THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

The University Login: Authentication for Web Applications – Implementation Comparison

Web Single Sign-On has been identified by the University as a major strategic direction we should be heading. This paper details the study undertaken by the Enterprise Architecture Team to identify an Open Source SSO application which would deliver on our requirements. After careful consideration of the results detailed below it is recommended the University use CoSign as the SSO application.

1. Overview

As part of the Simplified Sign-On project, it was found that web authentication, authorisation, and provisioning methods currently used within the University needed to be reviewed and greatly simplified. The document *The University Login: Authentication, Authorisation, and Provisioning for Web Applications* details this part of the project.

The current document details the study undertaken by the Enterprise Architecture Group to implement the authentication portion of Simplified Sign-On. Three candidate implementations[1] have been identified for consideration:

- CAS (Yale University),
- WebAuth (Stanford University), and
- CoSign (The University of Michigan).

2. Licence / Disclaimer

[1] The main criterion for selection was that the candidates must be open-source projects the University could acquire without any up-front costs.

PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

3. Table of Contents

1. Overview *1*

2. Licence / Disclaimer *1*

3. Table of Contents *2*

4. Goals *3*

5. Considerations *3*

6. The Results *5*

6.1 Central Authentication Service (CAS) **5**
6.1.1 Background Information 5
6.1.1.1 How it Works 5
6.1.2 Core Project Requirements 7
6.1.2.1 Critical Requirements 7
6.1.2.2 Desirable Features 8
6.1.3 Considerations 8
6.1.4 Other Features/Problems of Note 9
6.1.5 Overall Thoughts 9

6.2 WebAuth **11**
6.2.1 Background Information 11
6.2.1.1 How it Works 11
6.2.2 Core Project Requirements 13
6.2.2.1 Critical Requirements 13
6.2.2.2 Desirable Features 13
6.2.3 Considerations 14
6.2.4 Other Features/Problems of Note 14
6.2.5 Overall Thoughts 15

6.3 CoSign **16**
6.3.1 Background Information 16
6.3.1.1 How it Works 16
6.3.2 Core Project Requirements 18
6.3.2.1 Critical Requirements 18
6.3.2.2 Desirable Features 19
6.3.3 Considerations 19
6.3.4 Other Features/Problems of Note 20
6.3.5 Overall Thoughts 20

7. Results Comparison *22*

8. Conclusion and Recommendations *23*

9. Further Information *25*

9.1 Document History **25**

9.2 Resources **25**

9.3 Correspondence of Note **26**
9.3.1 CoSign deployment at The University of Michigan 26

9.3.2 CoSign Web Application to Kerberos TGT clarification 27
9.3.3 PeopleSoft Authentication 28

9.4 People 28

4. Goals

The primary goal of this project is to implement an efficient, robust, scalable, and easy to use (both for the End User and for the Web Developer) central web authentication system with Single Sign-On.

It is envisaged that one of the above implementations will best fit our needs, but it is highly likely some development will be required to both have it fully meet our needs and for some ongoing support and maintenance.

5. Considerations

The following table shows the main considerations for evaluating each implementation. These considerations are over and above the specific requirements for the system outlined in the design document *The University Login: Authentication, Authorisation, and Provisioning for Web Applications*.

Factor	Importance	Definition/Comments	Minimum Expectations
Resilience	High	Resilience is the application's ability to continue providing service in the event that one or more of its components fail.	The infrastructure must be resilient enough to continue working (although possibly in some degraded capacity, yet without sacrificing security) if it loses some of its components.
Efficiency	Medium	The efficiency of the application is its ability to perform its designed function with as little waste of resources as possible. Efficiency is related to throughput.	The application must be efficient enough to have a high enough throughput as detailed below.
Robustness	High	Robustness is a measure of how well an application can handle errors, whether user- or system-related.	The application must have a very good ability to handle errors. It must log all errors and warnings to allow system administrators to monitor them. The application must never elevate or allow a login when an error occurs.

Throughput	High	Throughput is the number of transactions the application can service within a given timeframe.	The throughput for this application must be high as it is envisaged this application will be used intensively. It must be able to service the peak authentication request load comfortably. It would be catastrophic if web applications around the University stopped working because the central authentication system could not service their requests.
Total Cost of Ownership	High	Total Cost of Ownership is the cost of installing, supporting, and maintaining the application and all supporting infrastructure within the University environment.	The Total Cost of Ownership must be as low as is reasonably possible, and at least meet the budgetary requirements for both the installation and its ongoing support.
Scalability	Medium	Scalability is how easily the application (either automatically or configured by a system administrator) can extend to use additional resources to meet a higher than expected load.	The application need not have an ability to scale automatically. The application must have an ability to use extra resources provided by a systems administrator in a timely fashion.
Supportability	Medium	Supportability is the measure of how easily the application can be supported when it is deployed into our production environment. Support is often the highest cost in using an application because of the long lifetime of software.	It is relatively important the application be easily supportable when in our production environment. It is important the application have sufficient logging to easily support user problems, etc. It is also important to have sufficient monitoring to gauge the application's current and historical ability to service user requests. Documentation of the architecture and installation, etc, is required.
Maintainability	Medium	Maintainability is the measure of how easily the application can be updated, patched, and bug-fixed, etc, during its lifetime.	The application must be reasonably maintainable. It is not a requirement that the application is a 'breeze' to maintain, but it must not be a labour-intensive and highly-specialised process. The application must also be relatively easy to administer on an ongoing basis.

6. The Results

6.1 Central Authentication Service (CAS)

6.1.1 Background Information

Developers:	Yale University
Website:	http://www.yale.edu/tp/auth/
Version evaluated:	Server – 2.0.11 Clients – 2.0.10
Server programming language:	Java
Clients (Web servers and client code) supported:	ASP Java Apache 1.3 and 2.0 PAM (*NIX) Perl Plsql Python MS ISAPI (IIS) (Experimental - The University of Indiana)

6.1.1.1 How it Works

CAS is based on the Kerberos model (see section 9.2 for further information on Kerberos). CAS uses an opaque session ID cookie (which is the Ticket Granting Ticket; TGT) that is only ever returned back to the CAS server. This cookie allows the CAS server to validate the user without challenging them to enter their credentials again. A Web application only ever sees its own Service Ticket (ST), which is associated (on the CAS server) to the TGT for the user.

The ST is a one-time-use-only opaque value that is invalidated when the web application verifies it for the first time. The ST is not designed as a session key for the application in any way; it must employ its own persistent state mechanism.

The CAS server is written in Java (Java Servlets and JSP) that must be deployed onto a J2EE-compliant application server (e.g., BEA Weblogic, Oracle, or Jakarta Tomcat, etc — see Section 9.2 for further information). CAS relies heavily upon the services the application server may provide, include any clustering, state replication, and load balancing. CAS does none of this by itself.

The CAS application is designed in a somewhat tightly-coupled design. The UI (User Interface) is separate from the core Servlets that handle everything including the cache of tickets and authentication to the back-end authentication server, etc.

The diagrams below show the components involved when a user attempts to access a CAS-protected Web application without having authenticated (Figure 1); by first authenticating to CAS (Figure 2), which also shows the SSO; and when the user continues to access the resource after authenticating (Figure 3).

