity Management 16
      2.2.1 Isolated User Identity Model 17
      2.2.2 Federated User Model 17
      2.2.3 Centralized Models 17
      2.2.4 User Centric Identity Management 18
      2.2.5 Shibboleth - A Federated Model Example 18
   2.3 Kerberos - A Centralized Model Example 20

3 Design Overview 21
   3.1 Design Guidelines 21
3.1.1 Keep Password Off Server 22
3.1.2 Limit the Number of Passwords 23
3.2 Threat Model 24
3.3 Stakeholders in OpenID 25
3.4 Architecture of the Split Auth System 27
  3.4.1 Identity Provider 27
  3.4.2 Split Auth Server 30
3.5 Operational Overview 30

4 Design Details 32
  4.1 Cryptographic Details 32
    4.1.1 Schnorr Signature 32
    4.1.2 2Schnorr 33
    4.1.3 Signing A Message 33
  4.2 Client Interaction 34
    4.2.1 Logging Into Split Auth 34
    4.2.2 Resetting the Keys 34
  4.3 IdP Configuration File 35
  4.4 IdP 35
    4.4.1 Authorize_Mode 35
    4.4.2 Validation_Mode 36
  4.5 Key Reset 36
  4.6 Register Script 36
  4.7 Split Auth Server 37
  4.8 Registering with the Split Auth Server 37
    4.8.1 Support Files 38
4.8.2 Javascript Random Number Generator 38

5 Implementation Details 39
5.1 PHP 39
5.2 GMP 39
5.3 PHPMyID 40
5.4 AJAX 40
5.5 Apache 41
5.6 Javascript 41
5.6.1 Javascript Cryptographic Libraries 41
5.6.2 New Pseudorandom Number Generator 42

6 Experimental Evaluation 44
6.1 Benchmarks 44
6.1.1 Software Setup 44
6.1.2 Logging In 44
6.1.3 Updating Keys 45
6.2 End To End 45
6.3 User Evaluation 46

7 Discussion and Conclusion 47
7.1 Discussion 47
7.2 Conclusion 48

Appendices 47

A IdP Details 49

B Glossary 55
Bibliography
List of Figures

2.1 OpenID Stack 11
2.2 OpenID Protocol Diagram 12
2.3 OpenID Phishing 16
3.1 Split Auth Architecture Diagram 26
Chapter 1
Introduction

1.1 Problem Statement

The Internet is now an important and established part of just about everyone’s everyday life. Whether for shopping and banking, seeking news and weather forecasts, keeping in touch with friends and family, or just socializing, most people rely on the Internet everyday for all sorts of more or less personalized services.

Personalizing user content and online experience requires the remote websites to be able to identify and authenticate the user who is requesting the content or service. This is conventionally done by having users register to the website using a unique username and associated password, so that returning users can be authenticated.

Because of the multitude of useful services that are available online, most users do not just have one or a handful of online accounts, but dozens and dozens of them. To avoid that compromise of one online account may cascade into compromise of a user’s entire online presence, it is important that passwords at different website be independent [9]. Thus, the number of unique passwords that users need to remember and manage grows with the number of their online accounts [23]. That makes remembering all of one’s credentials overwhelming, which prods the users to seek ways around conventional password system [9]. Therein lies the fatal flaw of the username and password combination system.

Probably the most widely used circumvention technique is to write down one’s passwords [14]. Whether on their computer or on piece of paper, externalizing passwords puts them in jeopardy of being lost or stolen, especially since “password-
hunters” are well aware of the most likely places where these written passwords would be kept. Another common form of circumvention is to use the same password for multiple online profiles [9] (or simple variations thereof, e.g., appending a different digit to a common ‘master’ password). Besides the adverse cascading effect in case of compromise of one account, this approach also enable a rogue website (e.g., a comics fan forum) to masquerade as the unknowing user at another website (e.g., the user’s bank). These “bad practices” are often compounded by the fact that, without mnemonic training, users pick passwords that are not just easy to remember, but also easy to guess. Pinning down precisely what makes a “secure password” is tricky, but at first approximation insecure passwords are short (usually less than eight characters), do not make use of capital letters, numbers or special characters, and are descriptive of the user (e.g., the name of their pet, or the date of birth of their daughter [14]) or of the account (e.g., “amazonpassword”) [20][39][9].

The length of a password greatly influences how secure the password might be: all other factors being equal, an increase in password length corresponds to an exponential increase in the number of distinct passwords [20]. This means a brute force attacker would have more passwords to try in the 10-character case than in the 8-character case. The same arguments can be made about passwords that do not utilize capitals or special characters. Another common password-cracking technique is for an attacker to research the background of a potential victim to create a list of likely passwords based on aspects of the potential victim’s life [14]. In fact, users surveys have repeatedly found that a small dictionary of 500,000 or 1,000,000 words captures the passwords of a large fraction of users [14], so that even without background research, an attacker who is intent in cracking a user’s password might be able to succeed by just trying the more common passwords first, employing what is known as a “dictionary attack” [14].
Despite the care that a user may take in choosing and managing their passwords, his/her web profiles might still be at risk. Indeed, the abundance of vulnerabilities in common operating systems and software packages means that the computers that people use are often infested with \textit{malware}—malicious pieces of software that users inadvertently install upon visiting rogue websites or opening the wrong attachment in a bogus email. A common typo of malware are the \textit{keyloggers}—tools that record all the keystrokes that a user makes. This produce a wealth of data that an attacker then use to extrapolate the user’s password. For example, since username typically follows a common pattern (like an email address, or an 8-character nickname that is close to the first or last name of the users), the keylogger software might be able to detect when a username has been typed. That in turns signals that the following string is very likely the user’s password.

Fortunately, there is an alternative to using the username and password combination. The alternative is public key cryptography. User authentication is a natural application for public-key cryptography, dating back to the seminal work in the late 1970’s by Diffie and Hellman [38], and by Rivest, Shamir, and Adelman [33]. In a nutshell, the idea is to employ the challenge-response paradigm together with a digital signature scheme. In a digital signature scheme, a user generates a pair of matching keys: a \textit{public} verification key, and a \textit{private} signing key. The user then registers the public key with the website under his or her username. At a later time, when the user needs to log into his/her web account, rather than submitting a username and a password, s/he just sends the username. Upon receiving a proper username, the server replies with a \textit{challenge}—a task that can only be carried out by the legit owner of the private key corresponding to the public key associated with the given username. In this context, the challenge is to produce a digital signature on a random number, freshly chosen by the web server. By the properties of secure digital
signatures, it is unfeasible for an impostor to compute the proper response, that is, a digital signature of the challenge. (The reason for picking a fresh random number each time is to prevent a middle-man from replaying an old signature at a later time.) Thus, when the web server receives a signature and verifies its correctness against the public verification key, it can rest assured that the username was supplied by the legitimate user, and the user is thus permitted to log into the website.

An important benefit of authentication based on public-key cryptography as compared to a conventional username/password authentication is that the public key can safely be reused at multiple website at essentially no loss of security. Nevertheless, arguably the main reason for the lack of widespread adoption of this paradigm is the difficulty in coping with compromise of the private key.

Private keys are much longer and more random-looking than conventional passwords, so why should one worry about their possible eventual exposure? There are several reasons, the most fundamental of which perhaps being that keys cannot exist just in the mind of their legitimate users: Even a security-conscious and mnemonically inclined user will have an excruciatingly hard time at remembering the 160 bits (or 30 characters, under BASE64 encoding) that make up even the smallest of the signing keys that is considered safe for (non-critical) security usage. Therefore, any realization of the public-key challenge-response authentication paradigm will involve an intermediate step for storing the private key securely. This in turns immediately makes its compromise possible, leading to the need to contemplate recovery scenario to cope with the eventuality that the private key is lost.

In its basic form, public-key based web authentication does not cope very well with compromise of the user’s private key: the user will need to generate a new public/private key pair, and update his public keys all websites where s/he had registered. Luckily, designs with greater sophistication have been proposed since the
1970’s, one of which is based on proactive two-party signatures [26].

Proactive two-party signatures are an enhancement to digital signatures whereby the private key is split between two parties. In addition, should either key half be compromised, the system can still be restored to a secure state by changing the two key halves in a way that does not require changing the overall public verification key. This enables a design in which the user holds one half of the key, and in case that should be lost, the system can be restored to a secure state without having to update the public verification key wherever it is used.