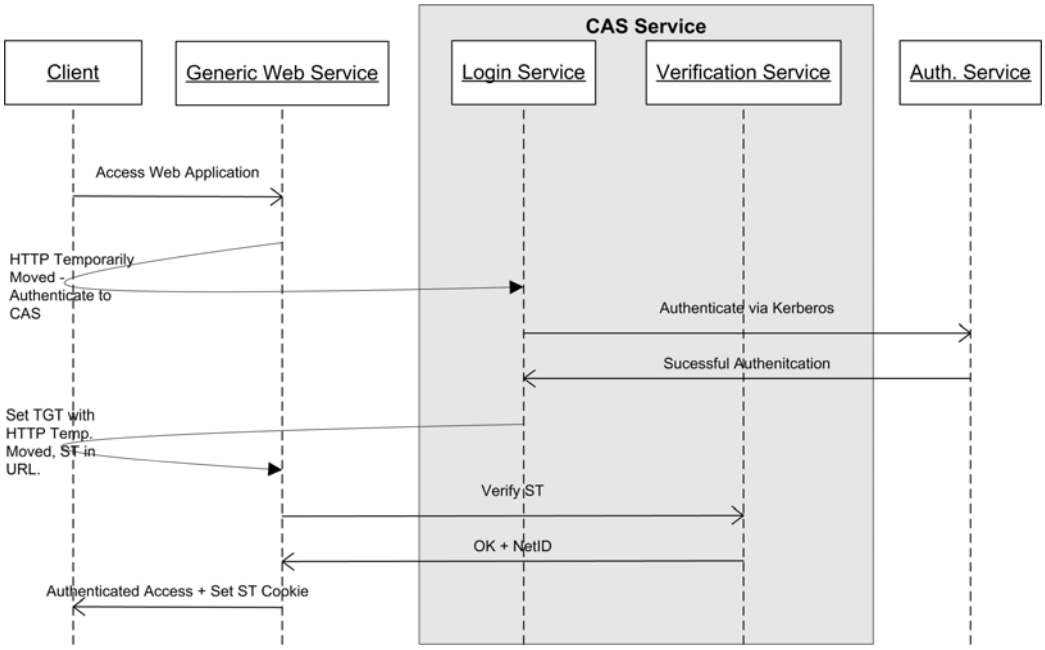

Figure 1: Unauthenticated access to CAS protected Web resource

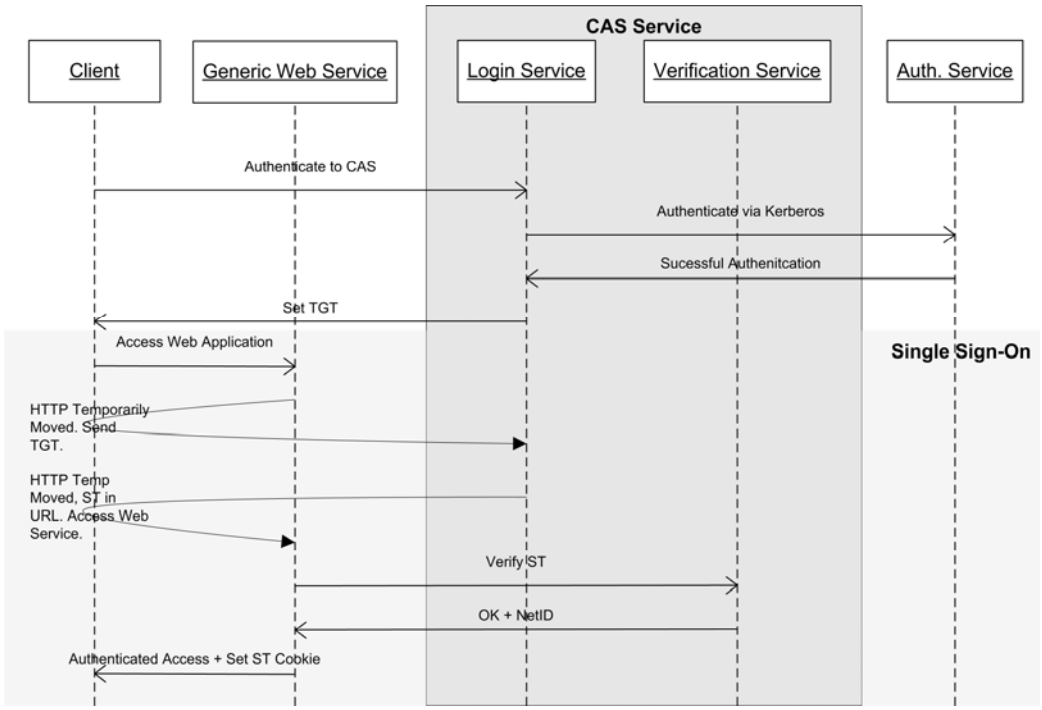

Figure 2: User authenticates to CAS server first

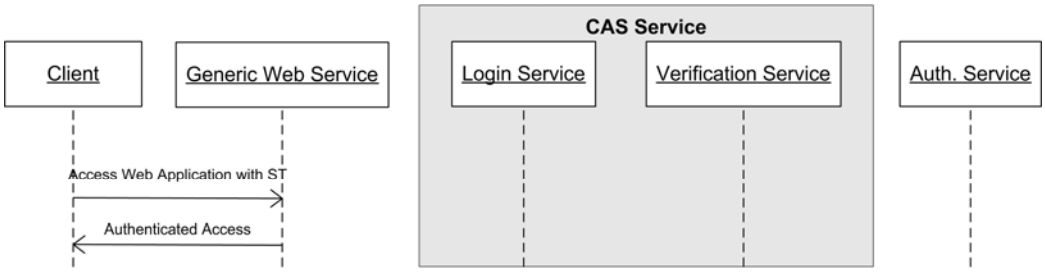

Figure 3: User's continued access to Web application after authentication

6.1.2 Core Project Requirements

This section details how well CAS meets the core design requirements outlined in the document *The University Login: Authentication, Authorisation, and Provisioning for Web Applications.*

6.1.2.1 Critical Requirements

Requirement	Measure	Comments
Secure Communication of user credentials.	Excellent	The cookies are sent only via HTTPS unless the system administrator specifically sets it up not to
Cached credentials do not contain user data.	Excellent	The cookies only contain session keys, which are mapped by the application to user data.
Cached credentials not easily re-playable.	Excellent	The service keys are only sent to the web server where the service is running. The authentication keys are only ever sent back to the CAS authentication server. For this reason, no services should run on the same server as the authentication web service.
User has a logout facility.	Yes	The user has the ability to logout, which invalidates their credentials. They must then re-authenticate if they wish to use additional protected web applications.
Timeout on cached credentials.	No	CAS implements no timeouts for either the TGT or ST. Having said this, CAS implements the ST as a one-time-use-only credential that is removed on first use.
Good logging on the authentication service.	Poor	There is minimal logging in both the server and CAS clients.
Support for Microsoft's IIS and Apache 1.3	Fair	CAS supports Apache 1.3 but currently only has IIS support in beta.

6.1.2.2 Desirable Features

Feature	Supported?	Comments
Web Application has ability to refuse SSO and force re-auth.	Partial	The Web application has the ability to force CAS to request the user to re-authenticate. The Web application will then consume the normal CAS ST like any other Web application. While this doesn't meet the desired feature fully, it does allow the application to centrally re-authenticate the user again.
Web server allows central auth to log the originating IP address during verification.	No	
Detection and minimisation of brute force attacks.	No	
Support for Apache 2.0	Yes	

6.1.3 Considerations

This section details how well CAS meets the considerations outlined in Section 5 above.

Feature	Measure	Comments
Resilience	Good	Given the tightly-coupled design of CAS it is difficult to have the system components resilient (because there are very few components). Any resilience CAS may attain is though the natural resilience offered by a best-of-breed J2EE application server.
Efficiency	Good	The code is clean and employs an in-memory ticket cache.
Robustness	Very Good	Java has excellent exception handling, which CAS employs to detect errors effectively and handle them appropriately.
Throughput	Good	CAS will inherit a lot of its throughput from the characteristics and tuning of the specific J2EE application server it is deployed in.
Total Cost of Ownership	Good	CAS may have a high cost of deployment given it reliance on the J2EE application server. It is thought the on-going support cost for CAS will be nominal.
Scalability	Excellent	The scalability of CAS is extremely dependent on the clustering the J2EE application server may provide. It has yet to be established whether CAS will support clustering, as it employs an internal state. Based on load the J2EE application server can deploy more Servlets can be automatically deployed to handle additional load.

Supportability	Very Good	• Most J2EE application servers provide very good tools to monitor the J2EE components deployed in it. Unfortunately, these tools do not extend within the component. The application server does provide centralised logging mechanisms. • There are no administrative web pages to see logged-on users, etc.
Maintainability	Good	The application will be easily patched and maintained, but it will require deployment to all of the J2EE application servers and updates to the clients used on the Web servers.

6.1.4 Other Features/Problems of Note

This section details any other features or problems of CAS worth mentioning and possible consideration when choosing the implementation.

Feature/Problem	Comments
IP Blocking on repeated failed authentications	CAS has the capability to block Ips for a short period of time after multiple authentication attempts. While this is a good thing, it is possible this could be problem on multi-user hosts, as everyone on the machine will be blocked from authenticating.
Use of J2EE Application Server	The J2EE application server may provide a lot of scalability, resilience, and throughput to CAS. Although commercial J2EE application servers are often very expensive (e.g., BEA Weblogic), there are (relatively) open source products such as Jakarta that are equivalently serious contenders.
Forced re-authentication	The Web application has the ability to force CAS to invalidate the TGT and request the user to reauthenticate. The Web application will then consume the normal CAS ST like any other Web application. This essentially allows the application to ensure CAS had recent contact with the user.
Session Key Size	The authentication ticket uses a 50-byte random number (pool size of approximately 2.6E+120). The service ticket is a 20-byte number (pool size of approximately 1.5E+48)
N-tiered support	CAS requires n-tiered applications authentication mechanisms to be 'pluggable'.
Support from CAS developers	The support received from the core CAS developers has been somewhat disappointing.

6.1.5 Overall Thoughts

CAS enjoys a good design. Its best feature is the ST being a one-time-use-only value: once the application validates it, it is thrown away and the application's normal session control takes over. This means when a user logs out of CAS, all existing STs can also be invalidated (because they have either been validated and thus destroyed or are still in the cache to be validated at some point, in which case they can be destroyed also). This still doesn't address the fact the user may still use an SSO-enabled application for which they have been validated, but this is not a problem because CAS is not a Single Logout service; most SSO implementations suffer this problem.

CAS supports proxied or n-tier authentication, but it has a very heavy-handed approach, as it requires the application to support customised authentication mechanisms. This requires one to be written for, at worst, all of the n-tier applications we use. Even so, most modern applications support Pluggable Authentication

Modules (PAM; where you can insert your own authentication modules without the application knowing or being concerned about it, see Section 9.2 for further information), or something similar.

CAS's heavy reliance on the J2EE application server is both a blessing and a burden. J2EE application servers can be very expensive (although there are a number of good open source implementations), but they do provide a good host of features. The University of Auckland has very little internal resource to deploy a J2EE application server (although BEA Weblogic is used as part of the PeopleSoft applications, it is relatively hidden in terms of servlet deployment and maintenance, etc).

The big draw-card and also the biggest possible problem to CAS is its use of Java. Java is a technology ITSS can support internally, but it can have speed problems, which is a big concern when the throughput of the application is a critical requirement. Speed problems, in these circumstances are often overcome by deploying well-written code in a well-architected environment. Under those conditions, Java can run as quickly and efficiently as implementations cast in other programming languages.

If CAS is deployed it is recommended the following work be undertaken to enhance the application and have it completely meet the University's requirements:

1. Choose the J2EE application server to be used in the deployment.
2. Ensure CAS will support clustering within the application server.
3. Add better logging to mod_cas and the Servlets to help maintainability, supportability, and to meet the *Invalid use of session key logged* requirement.
4. Implement TGT and ST timeouts to meet the *Timeout on cached credentials* requirement. The timeout on the TGT should both be an idle timeout and a hard timeout. The ST timeout should be a 'must be validated by the web application before *t*' kind of timeout.
5. Customise the JSP (UI) for The University of Auckland.

6.2 WebAuth

6.2.1 Background Information

Developers:	Stanford University
Website:	http://webauthv3.stanford.edu/
Version Evaluated:	3.2.2
Server programming language:	Perl
Clients (Web servers and client code) supported:	Apache 2.0 C/C++ API Perl API

6.2.1.1 How it Works

WebAuth uses a model similar to PeopleSoft's V8 Single Sign-On solution. This requires the WebAuth central authentication services (the WebKDC) to share symmetric keys with each WebAuth-Enabled Application Server (WAS) to encrypt or decrypt the tokens sent. Kerberos 5 and SSL are used to bootstrap and get session keys from the WebKDC.

Like the PeopleSoft design, WebAuth uses the cookie to store information about the authenticated user for consumption by Web applications (for example user id, creation and exppiry times etc). Unlike the PeopleSoft design, WebAuth uses ID and proxy tokens, which are consumed by the WAS and the main token is only consumed by the WebKDC. All of the tokens are encrypted with the appropriate shared key so only the recipient can read the contents. Since all of the data required is on the cookies, the WAS doesn't need to communicate with the WebKDC to validate the cookie and then get any user credentials.

All state data for both the WebKDC and the WAS are kept on the user's browser in the form of cookies. This makes the WebKDC and WAS stateless with respect to the WebAuth application.

The diagrams below show the components involved when a user attempts to access a WebAuth-protected Web application without having authenticated (Figure 4); by first authenticating to WebAuth (Figure 5), which also shows the SSO; and when the user continues to access the resource after authenticating (Figure 6).