The work of [26] demonstrates this design for the case of accomplishing authentication in a custom network protocol for secure remote file systems. In terms of deployment, this makes for an ideal scenario, because dedicated software can be installed at all ends of communication. The problem addressed in this thesis is whether the same design can be made to work for the case of web authentication in wide-area decentralized network and within the constraints of existing Internet protocols.

1.2 Problem Scope

This thesis looks at an alternative to the conventional paradigm of usernames and passwords for user authentication that may lead to a usable and more secure web experience. A full blown, ready-to-deploy web authentication solution, however, is beyond the scope of this thesis. Broadly speaking, the goal of this investigation is the establish whether the split-key paradigm for user authentication proposed in [26] can be successfully implemented in the Internet environment. To this aim, this work pursue an implementation that can interoperate with Internet standards as a proof of feasibility of the overall approach. The resulting prototype system, Split Auth, provides thus a middle ground between the current Internet authentication
mechanism of utilizing username and password combinations and purely public-key based solution that requires public-key cryptography at every website.

The value of Split Auth in particular stems from the system’s resilience in the face of compromise of a user’s credentials. Split Auth enable users to recover from compromise easily, without having to go through extensive rekeying that would require updating the user’s public key with all its web servers.

For usability, the Split Auth system does not completely forgo using a password, but it reduces the total number of user passwords drastically. The need for still having a password in such a system arises from having to encrypt part of the private key so that the legitimate users can recover it while attackers cannot. The system takes several steps to minimize the risk of exposure of this password and of the private key half that it protects. For example, when user enter their passwords in the web browser, all the computation that require the password or the user’ private-key half take place in the user (client) machine. In this way, no sensitive credential ever leaves the client’s machine, and thus even a malicious server would not have much to steal because it will never see anything sensitive. Since overall Split Auth requires just two passwords, it is reasonable to assume that users would be able to manage them properly, and that they thus would be very secure against password-crackers.

1.3 Thesis Approach

This thesis pursues an approach to web authentication based on public-key cryptography that can be made to work without requiring extensive changes to the existing web infrastructure. This latter constraint requires adapting protocols already in use, like the OpenID standard [13][11][37][10]. OpenID is an instance of an Internet-scale single sign-on system, which allows a user to log into a server and then gain access
to other online profiles without the need of resubmitting their credentials.

This approach is desirable for two main reasons. First, many popular websites already have OpenID support; therefore, adopting Split Auth would not require substantial restructuring of the current infrastructure, and would entail better security than what can be offered by a pure username/password solution. A second, more pragmatic reason is the availability of a wide open-source code base for OpenID software, which makes it possible to customize the operational behavior of OpenID clients and servers without undue coding effort.

1.4 Outline of Thesis Work

The research work for this thesis follows naturally from the problem statement, problem scope, and thesis approach. Its main elements are summarized below.

Because by default OpenID employs conventional username / password combination for user authentication, a first part of the work will be to implement the manipulation of the cryptographic signatures at both the client and the server ends. In terms of choice of programming languages, the cryptographic tasks that need to be carried out on the client side will be coded in Javascript, so that they can be run within most browsers. Server side functionalities like signature verification will use whatever server-side scripting language is used by the OpenID server implementation that will be selected in the preliminary stage of the thesis (most likely PHP or Python).

A key feature of the Split Auth design is the mitigation of the risk of private key exposure through the splitting of the private key between two machines: the user client machine, and the Split Auth server. Functionally, this entails the need to structure the code so that the computation of a digital signature take places via
the coordination of two separate machines. A specific type of digital signature scheme will be used for this purpose [26]. The Split Auth server will compute only half of a digital signature, while the client will compute the other half. After both parties have completed this step, the client will combine the signatures into one that “proves” the user identity. This trait is behind the name of the Split Auth system: at its core, its user authentication mechanism is split between two different servers. This feature in turn enable the other key characteristic of the system, that is, the capability to refresh these keys without resulting in change to the public key.

Besides interacting with the browser, the user will need an interface to the OpenID server for the initial user registration step. A common approach in open-source implementations of OpenID clients and servers is to accomplish this by executing a shell script on the server. Therefore, a command-line PHP or Python utility will be developed that will be run on the OpenID server by the user to create the public and private key pair. Part of the private key will be encrypted to prevent an attacker with access to the key storage from stealing the user’s private key. Next, a JavaScript login web page will be put together, for users to input their usernames. This username will be transmitted to a server-side script, which will bring up the pertinent information from the user record (which in the prototype would most likely reside in a PHP configuration file). This information will then be passed back to the JavaScript page along with a challenge. This challenge will then be signed by both the Split Auth server and the client before being transferred back. Another server-side script will then verify the signature and finally allow the user in. After finishing coding the core authentication sub-system, the key-refreshing scheme will be implemented. Besides presenting the user with the familiar prompt for old and new passwords, this scheme will require authentication in the form of a recovery key, which was originally created upon initial user registration. This key-refreshing functionality will be used
in the event that one half of the user private key becomes compromised: their private key will be manipulated via a refresh operation, thus avoiding the need to reissue a new public key.
Chapter 2
Background

2.1 OpenID

To gain an understanding of how OpenID works, it is first beneficial to analyze the protocol stack that it uses. URI and XRI are at the bottom of the stack, as pictured in figure 2.1, because the rest of the protocol hinges on one of these identifiers. The URL comes from a user’s website like a social networking profile or a blog. However, sometimes these URLs can change because the user no longer wants to use them. If this happens the user will lose all of their online profiles because their online identity is tied to this URL. XRI (Extensible Resource Identifiers) are much more robust to changes than URLs, and can be used to alleviate the problem of obsolete user profiles. This is because it consists of two things. The first is called an iname. This is comparable to an URL and quite frequently is some URL belonging to a user. These are the human readable pieces of the XRI usually corresponding to a domain [36]. The inumber is a unique number that is only assigned to one user within the given realm [36]. Using XRI is much more robust than the URL method because if the iname changes, the inumber does not. This allows the user to not lose their online profile information. [30] [31]. Next on the stack in figure 2.1 is the Yadis Protocol. This is what retrieves the identifiers for the user once a user submits their username to a RP. The RP will then employ the Yadis protocol to retrieve an XRDS document from the claimed location. This is stored in the URL or the XRI iname field. If there is no document at the supplied URL or iname, but there is another resource, the protocol will go to that address for the document. This will continue until the document is
Once the XRDS document is recovered, the RP will know who the IP is. The user is then redirected to the IP to login with their credentials [30] [2]. The next portion of the stack in figure 2.1 deals with the OpenID Authentication. This is the part of the stack that carries the proof that the user has logged into the IP successfully. Once the RP identifies the correct IP, a shared secret is generated between the two [30]. This shared secret has an expiration time. This expiration time is usually within 30 minutes [13] This expiration date stops anyone from using the credentials without the original user knowing. If the cryptographic proof that the user has logged in is either expired or missing, the user will be redirected to their IP to login just as if no shared secret had been submitted [37]. It also prevents replay attacks. A replay attack is where an attacker saves information from a previous transfer of information to play later to act like the user. The last part of the protocol stack to be discussed is the OpenID Transport Protocol. That is because this layer and all those above it are abstract layers. Having the upper layers of the protocol stack being abstract
allows the protocol to be extended in ways that were not originally thought of. This means that the protocol will survive changes in common practices with ease. The only thing that is certain with the upper layers is that this is where the information is exchanged in a secure manner [30].

The next step in understanding the OpenID protocol is to understand the steps that are taken for a user to be successfully logged in. First the user navigates to the RP of their choice and inputs their username, step (1) in figure 2.2. The username in OpenID is the URL, or the iname, that the user employs. The RP will then employ the Yadis protocol to get the XRDS document from the specified location, steps (2) in figure 2.2. Once the XRDS document is retrieved, the RP will parse the XML file to find out who the IP is [31]. Once the IP is identified. The RP and IP utilize the Diffe-Hellman Key Exchange to produce a shared secret [37] [36] [2] [21], step (3) in figure 2.2. The IP will then transfer the key to read the cryptographic proof that the user has logged in successfully [37]. Once that step is completed, the user is redirected to the IP to login, step (4) in figure 2.2. Once, the user is logged in,
the IP gives cryptographic proof that it was successful and redirects the user back to
the RP, steps (5) and (6) in figure 2.2. The user’s browser then hands the RP the
cryptographic proof that the user has logged in successfully. The RP can verify this
with the key that it obtained from the IP before, step (7) in figure 2.2. The protocol
changes slightly if the user is already logged in. Since OpenID can be used as a single
sign on protocol, if the user has already logged in, there is no reason for the user to
be redirected to the IP. In this case, steps (4) through (7) would be omitted.