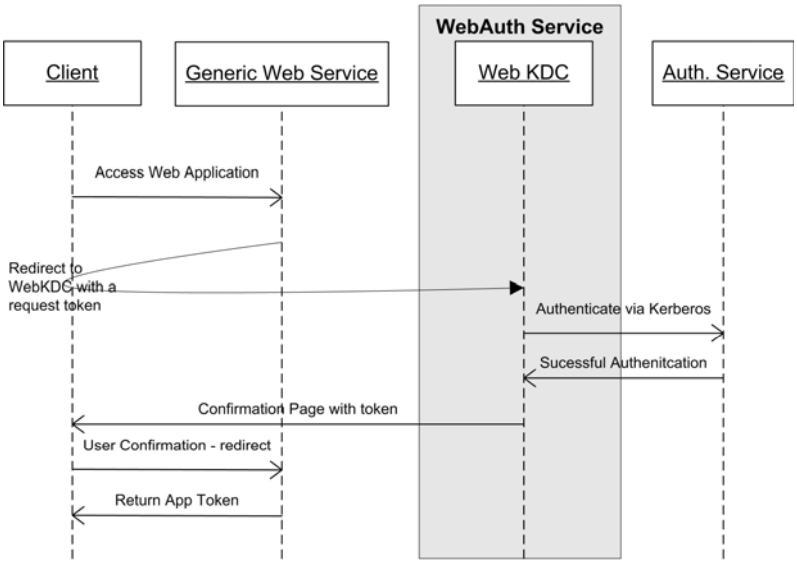

Figure 4: Unauthenticated access to WebAuth protected Web application

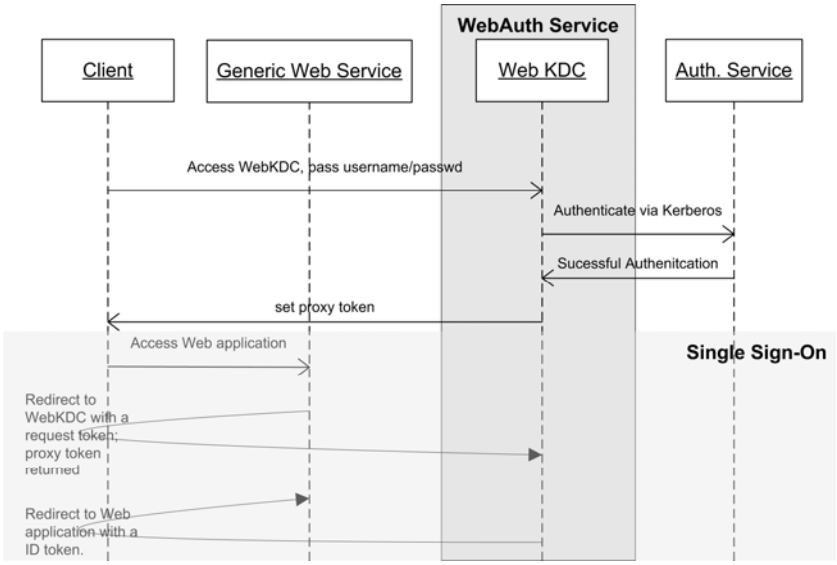

Figure 5: User authenticates to WebAuth server first

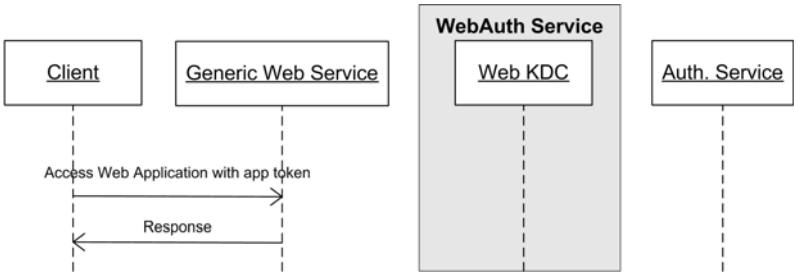

Figure 6: User's continued access to Web application after authentication

6.2.2 Core Project Requirements

This section details how well WebAuth meets the core design requirements outlined in the document *The University Login: Authentication, Authorisation, and Provisioning for Web Applications.*

6.2.2.1 Critical Requirements

Requirement	Measure	Comments
Secure Communication of user credentials.	Excellent	The cookies are sent only via HTTPS unless the system administrator sets it up not to
Cached credentials do not contain user data.	Poor	The tokens used by WebAuth contain cryptographically secure information about the user.
Cached credentials not easily re-playable.	Excellent	The tokens are only sent to the web server where the application is running. The authentication tokens are only ever sent back to the WebAuth authentication server. The timeouts for some tokens (like the ID and proxy tokens) are used to limit the ability to re-play these.
User has a logout facility.	Yes	The user has the ability to logout, which invalidates the credentials. They must then re-authenticate if they wish to use additional protected web applications.
Timeout on cached credentials.	Excellent	All the tokens have timeouts, which include hard and idle timeouts. Some also use 'must be used by' timeouts to limit the possibility of re-play attacks.
Good logging on the authentication service.	Fair	There is very little logging. Only fatal errors are being logged. Reasonable logging in the Apache 2 module.
Support for Microsoft's IIS and Apache 1.3	Poor	WebAuth doesn't support any of these platforms, and looks like it doesn't intend too.

6.2.2.2 Desirable Features

Feature	Supported?	Comments
Web Application has ability to refuse SSO and force re-auth.	Partial	The Web application has the ability to force the WebKDC to request the user re-authenticate. The Web application will then consume the normal tokens like any other Web application. While this doesn't meet the desired feature fully, it does allow the application to centrally re-authenticate the user again.
Web server allows central auth to log the originating IP address during verification.	No	
Detection and minimisation of brute force attacks.	No	There is no ability to detect a brute force on the private keys used to encrypt the tokens (although unlikely this will happen).
Support for Apache 2.0	Yes	

6.2.3 Considerations

This section details how well WebAuth meets the considerations outlined in Section 5 above.

Feature	Measure	Comments
Resilience	Fair	Given the tightly-coupled design of WebAuth, the WebKDC handles everything. Although, given the design, the WebKDC doesn't do any validation of the tokens (this is done by the Web server during the decryption of the token).
Efficiency	Good	The code is written in Perl, which is very efficient (even though it is an interpreted language). A potential problem is Perl tends to use more memory than, say, an equivalent C application would.
Robustness	Very Good	Perl provides for good error handling and WebAuth uses this to good effect.
Throughput	Excellent	Because the WebKDC doesn't do any verification of the tokens this is one performance penalty it will not pay. Because of the stateless nature of the WebKDC, this will give it a higher throughput also. The only possible drawback is the use of an interpreted language.
Total Cost of Ownership	Poor	The deployment cost for WebAuth may be quite high given the lack of required client support. It is thought the on-going support cost for WebAuth will be small but given it is in Perl it will be higher than Java.
Scalability	Excellent	Because the WebKDC is stateless (all state is stored in the cookies on the user's browser), WebAuth is highly scalable. Multiple servers can be operated with the WebKDC installed, and they do not require replication, etc.
Supportability	Fair	• There is very little logging to help with support. • There are no administrative web pages to see logged on users, etc.
Maintainability	Good	The application will be easily patched and maintained, but it will require deployment to all of the WebKDC servers and updates to the clients used on the Web servers.

6.2.4 Other Features/Problems of Note

This section details any other features or problems of WebAuth worthy of mention and possible consideration when choosing the implementation.

Feature/Problem	Comments
Use of Shared Symmetric keys	The use of Shared Symmetric keys will secure the data contained in the cookies, but if this key was compromised in any way (especially the WebKDC's private key) it would be catastrophic.
The Stateless nature of the WebKDC	Because all state for the WebKDC is stored on the user browser (although this has problems) this makes the WebKDC highly distributable and thus the WebAuth system highly scalable.
Ability to access Kerberos tickets	WebAuth allows for WAS access to Kerberos tickets, but doesn't appear to support the GSSAPI (see Section 9.2 for details) in the clients.

6.2.5 Overall Thoughts

WebAuth provides a number of good features, the most prominent of which is the scalability of the system through its stateless design. Because of the stateless design, the WebKDC can be highly distributed at very small cost to the service (because no replication is required).

Unfortunately this high scalability comes at a price to the overall security of the system, as it is considered the storage of data in the cookie which the WAS consumes is too risky, given browser hacks and cross-site scripting attacks to get some browsers to release their cookies to parties not entitled to them. Although CAS and CoSign use cookies, and are technically vulnerable to these attacks, their cookies contain opaque data from which it is more or less impossible for a hacker to obtain any information relating to the user or session. Also, WebAuth is at great risk if any one of the shared keys is compromised, which could give, in the worst case, access to the WebKDC authentication tokens.

The amount of client support for WebAuth is disappointing, as WebAuth supports only Apache 2 servers. The most notable omission here is Microsoft IIS support with Apache 1.3 a close second.

If WebAuth is deployed it is recommended the following work be undertaken to both enhance the application and have it completely meet the University's requirements:

1. Add better logging on the WebKDC.
2. Customise the HTML templates for The University of Auckland.
3. Develop IIS and Apache 1.3 support to meet the *Support for Microsoft's IIS and Apache 1.3 and 2.0* requirement.

6.3 CoSign

6.3.1 Background Information

Developers:	The University of Michigan
Website:	http://www.umich.edu/~umweb/software/cosign/
Version Evaluated:	1.5.1
Age:	
Server programming language:	C
Clients (Web servers and client code) supported:	MS ISAPI (IIS) Apache 1.3 and 2.0+ Java Servlet (In Development) Java/J2EE

6.3.1.1 How it Works

CoSign uses a model much like Kerberos, using the TGT to issue a ST, but its model may issue both the TGT and the ST before validating the user. It then associates these with internal state once the user authenticates.

CoSign uses a somewhat loosely-coupled design in which the CGI handles most of the user interface and the cosignd service handles the ticket cache while the monster process handles any replication and cache cleanup. Both the CGI and the cosignd/monster components need not be on the same host.

CoSign handles replication between multiple services via the monster processes, which walks through the TGT cache and replicates to known servers if required.

A CoSign-protected web application caches all the STs it has already validated so it doesn't need to access the cosignd service to validate every request to the web resource.

The diagrams below show the components involved when a user attempts to access a CoSign-protected Web application without having authenticated (Figure 7); by first authenticating to CoSign (Figure 8), which also shows the SSO; and when the user continues to access the resource after authenticating (Figure 9).

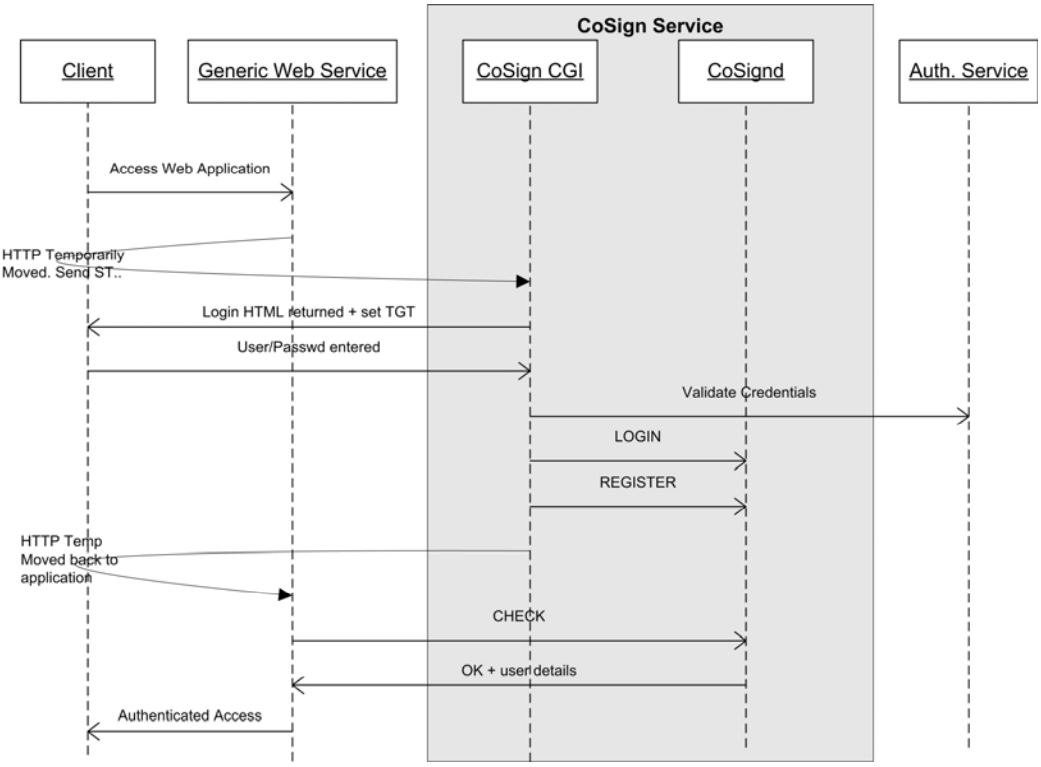

Figure 7: Unauthenticated access to CoSign protected Web resource

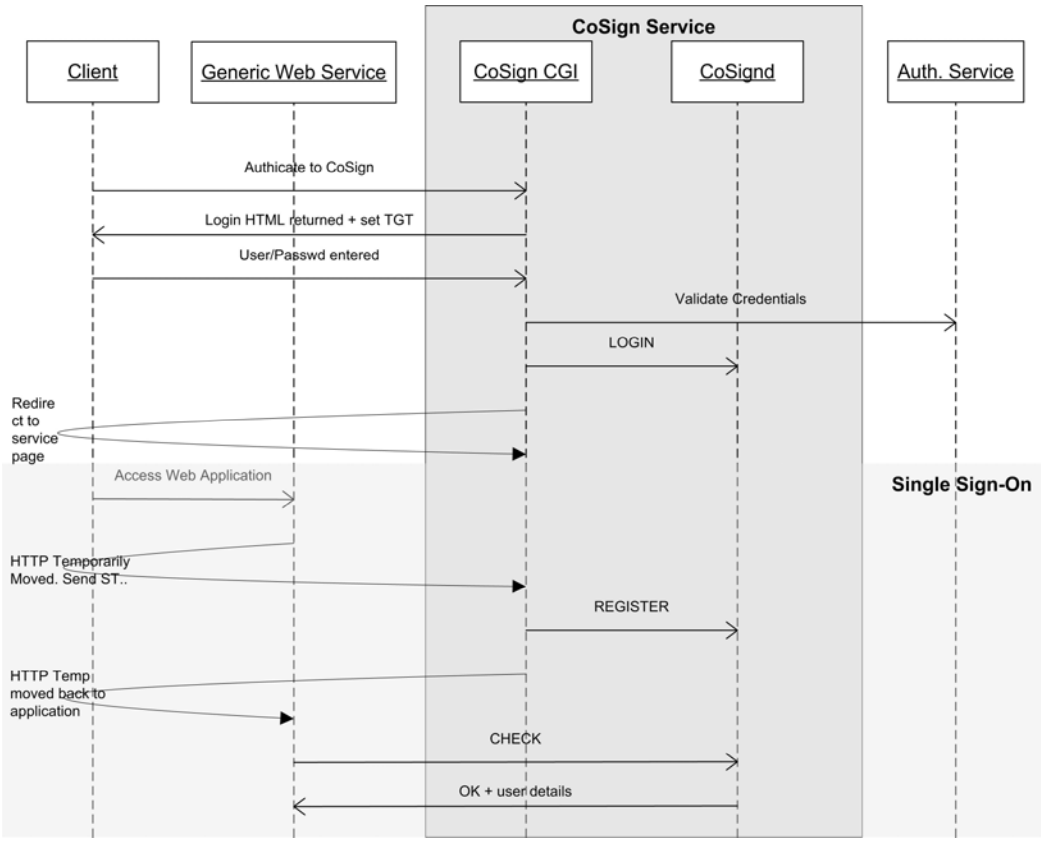

Figure 8: User authenticates to CoSign service first

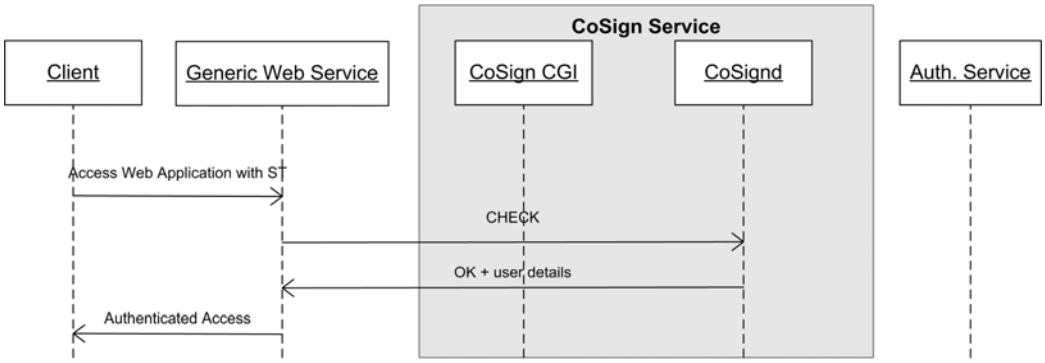

Figure 9: User's continued access to Web application after authentication

6.3.2 Core Project Requirements

This section details how well CoSign meets the core design requirements outlined in the document *The University Login: Authentication, Authorisation, and Provisioning for Web Applications.*

6.3.2.1 Critical Requirements

Requirement	Measure	Comments
Secure Communication of user credentials.	Excellent	The cookies are sent only via HTTPS unless the system administrator specifically sets it up not to.
Cached credentials do not contain user data.	Excellent	The cookies only contain session keys, which are mapped by the application to user data.
Cached credentials not easily re-playable.	Excellent	The service keys are only sent to the web server where the application is running. The authentication keys are only ever sent back to the CoSign authentication server. For this reason, no application should run on the same server as the cosign authentication server.
User has a logout facility.	Yes	The user has the ability to log out, which invalidates the credentials. They must then re-authenticate if they wish to use additional protected web applications.
Timeout on cached credentials.	Excellent	There is an idle timeout on the authentication cookie (which defaults to two hours) and an absolute timeout (which defaults to 12 hours).
Good logging on the authentication service.	Poor	There is minimal logging, and where there is, it varies in quality greatly.
Support for Microsoft's IIS and Apache 1.3	Excellent	Both are fully supported.