2.1.1 OpenID Security

Despite the usefulness of the OpenID mechanism, there are still some security con-
cerns. First, logging into the IP is typically accomplished using username and a
password combination [13]. As discussed in the Introduction, attackers have become
adept to breaking user passwords, which are often too short, are simply easily guess-
able. Another security concern in OpenID is the lack of coordination between the
IP and the RPs regarding credential expiration time. It takes 30 minutes for the IP
credentials to expire, whereas, the authors of [13] found that usually the credentials
on the RP do not expire in the same amount of time. Sometimes, valid credentials
stay around for days at end. This allows for a very basic attack. The cryptographic
authentication token is stored on a per-browser-window basis, meaning that multiple
windows of the same browser will be authenticated in different ways. It is possible
to have an unauthenticated browser window access information that should only be
accessed by authenticated windows by copying and pasting the URL from the au-
thenticated window to the unauthenticated one. This can be done for days after
the credentials expire because they will remain active on the RP for days after the
expiration occurs [13].

Another flaw is that if the RP and IP are both contacted over HTTP first, they
are simply redirected to the HTTPS, like in step (1), then the contacting information will be sent in plaintext during the initial connection, even through the connection is later redirected to HTTPS. While this is not an attack per se, the information can be stored and analyzed.

In secure network communication tools hash, all fields of a message are protected with a message authentication code (MAC). This is true in OpenID, only to a certain extent: only some of the fields are protected with a MAC, which allows for parameter injection. An attacker can also intercept messages and tell the RP which fields are to be protected with a MAC. If the attacker tells the RP that none of them need to be protected, then the attacker can modify all fields as they please.

Unfortunately, OpenID does not mitigate phishing attacks [21]. A phishing attack is when a user receives some sort of communication like an email or a phone call asking for their login credentials to a website but then is instructed to visit a rogue website which looks like the legitimate website as seen in figure 2.3. The vulnerability can be exploited when the user is redirected to the IP. If the RP is malicious, the RP could simply send the user to a page that looks like the IP’s page. As a result the user will enter their credentials into a system controlled by the attacker. Luckily, ways to combat OpenID phishing have been developed. For example, the method of [21] has the user submit a picture when registering. This picture will then be displayed every time a user logs in. The IP will know what picture to display through a cookie stored on the user’s computer, whereas it would be difficult for the rogue website to behave accordingly allowing the user to realize they have entered a malicious IP. The obvious downfall of this procedure is that the user will not see the picture displayed if the IP is accessed from different computers. Another way to combat phishing is by an Extended Validation Certificate also described in [21]. These certificates are provided by Verisign.
a website that has an Extended Validation Certificate then the background of the website will become green. The obvious drawback of this method is that the user needs to be using Internet Explorer 7. A not so obvious shortcoming of the method is that Verisign charges $1000 for the certificate. Another method in [21] is called SeatBelt. This is a joint venture between Verisign and Firefox. If the URL is different from the URL that is already set on the user’s computer, the user is given a warning message that the website is different. This feature is only available after a user registers and only on the computer that the user used when registering. Vidoop’s Password solution described in [21] is another way to combat phishing. This message uses a picture as the password. However, the activation code is sent to a user’s email address, so if the email address is compromised, then so is the picture password. Jabber’s Authentication by Messenger, described in [21] attempts to defeat phishing by sending authentication codes to a user through instant messenger or SMS text messages. The downside is that the user must have a secure way to access instant messages or text messages all the time. There is also a new method described in [21] to combat phishing. A user picks a message or picture to be displayed while logging in. The message or picture is encrypted with the company number of the IP, which all IPs have in reference to OpenID, and a randomly generated key. The user then stores this on a USB drive or a computer. Therefore, the encrypted file becomes a token. When logging in, a user will submit the file first. Only a legitimate company will be able to decrypt the file. However, the drawback of this method is that the user will need to have the file with them whenever logging into an IP [21]. Another way to log into OpenID that could be used to combat phishing and attackers breaking passwords is by using a mobile device. These devices would be used instead of the a username and password [19]. When the user goes to log into the IP, the user would select “SIM USB Dongle/Bluetooth”. The mobile device would be connected and
then the authentication would take place through the mobile device. Basically, the mobile device would be used a token for users to log in with [19].

2.2 Identity Management

Everyone has an identity online through their pseudonyms, commonly referred to usernames [8]. Identity management helps to identify users and give permissions. Every system has two parts, issuing credentials and restricting access to places. Phishing is the act of stealing some type of information from a user usually login information or financial information. Phishing can succeed because authentication is one way. The users never see the authentication of the service provider. Therefore, the user is left to guess whether or not the place that the credentials are to be submitted is a legitimate place or not. There are many different types of identity management systems [4], which are briefly discussed below.
2.2.1 Isolated User Identity Model

In the Isolated user Identity Model, the service provider issues credentials to the user. The service provider is in charge of what makes up the credentials. Each user gets different credentials for each service provider. This is the most popular model in use. However, the fatal flaw of this system is that there is too much information for the user to be able to remember all of it [4].

2.2.2 Federated User Model

In the Federated User Model, there is more communication between service providers in the realm of credentials. In this model, the service providers will create contracts that allow users to use the login credentials from other service providers. This means that the users only need one set of credentials for the federation. This allows establishing a single sign-on within the federation [4].

2.2.3 Centralized Models

There are two different sub-models that fall under the broad umbrella of the centralized model. The common user model and the meta user identification model. In the common user model, a user accesses all the service providers with one set of credentials. Normally, this entails the use of a public key infrastructure with the combination of a Certificate Authority. This model generally works best within an organization. The meta user identification model instead uses the metadata as the identifier of a user. This model is also best used within an organization because the metadata should not be publicly available to everyone in the world [4].
2.2.4 User Centric Identity Management

In user-centric identity management, users are in charge of their own credentials. Usually, the credentials are stored on a hardened removable disk or a hardened section of the hard drive. The user must authenticate himself to this hardened section before the credentials can be used [4]. Although, OpenID is not stored on a removable disk, it is still considered an example of User centric Identity Management. In accordance with User Centric Identity Management, not only does the user have to store their own OpenID credentials but they must also authenticate themselves to the OpenID IP, before gaining access to any of their websites [36] [30]. Since OpenID is considered as a User Centric Identity Management, Split Auth is also considered to be User Centric Identity Management. The same reasoning applied to OpenID can also be applied to Split Auth; the user is responsible for storing their own credentials and the user must first authenticate to Split Auth before any of their websites. The main difference is that users must authenticate to two different servers before gaining access to their desired website.

2.2.5 Shibboleth - A Federated Model Example

The Shibboleth system is an implementation of the Federated Model of identity management. This system allows users of the federation to access networked resources. The system has two components that are independent from each other but are essential to how it works. There is a Shibboleth Identity Provider (IDP) and a Shibboleth Service Provider (SSP). Users gain entry into the system by first trying to access a web resource where he must log in. The SSP then redirects the user to what is known as a navigation page. This navigation page has a list of participating organizations within the federation along with each organization’s permissions. The user must se-
lect an which organization the user belongs to, that is, one whose IDP will recognize
the user’s credentials. The user is then transferred to his home IDP, where the user
logs in normally. The IDP then redirects the user back to the SSP with some sort of
assertion that the user has signed on. The SSP validates the assertion data and asks
the IDP for more information on the user. After all the information is gathered, the
user is either allowed or denied access to the web application based on the permissions
of the user. The reason the SSP seeks additional information about the user is that
the SSP does not have access to the user’s record. This implementation also protects
the user’s privacy by allowing the user to control what information is sent to the other
organization.

There are a few reasons as to why Shibboleth does not scale well to the internet
environment. One is that the IDP is just a plugin to the Microsoft LDAP authenti-
cation system for companies. Most user’s will not have this on their home networks
so the IDP would not be usable in that context. Another reason that it is not used
in the Internet environment is that there are too many authentication items that the
parties have to agree on like the definition of the attributes, how to locate the servers
of other participants, and the procedures for sensitive information. In the federation
environment, the organizations have an IT department to make these decisions, so
that this process remains transparent to the users. Regular users would have a hard
time making these decisions [32]. Shibboleth also protects privacy by including Se-
curity Assertion Markup Language (SAML) user as the encryption method [32]. It
allows a user to use another organization’s resources while being anonymous to the
other organization. The IDP may assign an unique identifier to a username to keep
the user anonymous [34].
2.3 Kerberos - A Centralized Model Example

Kerberos is an implementation of the Centralized Model, more specifically it is an example of the *common user model*. Kerberos keeps passwords off the network so that they cannot be sniffed by an eavesdropper. It is also not an authorization service in its pure form but it can be linked to other systems to provide authorization [25]. It works by using a series of encrypted messages to prove a user is authenticated. The Kerberos protocol is rather complex; here we provide only a brief outline of its working; see [25] or [35] for more details. It is based on the Needham and Schroeder authentication protocol [35] with a few adaptations. One such way extension is the presence of a ticket-granting service, which allows single sign-on within the network. To authenticate, the user logs in using a password to their machine. The password is used as a key to encrypt a message from the user's machine to the authentication server. The message specifies what application the user would like to access. The authentication server shares a key with each server; this enables the authentication server to communicate securely to the specific server that the user is logged in. The protocol can be insecure if not all the tickets are stored because of the long ticket life. Timestamping does not stop replay attacks because network time is notoriously unreliable to the point where an attacker could convince a server the time has actually changed [35]. The protocol also relies on the fact that the servers are not compromised [35]. This implementation does not scale to the Internet environment because of the central authentication server. If this were to be implemented in the Internet environment, the central server would be a single point of failure as well as a bottleneck.
Chapter 3
Design Overview

The Split Auth authentication system is a unique Internet protocol that allows users to log into websites more secure as compared to the systems to which way most users are used to. The Split Auth system is built on top of the OpenID protocol. Instead of using the normal multiple username and password combinations to log users into websites, it uses public key cryptography to create a digital signature that no other user can forge.

3.1 Design Guidelines

There are many stakeholders who are involved in the Split Auth system. Of course, the main stakeholder is the user. This is because all of their credentials are tied to this system and they will use the system everyday. In the regular OpenID architecture, the users have the option to set up their own IdP server, or they can choose to use another’s website to act as their IdP e.g. Google. To take full advantage of the security of the Split Auth architecture, the user must set up their own IdP by installing the Split Auth IdP. This is the only option because none of the other relying parties employ the public key signatures to log users in. The prototype IdP that we have developed in the context of this thesis, includes a configuration server. The configuration server holds the user preferences as well as the cryptographic keys that are being used. Conceptually the same configuration server could be distinct from IdP but in this specific Split Auth architecture, the IdP and the Configuration Server are the same machine. This was done because of time constraints. Therefore, IdP will refer to both the IdP and the Configuration Server. In this case, the user will
also be responsible for the Split Auth server because this protocol is not implemented at the other IdPs. The websites are also a stakeholder in this architecture. This is because they must employ the RP part of the system. The RP of the architecture will not change from the current architecture of OpenID. If the websites do not do this, the protocol will not work for the users.

3.1.1 Keep Password Off Server

In order to keep the password off the server and not require the user to download software Javascript was used. Javascript is run locally in the user’s browser with no interaction with the server by default. Of course, there is communication with the server but the programmer can dictate what information the server is given which is the opposite of using other scripting languages. This may seem like overkill to keep the password from the server but consider the following scenario. A user tries to log into a webserver but forgets which password he has used so he tries multiple passwords to log in with. If the server is malicious, the server could save all username and password combinations and try them at a multitude of other websites trying to use the combinations to log in as the user. The other scenario that can be used is the one where the server uses the same username and password combination it has rightfully stored to allow the user to log into its own website to log into other websites. This scenario is very plausible because as already stated, users do not have different username and password combinations for other websites. If they do have different combinations, humans are not adept at remembering which combinations are for which website. In either case the server will try to log into other websites based on the user credentials that are submitted.

By keeping the password from the server, it prevents the server from logging into other websites with the user’s credentials. Admittedly, not every single user may
be running javascript but it is so widely used that the user would just have to turn on Javascript in their web browser to use the software distribution. When the user visits the IdP, the browser gets the Javascript from the IdP. At this point, the user can do one of two things. The user can proceed as normally or the user can print out the Javascript and inspect it to make sure that the software will do what is expected. Allowing the user to inspect the code will give them more confidence in using the system.

3.1.2 Limit the Number of Passwords

In the current paradigm, users are supposed to remember \( n \) username/password combinations, one for each website that is logged into. To make \( n \) passwords memorable, humans make them as short and as simple as possible. This means that the passwords are very guessable and not very secure. Therefore, this system reduces the number of passwords from \( n \) to 2. By reducing the number of passwords a user must remember from \( n \) to 2, the user can make the two passwords much longer. By making the passwords longer, the passwords become harder to break [20]. Therefore, the passwords required by Split Auth are supposed to be 15 characters long. By making the passwords need to be 15 characters, the user can use a sentence as their password such as *My favorite Major League Baseball team is the New York Mets.* which happens to be 60 characters. However, it is much easier than remembering a password like *Tr@!n!n9* which stands for *training*. The first password is much easier to remember can be compatible with the Split Auth system because the user only has to remember two of them.
3.2 Threat Model

There are many threats associated with authentication schemes. This is because authentication schemes hold the keys to personal information, email accounts, and even bank accounts. This means there is monetary value from breaking those schemes and gaining access to the accounts. This authentication scheme was designed to withstand many of these threats. Communication over the Internet can be monitored by a third party. Since this is a conceivable threat, all of the communication done between servers is done through HTTPS connections. By using HTTPS, an encrypted connection, the information gotten from monitoring is not readable. HTTPS does not necessarily prevent these replay attacks, but could make it harder. In storing the challenge on the server and using that challenge to validate the digital signature, replay attacks will not be possible. Also, encrypting the communication will not allow the user to know which part of the communication to playback to the server.

Attackers would love to get their hands on the passwords and secret keys that are involved with this system. To prevent this, the full secret key and passwords are not stored in plaintext. The private key is split into two separate keys. One of these pieces is encrypted with AES[5][6] so that the full private key is not known to the attackers. This encryption effectively stops the attacker from getting the full key. The recovery key is not stored in plaintext so that the attacker would have the chance to change the password on the user. Instead it is stored as a hash just like the password for the Split Auth server. In the hashes, salts are used so that nothing can be precomputed.

There is always the threat of an account being compromised. This is mitigated by allowing the user to change their keys and password whenever they desire. This has the added issue of attackers pretending to be the user and changing the key on
the user. This is mitigated by the recovery key being a secondary authentication mechanism.

There are threats that are not mitigated by Split Auth. One of these threats is the fact that the IdP can impersonate the user whenever it wants. This is because the IdP can tell any RP that the user is logged in and use that session to do what the IdP wants to do. This is not mitigated by Split Auth and is a problem with OpenID. Though this is a problem, the fact that Split Auth is a bridge to the overall goal of having public keys at each website, this flaw is not a major issue. This is because in the scenario of using public keys to log into websites, the IdP will be obsolete. Thus, this risk will be mitigated. One of the assumptions that is made with the Split Auth software is that all of the servers are available when needed. With that in mind the software does not protect against a Denial of Service, DOS, attack. If a DOS attack is successful against the IdP, the user will not be able to log into any of their websites. Therefore, the user should employ some sort of DOS prevention technique. This particular IdP utilizes a cookie to tell the web browser that it is logged in. If an attacker can gain access to the cookie, the attacker will be able to masquerade as the user. Lastly, the system does not protect against password cracking. The best the system does is to not give the password to the IdP and it doesn’t transfer it over the network. This password is also not stored anywhere in the IdP. It does partially mitigate this attack by making the length of the password be at least 15 characters long however it does not prevent a brute force attack to be performed.