6.3.2.2 Desirable Features

Feature	Supported?	Comments
Web Application has ability to refuse SSO and force re-authentication.	No	The web application may get the following data from the CoSign server: • Login Name • Authentication Realm • Kerberos TGT.
Web server allows central auth to log the originating IP address during verification.	No	
Detection and minimisation of brute force attacks.	No	
Support for Apache 2.0	Yes	

6.3.3 Considerations

This section details how well CoSign meets the considerations outlined in Section 5 above.

Feature	Measure	Comments
Resilience	Very Good	If the components of the system are distributed properly, the system will be fairly resilient. Given the loosely-coupled design of CoSign the possible configurations for deployment are numerous and varied. Because the CoSign server has the ability to replicate data to other servers, they can be easily distributed for both resilience and throughput.
Efficiency	Good	All authentication cookies and service cookies are stored as files on disk. This could affect efficiency under high-load situations.
Robustness	Good	Once the application is up and running it appears to handle errors in a reasonable fashion.
Throughput	Excellent	The University of Michigan are currently using CoSign on three dual 2.8Ghz machines with 4GB RAM and are servicing approx. 255k ST registrations and 180k login requests per day.
Total Cost of Ownership	Good	CoSign may have a high cost of deployment given it some of the coding changes which may be required. It is thought the on-going support cost for CoSign will be small.
Scalability	Excellent	• All authentication cookies and service cookies are stored as files on disk. This could affect scalability, as there is a maximum number of objects the OS allows in a directory. • The monster process caters for the replication of the TGT and ST to other cosignd services running on other hosts. • The relationship of CoSign CGI to cosignd /monster services need not be one-to-one.
Supportability	Fair	• There is very little logging to help with support. • There are no administrative web pages to see logged-on users, etc.

Maintainability	Good	• The core of the application will be easily maintainable, it will require bug fixes/patching to all of the central CoSign services. • Patching/Fixing the clients and client APIs will be a little more difficult because of their distributed nature.

6.3.4 Other Features/Problems of Note

This section details any other features or problems of CoSign worthy of mention and possible consideration when choosing the implementation.

Feature/Problem	Comments
Fit in with Shibboleth framework	CoSign fits into the Shibboleth framework for the inter-institutional sharing of resources subject to access controls. Details on Shibboleth can be obtained from http://shibboleth.internet2.edu/
Ability to access the Kerberos TGT and support for GSSAPI	There is the ability for applications to get access to the user's Kerberos TGT so as to facilitate authentication to n-tier applications via the GSSAPI (see Section 9.2 for more). This is a big advantage because a lot of network-based resources allow GSSAPI authentication.
All cookies stored on disk	The filename used is the name and value of the cookie. This means a user *may* have the ability to obtain access to the disk on the server and even execute arbitrary code if the application doesn't successfully check for ALL possible 'bad' characters.
Very University of Michigan (UM) specific	The code is very UM-specific, and will require some code changes (in addition to UI customisations) to enable it work within our environment easily.
TGT Replication	With the monster process handling replication at intervals that default to 120 seconds, there is a possibility (if using DNS round-robin or a Foundry switch) that the TGT might not have replicated in time, and thus the user will get an error. While it is considered this will be fairly rare, it is a possibility.
'Friend' or guest access	CoSign allows a person who is not a member of the institution (if configured) 'friend' or guest access. These account logins have the form of an e-mail address. They are **not** self-service accounts: the account and password must have been created previously.
PeopleSoft support	The University of Michigan have a J2EE client working against their PeopleSoft application.
Session Key size	The service and authentication keys are a 93-byte base64 encoding of randomly-generated data. This gives a pool of approximately 9.3E+223 possible combinations.

6.3.5 Overall Thoughts

CoSign provides some good additional features, the best of which is the ability to use the GSSAPI for n-tiered applications. Having said this, CoSign also falls short on some of the critical requirements, the most concerning of which is the robustness, as the logging short-comings can be easily fixed. The robustness of the application fails most during the installation, when it may not be configured correctly. Once the application is correctly configured it appears to behave well.

One of the common flaws with centralised systems and especially authentication and authorisation systems is their rigidity. They do not allow people to authenticate who are not actually members of the institution but may need access to some of the resources it provides (e.g., a professor has papers which they require a person from

another university to collect, but do not want to e-mail them or just put them on their web page for all to get to). CoSign alleviates this problem by providing 'friend' or guest access. The logins are in the form of an e-mail address, so Web applications that do not want guest access can easily disallow them.

The biggest drawcard to CoSign is its IIS module as most of the Web applications at the University of Auckland are on IIS.

CoSign employs a loosely-coupled designed, which allows us to configure the components separately according to our requirements and our perceived loading on the application. This makes for excellent scalability and resilience of this application.

CoSign implements a local disk-based cache for all TGT and ST, both on the Web server and the CoSign server; it is considered this could be a reasonably big security risk especially when the cookie name and value are used to name the cache file on disk. While no exploits for this have been found (as it checks for certain bad characters), it is consider this is too big a risk to leave as it is.

Given the small group of developers within the CoSign community (approximately three at UM), it should be easier for us to get changes we wish to develop put back into the main distribution. If this were not possible, maintenance of the application will become increasingly more difficult (c.f., PeopleSoft Student Administration!). The support these developers provide is very good.

If CoSign is deployed then it is recommended the following work be undertaken to both enhance the application and have it completely meet the University's requirements:

1. Alter the codebase and configuration scripts to be more generic and configurable to environments other than The University of Michigan.
2. Alter the mod_cosign, cosignd and monster programs to cache the TGT and ST using either a database or an in memory shared table mapped to a file. This will improve security and, possibly, throughput.
3. Alter the logout process to invalidate all ST in addition to the TGT, or implement a one-time-use-only system for the ST, like CAS, to limit this problem.
4. Add better logging to mod_cosign and cosignd to help maintainability, supportability, and to meet the *Invalid use of session key logged* requirement.
5. Customise the HTML and M4 scripts for the University of Auckland.

7. Results Comparison

The results of the previous section have been placed into a ranking in order to produce a quantifiable comparison between the three implementations.
Each factor and section (e.g. *Critical Requirements*) is given an importance, which is used to weight the ranking(s).
The implementations are ranked against each other for each of the factors outlined earlier in the document, with the exception of *Additional Features/Problems* where the implementations are ranked on the quantity and quality of the additional features or problems. This table essentially is a rank of the weighted averages.

Table 1: Implementation Comparison Results

Factor	Importance	Implementation Ranking		
		CAS	WebAuth	CoSign
Ciritical Requirements	***High***			
Secure communication of credentials	High	1	1	1
Cached credentials not easily re-playable	High	1	1	1
Timeout on cached credentials	High	3	1	1
Microsoft IIS and Apache 1.3 support	High	2	3	1
Cached credentials do not contain user/session data	Medium	1	3	1
User logout facility	Medium	1	1	1
Good logging on authentication service	Medium	2	1	2
		3	**2**	**1**
Considerations	***High***			
Resilience	High	2	3	1
Robustness	High	1	1	3
Throughput	High	3	1	1
Efficiency	Medium	1	1	1
Total Cost of Ownership	Medium	1	3	1
Scalability	Medium	1	1	1
Supportability	Medium	1	2	1
Maintainability	Medium	1	1	1
		2	**3**	**1**
Desired Features	***Medium***			
Ability to refuse SSO based on security rating		Partial	Partial	No
Can centrally log originating IP on verification		No	No	No
Minisation and detection of brute forse attacks		No	No	No
Apache 2.0 Support		Yes	No	Yes
		1	**3**	**2**
Additional Features/Problems	***Low***	2	3	1
Overall Ranking		**2**	**3**	**1**

8. Conclusion and Recommendations

Each of the applications chosen for comparison has both strong and weak points to them. There are areas where the entire group do badly, which are the logging; administrative web pages, and; good distributed log out support. While the logging is the most concerning of this group, it is considered this can be easily fixed before deployment.