3.3 Stakeholders in OpenID

The main stakeholder in OpenID are the users. This is because this system is how the user access all of their websites. If their credentials were to not be a secret, then
anyone could impersonate the user and gain access to their accounts. In OpenID, the user has the option to run their own IdP or give their credentials to an already existing IdP like Google. A user can make this decisions so that they feel very comfortable with who is guarding their credentials. Another stakeholder are the websites that employ OpenID as an authentication measure. This is because this is how the users who regularly visit their website authenticate themselves to access their accounts. If there is something wrong with the system, the users will stop visiting. Lastly, the IdPs that users can choose are a stakeholder because they are harboring a user’s login credentials for a wide array of websites. The IdP’s would like to be trusted by the user’s who use their services.
3.4 Architecture of the Split Auth System

3.4.1 Identity Provider

The Identity Provider is the biggest component of the Split Auth system. It includes login.html, MyID.config.php, MyID.php, Password_Reset, Password_Delta, and the register.php. A summary of each component of the IdP is next.

Client Interaction

Login.html is the only part of the entire Split Auth system that the user interacts with. The file is composed of HTML and Javascript. The HTML part of the file is for allowing the user to enter their information into the system. The Javascript part of the file is for manipulating the login credentials of the user to create their digital signature. The HTML creates two forms, the HTML mechanism for creating places for the user to enter information. One of these forms has three text fields for logging into the Split Auth system. They are the username for the Split Auth system, the password for the Split Auth system and the password for the Split Auth Server.

To accomplish the goal of not sending the password to the server, to mitigate the instance where the server is taken over by an attacker. If that is the case then the server is malicious and saves all the passwords of the user, instead Javascript is used to send only the username to the IdP. Javascript runs in the user’s web browser meaning the server the code came from will not have access to the password.

Once the IdP has the username it parses the configuration file for the username field. If the usernames match, login.html will receive half of the signing key, the public key, the salt for the password and the challenge. Login.html then computes the rest of the signature and sends it to the MyID.config.php’s[7] validation mode.

The other four fields in login.html are for changing the private key that is
used to create the digital signature. The user would want to change their private
key if they suspect that someone has broken into their account or if they accessed
their information from a public place and want to be overly cautious. The first text
field that is encountered is the recovery key field. The recovery key is a secondary
authentication mechanism given to the user when they register or change keys. This
is needed because if the account has been compromised it is logical to assume the
password has been compromised as well. This would allow an attacker to change keys
and passwords without the user’s knowledge. Therefore a new way of authentication
is needed thus, the recovery key. The old password is still needed by the system
to further establish the user’s identity. It would be very unlikely that another user
would have both the recovery key and the old password. Lastly, the user is required
to input their new password twice. The user must input it twice so that the likelihood
of a spelling mistake is minimal. After all the required information is entered, the
recovery key is sent to Password_Reset.php to check if the recovery keys are the same.
If they are the same then the private keys are updated.

IdP Configuration File

This is the configuration file for the IdP. It stores the cryptographic parameters that
are to be used in the creation of the user’s digital signature. It also stores the user’s
preferences for the IdP. As stated before, this can be a separate entity from the Idp.
However, because of time constraints, the IdP and the configuration file are the same
entity.

IdP

MyID.php[7] is the IdP that the user must use. The IdP was created for the OpenID
protocol but modified for the Split Auth system. Therefore, only those functions that
have been modified will it be discussed here. There is an appendix that details all of the functions associated with the IdP. The Authorize Mode function receives the username of the party trying to login. To verify the username is correct, it parses the configuration file for the IdP. If the username is correct, the rest of the file for the information to complete the signature. It will then compute a random challenge which is sent to the user’s web browser. The challenge is a very long number that is very hard to guess. The challenge is what is digitally signed with the user’s digital signature. The Validation Mode verifies that the user submitted digital signature is correct.

**Key Reset**

The Password Reset.php script is one of two scripts that will reset the password and update the keys for the user. First is parses the file to make sure the recovery key that is entered is the correct one. If it is the correct one, the encrypted private keys are parsed from the configuration file and sent to the user’s web browser. This is the other script that updates the keys for the user. This file updates the public key used to create the digital signature for the user. It also creates a new recovery key for the user and gives it to the user. The new information that the digital signature will be based on is written to the configuration file.

**Registering with the IdP**

The Register.php script is used by the user when setting up their IdP. This script creates the username, accepts the password, and creates the keys that will be used by the Split Auth system. It will then automatically update the configuration file.
3.4.2 Split Auth Server

This is the part of the system that creates half of the digital signature. The login.html creates half of the digital signature and the Split Auth server creates the other half. Before computing the signature, it checks to make sure the user is authorized by checking the Split Auth server configuration file which is just a text file with the login information. If they match, the half signature is computed. If they do not match, no signature is computed because the half signature should not be given to anyone who wants it. It will still not allow an attacker to guess the other half of the signature but it is not good practice to do so.

Registering with the Split Auth Server

This is the register script for the Split Auth server. Much like the IdP register script, it allows the user to pick their username and password. However, the username must be the same as the IdP username but does not check to see if they are the same. It stores the information in the Split Auth server configuration file.

3.5 Operational Overview

The first step in using the system is to register with the IdP and the Split Auth Server. To login to a RP, the user will submit their OpenID URL. The OpenID URL is where the IdP is located. The RP will first check if the user is logged into the IdP. If the user is logged in, the user will gain access to the website. If the user is not logged in, the user is redirected to login.html. The user would then submit their login credentials. Login.html will then only send the username to the IdP, Authorize_Mode, to make sure that it is valid. If the username is valid, the information used to create the signature is sent back to login.html. The encrypted part of the digital signing
key is decrypted and half of the signature is created. Then login.html sends the username, the Split Auth password, and cryptographic information to the Split Auth Server to create the other half of the signature. Login.html then combines the two halves and sends them to the Validation Mode of MyID.config.php[7]. If the signature is correct the user is logged in. If not, a message is displayed to the user.
Chapter 4  
Design Details

This section details the design decisions that were used in creating Split Auth. Some knowledge of discrete mathematics might be necessary to fully understand the cryptographic mathematics.

4.1 Cryptographic Details

4.1.1 Schnorr Signature

The Schnorr Signature Scheme relies on a cryptographic hash function and the discrete log problem [26]. The cryptographic hash is used to create the signature and the discrete log problem is used to prove that the scheme is unforgeable. This can be done over group G.

Key Generation

The first thing that must happen is the key generation. In this generation, two very large primes, q and p, are picked such that $q | (p - 1)$ and an element $g$ is picked such that $g$ is an element of $\mathbb{Z}_p^*$ and has the order $q$. The elements $p$, $q$, and $g$ are shared between all of the parties and can be even known by an eavesdropper or attacker. This is because the elements are so large, that an exhaustive search is not possible. The next step is to generate the public key. This is done by picking a random $x$ that comes from $\mathbb{Z}_q^*$. The random $x$ is used to calculate $y = g^x \mod p$. This makes the public key "p,q,g,y". The signing key is $x$. As stated before, $p,q,g$ can be shared with everyone. If this is done that the public key is just $y$. 


Signing a Message

The keys are to be used to sign messages so that the recipient knows who it has come from. To sign a message \( m \) a random \( k \) is picked from \( \mathbb{Z}_q^* \). The value of \( k \) must be kept secret. Next \( r \) is computed to be \( g^k \mod p \). Then \( e \) is computed to be \( e = \text{SHA-1}(m, r) \) and \( s = k + xe \mod q \). This makes the signature ”\( r, s \)”. 

4.1.2 2Schnorr

A Proactive Two-Party Signature Scheme (P2SS) is where two parties produce digital signatures in which their keys must be updated after a period of time in a way that does not allow old shares to impersonate the signer[26]. Fortunately, the Schnorr Signature Scheme can be modified to be a P2SS called 2Schnorr. The process described above is the same with only a minor difference. The difference is that the signing key \( x \) is split into two pieces, \( x_L \) and \( x_R \) and are both elements of the \( \mathbb{Z}_q \). This means that the full signing key is \( x \equiv x_L + x_R \mod q \).

4.1.3 Signing A Message

Signing a message is virtually the same thing in the 2Schnorr paradigm as it is in the normal Schnorr paradigm. Each party picks a random \( k \) in \( \mathbb{Z}_q \) labeled appropriately as \( k_L \) and \( k_R \). This allows both parties to compute \( r_L = g^{k_L} \mod p \) and \( r_R = g^{k_R} \mod p \). One party will compute \( s_R = k_R + x_R e \mod q \). The other party will then compute their side by doing \( s_L = k_L + x_L e \mod q \) and \( s_R = k_R + x_R e \mod q \). Then one side will assemble the message. To do this, \( r = r_L + r_R \), \( e = \text{Hash}(m, r) \), \( s = s_L + s_R \mod q \).