All of the applications fail to fully address the logout functionality. While each of the applications have a logout facility which in effect invalidates any internal state and the TGT cookie (or similar); they all fail to address any session which may have already been initiated on the Web applications through the SSO mechanism. While the CAS documentation states "…CAS is *not* a 'single sign-off' facility; a user that logs out of CAS will still have access to your application if your application keeps a persistent session with the user" this is considered to be an excuse. While there is no single solution to this problem, it needs to be highlighted to the Web application owner as a possible problem they may face.

The applications have a number of short-comings in their design, some of which can be fixed. The most concerning of these are the heavy reliance on a J2EE application server for CAS; WebAuth and its lack of Web server support, and; CoSigns use of the file system as a cache of cookies. It is important to note that while CAS relies heavily on the J2EE application server, this is only a problem for the University because we have no resource internally to fully support it and because of the often prohibitive purchase and maintenance costs (although there are viable free alternatives).

All of the applications do well when it comes to the resilience, efficiency and throughput which is pleasing as these are very important to the overall design of a central Single Sign-On framework. WebAuth does score lower in the resilience because it has a very tightly-coupled design.

N-tiered application support is quite important because a lot of Web applications are front-ends to the actual service (e.g., PeopleSoft 8+, Webmail, and nDeva). CoSign handles this scenario extremely well with its use of the GSSAPI (see Section 9.2 for more) with Kerberos. Most good applications allow for authentication via the GSSAPI. While WebAuth does allow the application to get access to the Kerberos tickets, it doesn't seem to natively support the GSSAPI. CAS requires the application support some form of pluggable authentication module or authentication exit.

The choice of programming language for the applications is very diverse, Java for CAS, Perl for WebAuth, and C for CoSign. Java is a fully supported language within ITSS, with internal resources and an existing development infrastructure. Perl and C are languages which do not have full support within ITSS; with only a small amount of resource in Perl, and a good amount of resource in C (both internal and NetAccount).

The total cost of ownership is considered to be about the same for each of the applications, except for CAS if a commercial J2EE application server is required for deployment. Each of the applications will require the same amount of hardware in

general and each of the applications will require roughly the same amount of development before deployment.

The support from the CoSign developer community has been very good, but the support from the other communities has been somewhat disappointing. The support the community provides us is very important in getting questions answered and also suggestions for further development.

Given the results of this study and the quantifiable comparison from Section 7, it is recommend the University use CoSign as the SSO implementation. This is for a number of reasons:

1. Superior results in the *Critical Requirements* and *Considerations*.
2. Superior developer and community support.
3. Superior Web server support.
4. Use of Kerberos and GSSAPI for n-tiered applications.

It is recommended CoSign be deployed using three machines, all replicated via the monster process. Two would be load balanced by the foundry switch (or something similar). The third machine can be put in a location on the network where it is considered there may be a large amount of local traffic. This is shown below:

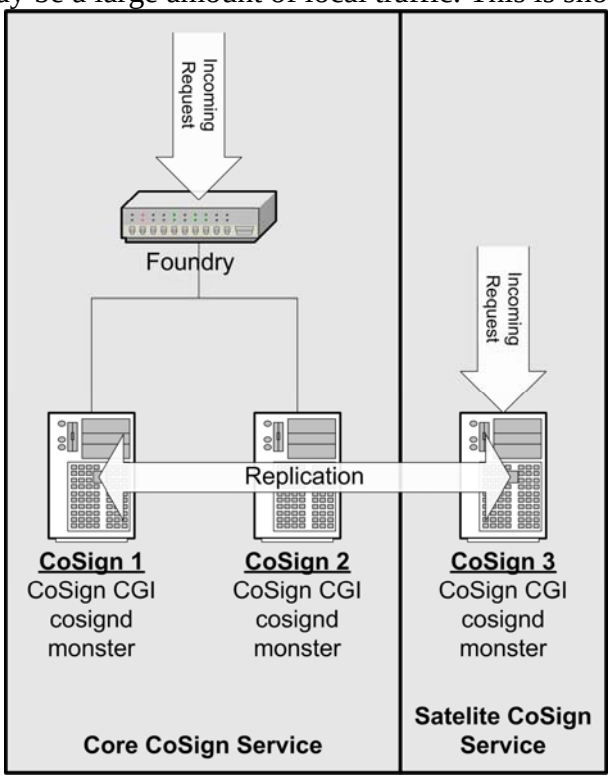

Figure 10: CoSign deployment recommendation

9. Further Information

9.1 Document History

Date	Version	Author	Comments
23/03/2004	0.1	Brett Lomas	Document Created.
02/04/2004	0.2	Brett Lomas	Document submitted to EAO for peer review.
06/04/2004	0.3	Brett Lomas	Changes after EAO peer review, the major of which are: • Added MS IIS and Apache 1.3 support for *Critical Requirements*. • Added "Total Cost of Ownership" to the *Considerations*. • Altered the *Result Comparison* section to give a better and fairer comparison.
07/04/2004	0.4	Creative Integrity	Minor changes.
08/04/2004	0.5	Brett Lomas	Final Draft.
16/04/2004	1.0	Brett Lomas	Public Release

9.2 Resources

- *The University Login: Authentication, Authorisation, and Provisioning for Web Applications* is located at \\petrarca\itarch$\Docs\projects\sso\WebSSODefn.doc
- Candidate implementation home pages:
  CAS: http://www.yale.edu/tp/auth/
  WebAuth: http://webauthv3.stanford.edu/
  CoSign: http://www.umich.edu/~umweb/software/cosign/
- The Web Initial Sign-On working group has a web site at http://middleware.internet2.edu/webiso/. Of particular interest are the submissions to the *WebISO Web Application Agent Questionnaire* of 3 Oct 2002 (http://middleware.internet2.edu/webiso/docs/webiso-questionnaire.txt) which are:
  CAS: http://middleware.internet2.edu/webiso/docs/waa-questionnaire/yale.txt
  WebAuth: http://middleware.internet2.edu/webiso/docs/waa-questionnaire/stanford.txt
  CoSign: http://middleware.internet2.edu/webiso/docs/waa-questionnaire/umich.txt
- Kerberos details can be obtained from:
  http://web.mit.edu/kerberos/
  http://www.ietf.org/html.charters/krb-wg-charter.html
  http://www.ncsa.uiuc.edu/UserInfo/Resources/Software/kerberos/krb5api/krb5api1.html
  http://www.faqs.org/faqs/kerberos-faq/
- Java 2 Enterprise Edition (J2EE), Java Servlets and Server Pages documentation can be obtained from Suns J2EE website http://java.sun.com/j2ee/index.jsp
- Some J2EE Application Server implementations and their web pages are:
  Sun: http://wwws.sun.com/software/products/appsrvr/home_appsrvr.html
  BEA Weblogic:
  http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server/

Oracle: http://www.oracle.com/appserver/
Apache's Jakarta Tomcat: http://jakarta.apache.org/tomcat/index.html

- Information about Pluggable Authentication Modules (PAM) can be obtained from http://wwws.sun.com/software/solaris/pam/ or http://www.freebsd.org/doc/en_US.ISO8859-1/articles/pam/
- Generic Security Services Application Programming Interface (GSSAPI) information can be obtained from http://www.faqs.org/faqs/kerberos-faq/general/section-84.html
- Simple Authentication and Security Layer (SASL) Information can be obtained from http://www.faqs.org/faqs/kerberos-faq/general/section-85.html

9.3 Correspondence of Note

9.3.1 CoSign deployment at The University of Michigan

```
On Mar 28, 2004, at 9:17 PM, Brett Lomas wrote:

> Hi Kevin,
>
> How are things? I have just one quick question for you if you are able
> and willing.

always willing, generally able. :)

> In your last email you detailed that you have CoSign running on 3
> linux boxes. How have configured these? More specifically I am really
> asking if each of these boxes run the CGI and cosignd/monster or have
> you separated them out? Also how have you configured CoSign web
> servers to talk to these machine? via DNS round-robin or something
> else?

Each of the 3 central weblogin servers is running Apache, the CGI, cosignd, and
monster.  Additionally, they host /cgi-bin/logout and /services/ (our rudimentary
"service menu").

The services are configured to use "weblogin.umich.edu" as the weblogin server and
they convert this name into all of its A records to establish connections for 'CHECK.'
We do not actually use round-robin at the service level as the goal here is
thoroughness.  However, user's are directed to one of the 3 weblogin servers for LOGIN
and REGISTER using DNS round robin.

Kevin
```

```
On Mar 23, 2004, at 8:24 PM, Brett Lomas wrote:

> Hi Kevin (and others :) ),
>
> I have a few more questions about Cosign:
>
> 1. Why was it decided to store the cookies on disk? There are a couple
> of reasons I ask this question. Linux/Unix default to only allowing
> around 65500 objects under a single directory. This in itself wouldn't
> normally be a problem, but because the cosign and cosign service
> cookies (for the
> daemon) are stored in one directory; this will fill up very quickly.
> If it
> is suggested we go ahead with CoSign I will be suggesting to
> management that we spend some time changing the storage to either a
> shared memory table (because the cosignd and children have a shared
> parent) OR change it to use a DB backend for storage (and it may also
> possible handle the replication to other cosignd server databases).
> What do you think of these?

These are completely justifiable concerns, I had them myself when we started.  We
chose the filesystem because:  it was simple to write, easy to debug/test with
standard unix tools, its contents survive a reboot, and it was adequate for cosign's
needs.  At peak load we're seeing approximately 120,000 objects in /var/cosign/daemon
on our cosign servers.  I'm certain there's a hard limit (for files in a directory),
but in our previous Solaris deploy we found that we ran into performance problems long
before we hit a real limit. In our current Linux deploy we've yet to see any
noticeable slowdown.  If someone does encounter such a slowdown, there are, as you
point out, several options for addressing the situation.  This is the forum in which
those alternatives would be discussed/hashed out.
```

> Also having the
> files named the same as the cookie is big security risk (in my
> opinion)...
> it could possibly lead to exploits, although I am still doing my
> security review and have not found a concrete example yet.

I welcome the review and am eager to learn what you discover. I feel confident that we've explored the possibilities here (note the checks in daemon/command.c for '/'), but having someone else explore them too is fantastic. What sort of security vulnerabilities are you concerned about?

> 2. What size is your deployment at UM? Can you give me some stats,
> like the hardware you are using and the number of authentications etc
> you service a day. Only if this is not too much of an effort on your
> part, because I will be stressing the application myself, I am just
> curious.

So far this month we are seeing approximately 180,000 LOGINs per day on a monday - thursday, as few as half that on a weekend day. These are LOGIN events, of course, not unique individuals. These 180,000 weekday LOGINs are associated with approximately 32,000 LOGOUT events.

We REGISTER approximately 255,000 service cookies on an average weekday. 120,000 of these are for mail.umich.edu. The rest are divided among the roughly 50 other cosign-protected services in active use on campus.

We're running our weblogin service on three 1u linux boxes. Each machine has dual 2.8 Ghz Xeons and 4 gigs of ram. We're using hardware this (relatively) "beefy" only because it was pretty much the cheapest thing we were willing to deploy a service on. We have three of them so we can have the service physically located in multiple server rooms.
Load average on these machines is typically around .2 (point two). :)

> 3. Have you successfully deployed CoSign to an n-tiered application? I
> specifically ask because will needing the chosen implementation to be
> able to SSO to our Cyrus IMAP server via Horde/IMP

We have several N-tier applications deployed:

- o afs.umich.edu -- a web-based AFS file manager, Horde's gollem running with the user's AFS token.
- o kpasswd.cgi -- kpasswd, but a cgi
- o mail.umich.edu -- IMP, see below
- o directory.umich.edu -- gui client to our ldap directory service
- o flume -- our web log analysis software (runs reports of user's web statistics and writes the report to the appropriate directory in AFS).

mail.umich.edu is our most popular web app right now, seeing upwards of 15,000 simultaneous users at any given moment on weekday afternoons with, as I said above, roughly 120,000 users per day and 60,000 - 70,000 unique users accessing the service per month.

We made a few changes to mod_cosign a couple of years ago to correctly set up the GSSAPI environment that IMP/c-client needed. Otherwise IMP should just work out of the box with Cosign. I can put you in touch with Liam Hoekenga, the person primarily responsible for our IMP installation, if you have specific questions there.

Kevin

... "In, as you say, the mud." ...

9.3.2 CoSign Web Application to Kerberos TGT clarification

Hey,

Yes Johanna's reply was good (thanks Johanna, I didn't get back to you on this; I haven't had the time to test the bug fix, and I may not until it is decided to with CoSign), and it is pleasing to see the turn around for bug fixes is excellent!

The Kerberos issue is not such a problem, I always assumed it was the TGT, but the submission you made to the WebISO Web Application Agent Questionnaire seemed to suggest otherwise (although it is possible I misread it).

I think it would be nice to, as you suggested, implement a finely grained approach as opposed to just a yes/no type of authorization. Like you say, the approach would be to have a list of services the application can request ST for, and possibly a special tag as 'tgt' to allow the application to actually get the TGT for the user as well.

BTW: If we do go into production with CoSign, we will be very willing to implement changes which we think are good, and submit them back into the CoSign main release if they are of benefit to anyone else, we will not be expecting you guys to implement changes we need :).

```
Cheers

Brett

-----Original Message-----
From: kevin mcgowan [mailto:clunis@umich.edu]
Sent: Thursday, 1 April 2004 6:36 a.m.
To: Brett Lomas
Cc: cosign-discuss@umich.edu
Subject: Re: Kerberos Tickets

On Mar 30, 2004, at 10:20 PM, Brett Lomas wrote:

> Thanks for the reply on the hardware and all, very helpful.

glad to help.  I trust Johanna's reply was helpful too?

> CoSign and Kerberos question. When an application requests a Kerberos
> ticket (the RETRIEVE command to cosignd) it appears to be allowed to
> specify the ticket name (eg imap/imap.auckland.ac.nz@AUCKLAND.AC.NZ).
> This looks to be a service account (in the examples I have seen), does
> this mean that a service ticket is passed back to the application, and
> not a/the TGT the cosign CGI obtained to authenticate the user?

That's the eventual plan, Brett, but currently it is the TGT that is returned.  You'll
note the 0/1 in cosign.conf to determine whether a service can request Kerberos
credentials?  In theory this could eventually be a list of services for which a
service is allowed to request service tickets (e.g. mail can ask for imap, directory
can ask for ldap, etc.).  We were sure, during early development, that not
distributing the TGT would be a major feature requirement.  So far it just hasn't been
(for us, anyway).

Is this a make or break feature for your site?

Kevin
```

9.3.3 PeopleSoft Authentication

The following was an email to the CAS mailing list. It details an attempt to work on producing a CAS module to authenticate to PeopleSoft applications. After this email it was discovered PS 8 allows for a *Web Server Security Exit* which is where the PS application server trust the web server to authenticate the user and pass the username to PS through PeopleCode (See PeopleBooks for more information).

```
Date: Mon, 01 Mar 2004 10:56:41 -0700
From: Chris Michels <Chris.Michels@NAU.EDU>
Subject: BEA WebLogic Identity Assertion Provider
To: cas@tp.its.yale.edu
Message-ID: <6.0.0.22.2.20040301105329.024a2a68@mailbox.nau.edu>
Content-Type: text/plain; charset=us-ascii; format=flowed

Has anyone done any work on a Identity Assertion Provider for BEA WebLogic that uses
CAS to authenticate?  If so, any advice or code you are willing to share would be much
appreciated.

We are trying to get WebLogic to use CAS and then have PeopleSoft trust WebLogic's
authentication.


-- Chris
```

9.4 People

- Tim Chaffe, ITSS Enterprise Architecture Manager, t.chaffe@auckland.ac.nz
- Brett Lomas, ITSS Enterprise Architecture Office, b.lomas@auckland.ac.nz
- Creative Integrity Ltd, support@creativeintegrity.co.nz

[end of document]