In Split Auth architecture, the client’s machine will compute the left side of the signature and the Split Auth server will compute the right side of the signature. The client’s machine will be the machine that combines the two halves of the signature.
To verify a signature, the same process is done in the single party version of the Schnorr Signature Scheme.

### 4.2 Client Interaction

#### 4.2.1 Logging Into Split Auth

Login.html is the main script of the Split Auth software. This is the script that the user has interaction with and it is run locally on the user’s computer instead of on a server. After the user submits their username and passwords, the Schnorr Signature begins to compute. To begin this, the encrypted part of signing key is then decrypted using the AES.js[6] file, the salt, and the user’s password. This part of the key is encrypted to prevent anyone from having the entire signing key if they are able to gain access to the configuration file. It is encrypted by using counterAES.php[5] using a SHA-1 message authentication code hash of the password using the salt as the key. After the decryption, a 512 bit random number, $r_L$ is created to create $a_L = g^{r_L} \mod p$. Then the username, Split Auth password, and $a_L$ to the Split Auth server. The response from the Split Auth server is $a_R$ and $z_R$. Login.html will then compute $a = a_L \cdot a_R \mod p$ and $c$ by hashing $p$, $q$, $g$, $y$, the challenge and $a$. After computing $c$ login.html can compute $z_L = c \cdot x_L + r_L \mod q$. After computing $Z_L$, $z = z_L + z_R \mod q$ can be computed. Now $a, z, y$, and the challenge can be sent to the Validate_Mode of MyID.config.php[7].

#### 4.2.2 Resetting the Keys

Login.html first extracts all of the data from the text fields. The recovery key is then sent to Password_Reset.php and gets back $x_L$, the salt used with the old password and $salt1$ which is the salt for the new password. The encrypted part of the signing key
is decrypted and $x_L$ is modified by doing $x_L = (x_L - \text{delta}) \ mod \ q$. It then encrypts the private key by using salt1 and the new password. Then $x_L$, delta, and salt1 are sent to Password_Delta.php.

### 4.3 IdP Configuration File

The configuration file stores information about the IdP and the cryptographic parameters that are used to create the digital signatures. The user preferences that are stored are the username and other details about the user. The preference that needs to be changed is the option that allows the PHP GMP library to be used. The cryptographic details that are stored are $x_R$, $y$, salt, and the recovery key. The actual value that is stored for the recovery key is the hash of the recovery key. This is so that an attacker cannot change the password and keys at their discretion and lock the user out of their accounts. The other cryptographic piece that is stored is $x_L$. This is the encrypted portion of the signing key. It is encrypted so that an attacker cannot produce forged signatures allowing an attacker to gain access to the user’s accounts.

### 4.4 IdP

#### 4.4.1 Authorize Mode

This is the mode of the IdP that checks to see if the username submitted is valid. If it is valid, the configuration file is parsed for $x_L$ and salt. It then creates a challenge for the user’s web browser to digitally sign. The challenge is stored in a PHP session variable so that the server has a stored version of the challenge instead of relying on the one submitted by the user. If the IdP relied on the challenge submitted from the user, there would be a chance of a replay attack. Implementing the IdP this way mitigates that risk. All three elements are sent back to login.html.
4.4.2 Validation Mode

This is the mode of the IdP that verifies the digital signature that the user submits. To do this, it first computes the hash of \( p, q, g, y, \text{the stored challenge}, \) and \( a \). The hash is stored in \( c \). The signature is then validated by computing \( z < q, a^g \mod p \equiv 1, \)
\[
g^z \mod p = (a \cdot (y^c) \mod p),
\]
and \( \text{the stored challenge} = \text{user submitted challenge} \). If any of these calculations are false, the validation is exited and the user is not logged in. If the signature passes the validation, then the user is logged in.

4.5 Key Reset

Password_Reset.php is the first file that is accessed when resetting the keys. This parses the file for \( x_L, \text{salt}, \) and \( \text{recovery key} \) if the \( \text{recovery key} \) hash is a match to the one stored. The \( \text{recovery key} \) is not stored as plaintext because an attacker would be able to change the user’s credentials if it was stolen. The script then sends back \( x_L, \text{salt}, \) and a newly created salt for the new password. Password_Delta is the second file accessed when resetting the keys. It receives the new \( x_L, \text{the new salt} \) and \( \text{delta} \). The IdP’s configuration file is parsed for \( x_R \) and it is updated. It is updated by calculating \( x_R + \text{delta} \mod q \). A new recovery key is created and hashed by using the new salt. The elements \( x_L, \text{the new salt}, x_R, \) and the hashed \( \text{recovery key} \) are written to the IdP configuration file. The plaintext \( \text{recovery key} \) is given to the user to store offline.

4.6 Register Script

Register.php is the registering script for the IdP. It allows the user to input their desired username and password. When the user enters their desired password, they must enter it twice and it must be at least 15 characters. The user must enter it twice because it is not written to the screen so there is a chance for spelling mistakes.
Entering the password multiple times cuts down the probability of the mistakes happening. The script will compute $x_R$, $y$, and an encrypted version of $x_L$. It encrypts the $x_L$ using the counterAES.php[5] file. It then automatically updates the configuration file for the IdP. The recovery key is then given to the user to store somewhere offline.

### 4.7 Split Auth Server

It first makes sure that the usernames and passwords match. Passwords are stored as a hash of the password concatenated with the salt. This is done so that an attacker cannot gain access to the Split Auth server whenever the attacker desires. If the username and passwords match, the IdP configuration file is parsed for $x_R$ and $y$. It then computes $a_R = g^{r_R} mod q$ where $r_R$ is a random number. Since $a_L$ is sent to the server by login.html, the server can compute $a = a_R + a_L mod p$. The hash of $p, q, g, y, a, and the stored challenge$ in the $c$ variable. This is used to compute $(c \cdot x_R) + r_R mod q$. Then $z_R$ and $a_R$ are returned to login.html.

### 4.8 Registering with the Split Auth Server

Register_SA.php is the script that allows a user to choose their username and password for the Split Auth server. The username must be the same as the username for the IdP the password, however, should be different. Again, the username is not checked to make sure they are the same. The desired password is entered twice and must be 15 or more characters. The desired password is entered twice to minimize spelling mistakes since the password is never written to the screen. This script also creates a salt to store the password with. The password is hashed as a concatenation of the salt and password to prevent attackers from guessing the password and reading it in
plaintext.

4.8.1 Support Files

AES.js[6] allows for items to be encrypted using the Advanced Encryption Standard, AES, in counter mode. Part of the private key is encrypted so that an attacker cannot steal the whole private key easily. Another file is aesCounter.php[5]. It is the same as the AES.js[6] file however, it is implemented in PHP so that the register script can encrypt part of the private key. Cryptographic digital signatures require very large numbers that cannot be easily implemented in the normal Javascript language. Therefore, BigInt.js[22] is used so that login.html can compute cryptographic digital signatures. It must be used because Javascript does not have inherent support for Big Numbers.

4.8.2 Javascript Random Number Generator

The random number generator was redone to be a better source of psuedorandomness. The current Javascript implementation was not strong enough for cryptography so, it had to be changed. The random number generator is now seeded by using timing from the user typing characters into a webpage.
Chapter 5
Implementation Details

This chapter outlines the decisions that were made in regards to specific libraries and software.

5.1 PHP

The language of PHP was used because it was easily compatible with the Apache [3] webserver. This made setting up the original IdP very easy. PHP is also very easy to learn how to code and the language is well documented on the Internet. Therefore it made it easy to navigate problems that would inevitably be encountered during the implementation phase. PHP also, has an implementation of GMP for the computing the cryptographic portions of the system. Originally, the goal was to use Python to implement the IdP because of the cryptography library Charm being developed by Johns Hopkins University. However, making Python work with Apache [3] proved to be difficult as well as finding a simple IdP that can be modified.

5.2 GMP

The decision to use GMP for PHP was rather simple. There are two reasons for this. The first is that the PHP GMP library is widely used and well documented. This means that it was guaranteed that the functions operated correctly so there would be no mistakes in the library. The fact that it is well documented means that learning the library was a trivial task. The other reason the library was chosen, was that the IdP had PHP GMP support already built into it. The only thing that needed to be changed was the part of the configuration file that tells the IdP to use GMP instead
of its defined Big Number implementations. Since GMP PHP is a better library it was used instead of the ones that come with the IdP.

5.3 PHPMyID

The decision to use PHPMyID[7] came through trial and error. In the beginning a few different IdPs were tried like poit [40], DjangoID[1], w2p OpenID[12] and samadhi[15]. PHPMyID[7] is because the IdP of choice because it was the easiest to setup. In fact many of the IdPs that were tried had very little setup instructions. PHPMyID[7] was no different but there were other references on the Internet that helped in this process. It was one of the few IdPs that came ready to be used with only minimal setup on the part of the user. This made it attractive because user involvement should be kept to a minimum in the setup process. PHPMyID[7] also has very easy code to follow. Having easy code to follow made it easy to visualize where the changes should be made and implementing those changes. Lastly, PHPMyID[7] had inherent support for PHP GMP making it easy to add the GMP functionality to the IDP.

5.4 AJAX

AJAX[18] is the accepted way of having Internet scripts communicate with one another. This made it easy for the client Javascript calculations and the server calculations to be known by both parties. Also, AJAX[18] allows for HTTPS connections to be honored throughout the execution of the Javascript which is the preferred communication method. However, the IdP used in this specific architecture does not have great support for HTTPS communication so all communication is done in HTTP. If HTTPS could be used, it would mean that since the initial connection is in HTTPS, the only connection allowed, the subsequent communications between
the servers and user all used HTTPS connections. This comes from the same origin policy of Javascript but is accurately reflected in the AJAX communication. This allows the information being exchanged to be encrypted and unmonitored by a third party. This allows the user’s information to remain confidential.

5.5 Apache

Apache [3] was used because it is the default webserver of the Linux environment, where the development took place. Apache [3] also has built in support for PHP. This means that the webserver will run the PHP script instead of serving it to the client. This was made easy through the mod_php module and enabling the module. This made the setup of the IdP very easy.

5.6 Javascript

The reason for using the Javascript language is for the simple fact to keep the password off of the server’s machine. This keeps the server from ever knowing the user’s password since it is not stored anywhere in the configuration file that is saved on the IdP. If the server is compromised, the attacker will not be able to impersonate the user.

5.6.1 Javascript Cryptographic Libraries

Big Number Implementation

The reason for using BigInt.js[22], a Big Number implementation for cryptographic computations, is that it was fairly simple and well documented written in the code file. The fact that it was well documented means it was easy to follow when manually reviewing the file before using. BigInt.js[22] contained all of the necessary functions
adding, multiplying, a modulo function and conversion between big numbers and strings. While the code was easy to follow is big factor in choosing it, the biggest factor is that it was correct. Since it was correct, it could be trusted to turn out the correct answers to the cryptographic operations.

Advanced Encryption Standard Implementation

Using AES.js[6] implementation of AES was a simple decision. This is because both AES.js[6] and aesCounter.php[5] implementations are from the same website meaning they are compatible with each other without question. There was some concern that switching languages could pose a problem when using long strings representing numbers. To minimize this risk, this implementation was used so that the logic of the program was the same.

Various Hash Functions

The PHP language has hash functions like SHA-1 already built into the language so they were used for convenience. Since the Javascript language does not have a built-in hash functions, [16][17] were used to fill this gap in the Javascript.

5.6.2 New Pseudorandom Number Generator

The Javascript pseudorandom number generator is not adequate for cryptographic use. Therefore a new one had to be created that is stronger than the current Javascript implementation. This presented a unique problem because nothing specific to the operating system can be used. This is because the application is an Internet device meaning it is only browser specific and not operating system specific. Therefore, no assumptions about the underlying operating system can be made. This means that utilizing devices like /dev/random, a device that provides truly random data for
seeding pseudorandom generators, cannot be used. In order to get the randomness, the timing of typing certain characteristics of the user’s typing habits were used to seed the pseudorandom number generator.
Chapter 6
Experimental Evaluation

This section lays out an evaluation plan for Split Auth, in terms of efficiency and usability. To do this evaluation, benchmarks would be inserted into the code so that timing could be done. The results would be compared to the normal PHPMyID[7] to see how they compare.

6.1 Benchmarks

The benchmarks will be used to time for how long the software will be executing.

6.1.1 Software Setup

Benchmarks are needed for the key generation within the register.php script. This needs to be benchmarked because this could take some time. Since there are no keys in the original PHPMyID[7], the key generation could be a huge point of delay.

6.1.2 Logging In

The entire logging in portion of PHPMyID[7] was completely changed, the entire login process should be benchmarked. This will allow a comparison of the login process for the regular PHPMyID[7] and the Split Auth version to be compared. Benchmarks after each piece of the login procedure can be used in addition to the benchmarking of the full process. Authorize_Mode will need benchmarks because it was completely rewritten. This means that the entire thing can run much longer than the normal PHPMyID[7] version of Authorize_Mode. Along the same lines, Validate_Mode should be benchmarked because it is completely new to PHPMyID[7]. Since it is new, the
mode could add a lot of time to the overall execution time of the IdP. At the present
time, it is unknown how much time it actually adds to the login time. Creating the
left side of the signature should be benchmarked along with the right side or the Split
Auth signature should be benchmarked. This is because the original login process
does not have these processes in it, so it should be benchmarked to see how much
more time it adds to the overall process. Lastly, the signature combination was no
included in the regular logging in process, so it should also be timed.

6.1.3 Updating Keys

Benchmarking the updating of the keys should be done because it is completely new
to the IdP. The benchmarking of updating of the keys should be benchmarked the
same way that the login process is done. It should be done as a whole and in pieces.
Both the Password_Delta.php and the Password_Reset.php need to be benchmarked
by themselves. This means that the parsing of the configuration files and the public
key update will be benchmarked. The only part that needs to be benchmarked in
login.html is the updating of the private key.

6.2 End To End

Perhaps the best way to benchmark the Split Auth software is to do end to end
benchmarking. This means that the entire program will be timed. This will give
the impression of how much longer the Split Auth software will run than the regular
PHPMyID[7]. The hope is that it will not take too much longer to execute because
users are generally impatient and would like to login as fast as possible.
6.3 User Evaluation

The only way to get a user’s true opinion of how they feel about the Split Auth system is to have them use it for a period of time. This experiment will take some time because they must use it for a longer period of time because not only will there be an adjustment period but also it would be beneficial to see how the user feels about the key reset functionality when their keys are in danger. A true opinion can only be received if their keys are actually in trouble. This may not happen right off from the start and may take some time to happen. As part of the experiment the user will be directed to download the software distribution and follow the instructions on how to set it up. The user will have to setup the software without the help of the designers. This will give an accurate description of how easy the software is to setup. If the software cannot be setup easily, there is a much less chance that the software will be used. If the user can set it up, then the user will be required to use the Split Auth to log into all the websites they sign into regularly. This is will allow the user to form an accurate opinion. Their opinion will be used to make any changes to make the software easier to use for users.
Chapter 7
Discussion and Conclusion

7.1 Discussion

Unfortunately the Split Auth software does not mitigate all of the issues associated with the regular OpenID protocol. For instance, the server can still impersonate the user because IdP can tell the RP that the user is logged in whenever the IdP wants to. This allows the IdP to masquerade the user without the user having any idea that it is going on. This can be easier for the IdP if it logs all of the RPs that connect to it and asks if the user is logged in or not. By logging all of the RPs that connect to it, it allows the server to know which RPs are susceptible to the IdP masquerading as the user. This is an acceptable risk to accept because in the end Split Auth is only a middle ground between the current authentication scheme of username and password combinations and having a public key at every website. In the later scenario, will become obsolete. This will alleviate the risk introduced by using the IdP. The user will only need the Split Auth server and their client machine neither of which will be able to masquerade as the user. Since the user is allowed to pick which server hosts the IdP, the hope is that the user only picks a server that they can trust in regard to this issue. The Split Auth implementation does mitigate the use of a password to authenticate a user. This is done by using a 15 character password and reducing the number of passwords from \( n \) to 2. By reducing the number of passwords and making them more complicated, the chance of them being discovered is a lot less in this scenario than in the original OpenID protocol. Even with the shortcomings of OpenID, it should be used because it allows for the new implementation to be
distributed with the user only having to do minimal work to set it up and it also allows more security for users. Lastly, it does not introduce a new protocol into the Internet.

7.2 Conclusion

As stated before the normal Internet user has too many username and password combinations to remember which leads to insecure passwords. Split Auth reduces the user’s username and password combinations from $n$ to 2. By no means is this a complete solution to the Internet’s identity management problem. Instead is a middle ground from the username password combination paradigm to the public key cryptography paradigm. In the public key cryptography paradigm, the user would have 0 passwords to remember but a new protocol would have to be introduced to the Internet. This scenario has the fault of having to replace all of the user’s public keys when there is a compromise which can be very cumbersome. Split Auth introduces a way that the fault can be overcome; by splitting the signature between two different servers. This means that the private key needs to be split in two, one for the client machine and one for the Split Auth server. Both parties will create half a signature before the client machine combines them. In splitting the key like this, the two halves of the secret key can be manipulated by adding a number to one part of the secret key and subtracting the same number from the other part so that the public key does not need to be changed. Split Auth is merely the middle ground between these two scenarios. This paradigm is useful because it will allow users to judge the new paradigm without having to implement the new protocol. By using the Split Auth software, a study can be done to see if creating a new protocol will be worth it in terms of users using the new protocol.
Appendix A
IdP Details

1. Accept_mode

(a) This will check to see if the user has accepted the trust of the Relying Party previously or not and allows the user to accept the trust for the given URL. First, the mode checks to make sure that the mode is not being accessed directly. If it is, error 500 is thrown. It then creates a user session. If the user has accepted it, PHPmyID will go to the original http post. If the user has not trusted the client, it will redirect the user back to the client. If neither has been done, the user is asked to make a decision.

(b) This mode is not accessed directly from login.html

2. associate_mode

(a) This creates an association with a RP. It creates the keys once the user has been logged in successfully. First it makes sure that an associate request has been given to the mode. Next, it gets the association type, session type, session type, modulus, generator, the RPs private key, and how long the keys lifetime will be. Then the key generation starts. It creates a public and private key pair for the Diffe-Hellman key exchange. The keys are then encoded as base64. Then the keys are returned to the calling function.

(b) This function is not accessed directly by login.html but it is accessed by the relying party.
3. authorize_mode

(a) This is a function that was modified to be able to use Split Auth.

(b) First the mode sets the username and the configuration file. Then it opens the file and gets the username from the file. If it matches the mode will then get x, L, y, the salt of the password, and chooses a challenge at random. It then sends that information back to login.html and closes the configuration file.

(c) This mode is explicitly called by login.html.

4. validation_mode

(a) This mode is completely new to allow Split Auth to work.

(b) First this mode sets p, q, and g. Then it gets a, z, y, and the challenge from login.html. It then hashes p, q, g, y, z, a into a long string of hashes using sha1. Then it hashes the one long hash string so that it is a normal 20-byte sha1 output. The the signature from login.html is validated. first it is made sure that z < q. Then \( a^q \mod p \) is checked to make sure that it equals 1. Then \( g^z \mod p = (a \cdot (y^z \mod p)) \mod p \). Lastly, the challenge that was stored in the session variable is checked against the one that is sent to the validation_mode to see if they are equal. The calculations are completed with the stored session challenge and not the challenge that is sent from login.html. If it were the other way around, a replay attack could be possible. If the signature is validated, the the profile is authorized and the user is redirected back to where they came from. If the signature is not authorized, the username is unset and the profile is unauthorized.

(c) This mode is explicitly called by login.html.
5. Cancel_mode

(a) Allows the RP to cancel a request. it will then display a cancellation message.

(b) This is not explicitly called by login.html.

6. Check_Authentication_mode

(a) This checks if a user has been authenticated by request from RP. To do this it gets the association handle, signature and signed information. It also gets the key pair used before and computes the signature on the shared secret. It is then compared to the one that was already used. The key pair is stored in signed stuff and extrapolated out. It then checks if the user has been authenticated

(b) This is not explicitly called by login.html.

7. CheckID_mode

(a) This handles the RPs request to see if the user has been logged in. The function takes a boolean in as a parameter. This boolean will allow the user to login if the user is not already logged in. It must get the id, association handle, trusted roo of the URL, and the sreg required and optional OpenID information. It must determine if the return to URL is a descendant of the trusted root by using the url_descends function. Allows the user to explicitly trust the return to URL if the user is running with paranoid variable set. If so, the user has to enter accept_mode to accept the trust on the return to url. This means that it must exit the check_id mode. If does not exit, it must make sure that the identity is correct for
the user if not the mode is exited and goes into error_mode. If the user is not authorized yet, control is passed to authorize_mode after necessary setup. If the user is logged in, it checks the validity of the shared secret. If expired, a new shared secret is created. Then it combines the keys to be sent, signs it and redirects the user back to the return to URL.

(b) This is not called by login.html explicitly.

8. CheckID_Immediate_Mode

(a) Checks to see if the user is already logged in. If they are not logged in, do not allow the user to log in. This sends the wait variable as false.

(b) This is not explicitly called by login.html.

9. Check_id_setup_mode

(a) Check to see if the user is already logged in. If they are not logged in, allow them to login. This means that wait is true.

10. error_mode

(a) Handles the errors. If there is an error URL, the mode will take the user there. If not, it will throw error_500.

(b) This is not explicitly called by login.html.

11. ID_res_mode

(a) Shows a user if they are logged in or not. If the profile has been authorized, display logged in message on the screen. If not, display not logged in message on the screen.

(b) This is not called by login.html
12. login_mode

(a) This mode was modified to be better suited for Split Auth

(b) First it sets the session username to the one given. If the profile is authorized, it will tell the user they are already logged in. If the session return to variable is set, the user will be returned to that address after logging in. Otherwise, the user will just be told that they are logged in. The latter option is used for static logins.

(c) This mode is called by login.html.

13. logout_mode

(a) Allows user to perform static logout. First checks if the user is already logged in. If not, it displays a not logged in message. If logged in, it will destroy the session through session_destroy(). This exits the mode. If the user doesn’t fit either of these, redirected to idp_url because the user is unauthorized.

(b) This mode is not called by login.html.

14. No_mode

(a) Displays the default information on the screen when no mode is specified.

(b) This is not called by login.html.

15. test_mode

(a) This does the testing for setup. It first checks the profile to make sure that the profile will allow the test. It then sets the duration of testing to be 180. All of the keys are hardcoded into the mode. It then goes through and
computes all of the Diffe-Hellman keys. It also does the MAC calculations as well. It then outputs the findings into an html table. This function is basically a way of making sure that the Big Math functions will work.

(b) This is not called by login.html.
Appendix B
Glossary

*Single Sign On* - A type of authentication system in which once a user is logged into a server, the user is no longer asked for their credentials.

*OpenID* - It is a decentralized, user-centric, authentication scheme that can span many operating systems [2][11]. It is also a single sign on framework for the internet scale. [2]. OpenID is common throughout many different services, including blogs, social networks. Users are able to choose where they store their credentials identity servers and the user chooses where they would like to store their login credentials. Allows the user to reduce the number of usernames and passwords down to one combination [11].

*Relying Party (RP)* - The website that the user is logging into. This is the party that wants to get the credentials of the user from the identity provider.

*Identity Provider (IP)* - The server that houses the user’s identity information. The IP also computes half of the user’s signature. This is the party that confirms the user’s identity. It sends successful login information to the Relying Party.

*SplitAuth Server* - The server that completes half of the signature. Conceptually, it is separate from the IP.

*XRI* - Extensible Resource Identifier. It is like a URL except it is unique an OpenID member Once a XRI is assigned to a member, it is there’s forever. URL’s can change
if related to things like blogs or social networking profiles.

**URL** - Uniform Resource Locator. This is how most network resources are found. For example, www.google.com will bring users to Google without them having to know the websites IP address.

**XRDS Document** - Extensible Resource Descriptor Sequence, an XML Document. This is the document that tells the Yadis Protocol where to discover the IP.

**Yadis** - Protocol that retrieves the XRDS Document from the the claimed source.

**Identity Management System** - Tool for managing a partial identities in the digital world[8].

**Service Provider** - A networked asset that provides a service for a user

**Federation** - Group of service providers that have a contract that allows users to use credentials from another service provider as long as the service provider is within the federation.
Bibliography


