Extended Authentication Walkthroughs

In most cases, a user authenticates to CAS today by submitting a form over SSL containing a Userid and Password. Internally CAS presents the userid and password to some backend authentication system (Kerberos, LDAP, ...). CAS then generates a ticket and a transient cookie transmitted over SSL to be stored in Browser memory. In subsequent transactions, CAS may use the cookie as the authentication token rather than forcing the user to reenter the Userid/Password.

The purpose of CAS is to perform a single signon to the network using a closely guarded credential (your Netid password), and then through CAS gain access to a large number of network resources managed by possibly less trusted administrators. However, CAS operates in a complex environment of operating systems and network protocols where there may be several existing reliable authentication mechanisms. If you have already obtained a Kerberos ticket, or logged on to Windows Active Directory, or installed an X.509 certificate from the campus Certificate Authority, or carry a Smartcard, then any of these credentials can be trusted by CAS as sufficient proof of identity. The challenge is to expand the CAS 3 architecture and Web interface design to allow these and any other reliable proof of identity.

CAS is not an authentication architecture or network protocol. It is a deployable system in which the user interaction has been carefully considered. However, it is open source and easily reprogrammed, so institutions that have other opinions can change the look and feel. It would be controversial and generate support issues if CAS were to automatically initiate some complex challenge or negotiation protocol with every Browser. Therefore, if CAS 3 is to bypass the logon form and try to automatically log a user in using existing foreign credentials, it will be based on some tests that can be made on the initial Request object just as current CAS bypasses the form if the Browser presents a Cookie indicating that the user has already logged on. For credentials that cannot be detected automatically, there may be a button on the Form suggesting that, as an alternate to presenting a userid and password, you want CAS to "Use my existing Windows logon" or some such thing. CAS may optionally write a long term persistent configuration cookie to the Browser that could subsequently be detected before presenting the Form indicating that CAS should "try Windows NTLM authentication non-interactively before displaying the Form".

CAS 3 can at best provide a general extensible architecture. Even it if were possible to completely enumerate all currently available third party credentials, so other mechanism will be created tomorrow. There are some sources that enumerate authentication mechanisms in wide use (OASIS SAML 2.0 standards) but even then they intentionally omit politically disfavored options (Microsoft Passport). CAS is customer extensible and should have no bias for or against any authentication system.

The traditional use case for CAS was interactive access to a Web service by a Browser. However, there is an alternate formulation of the CAS function. CAS takes one form of authentication (permanent password, Certificate, key, token) and generates a transient, one-time-use character string token (the ticket) that can authenticate against a particular service. CAS also provides an API through which almost any service written in any language can authenticate the ticket. This has a number of advantages that have nothing to do with Browsers:

It converts a closely guarded secret (the permanent password) into something that doesn't have to be so carefully protected and which can be transmitted to untrusted services.
It converts complex authentication tokens (and X.509 certificate) into a simple string that can be embedded in existing protocols or message formats that only have room for a password.
It converts a broad variety of complex network authentication protocols (Kerberos, X.509) into a single very simple protocol (or maybe a family of protocols) that can be used by any service written in any language. Once the application is "CAS-ified", it inherits any sophisticated authentication mechanism that you can add to CAS without the application becoming any smarter.
CAS already supports a variety of standard front-end interfaces (Filters, PAM) so that applications can be automatically protected by CAS without writing any new code.
These features are meaningful to any client-server relationship. When proxy CAS is used by a Portal, some version of CAS function is used by backend service protocols (IMAP) that are unrelated to Browsers or HTTP. Thus, while CAS must have a Web based HTTP Servlet client interface, it may also develop a Web Services SOAP or other forms of client request interfaces.

It is important to distinguish between the actual mechanism of authentication presented by the user and the use of any network services by CAS for its internal housekeeping. For example, if we were to say that CAS authenticated a user through Kerberos, we would mean that the user had logged on to a real KDC and had obtained a real Kerberos Service Ticket and presented it to CAS. It the user fills in a userid and password in a form, and CAS happens to validate the password using the Kerberos database (instead of using LDAP, JDBC, or an JAAS backend service), then this is still "password authentication" because the user presented a password to CAS. It doesn't become a Kerberos authentication even though Kerberos happened to be used to validate that password.

Flows

There appear to be three basic models of CAS interoperation with an external authentication source.

1. Passive - In this model, CAS operates behind a Servlet Filter or Container Based authentication system that validates the user automatically and which CAS can accept implicitly. When CAS gets control, the validated user is already stored in the ServletRequest object where it can be fetched through operations such as getRemoteUser(). CAS trusts the decisions already made and simply transfers this external data to tickets.
2. AuthHandler - The information needed to authenticate the user is stored in the ServletRequest (as Form data, or in HTTP Headers) but it has not been processed and validated yet. The data needs first to be extracted from the Servlet control blocks through the Web interface layer and turned into a "Credentials" object, and then later the Credentials are processed and validated by an AuthHandler under the control of the Authentication Manager.
3. Redirect - CAS begins processing, but discovers from information in the Request or data entered into the Form that an external authentication service is need. For example, the user accessing a resource at Yale types into the form a Netid of "somebody@rutgers.edu". CAS has started processing and is hardly passive, but now it discovers that it needs to use either Shibboleth or Windows 2003 R2 ADFS to perform the actual logon. So it can redirect the Browser to another network URL where the required logon can be complete and the necessary credentials obtained. Eventually the Browser is redirected back to CAS, but now with an attached set

of credentials sufficient to successfully log the user in to CAS non-interactively. Although it is something of a degenerate case, this situation can be modelled by assuming that there are two CAS servers, and the first server redirects the Browser to the second CAS where the user logs on. The the Browser returns to the first CAS server with a ticket from the second CAS, and the first CAS validates the ticket and issues its own ticket. This scenario applies to any situation where the two CAS systems know each other, but a protected application is configured to trust one CAS while some group of users accessing the services has to log on through the second CAS.

Principals

When pressed to define the technical term "Principal", most computer professionals will try to say that it identified a particular individual. Unfortunately, in reality a Principal is the identifier for an account in a particular type of authentication database. While corporations and universities may aspire to have a single identifier that represents the same individual to all systems, a combination of history, conflicting technical standards, and vendor implementation details forces some differences.

A Yale Netid is a simple unqualified identifier such as "gilbert". Originally this was a logon name manually defined to a set of independent computer systems (IBM mainframes, DEC VAX, Unix, Windows NT, ...). Today it is maintained centrally by a combination of database tables and a Kerberos KDC. Technically, in its Kerberos use it is an abbreviation of a principal qualified by a realm (gilbert@NET.YALE.EDU).

As the simple Windows NT4 domain became Active Directory, Microsoft attempted to integrate the old domain structure, Kerberos 5, LDAP, and PKI within a single administrative mechanism. This exposed conflicts in the view of "principal" implied by all these standards.

The old simple domain account name of "gilbert" (essentially the same as a mainframe or Unix account name) became the "legacy" or "pre-Windows2000" identifier. In the worst case, it is the SAMAccountName (its LDAP attribute name in AD).

There is also a Kerberos principal-realm name. In the Unix based Yale KDC, this is gilbert@NET.YALE.EDU, but Active Directory maintains a separate parallel tree of realms in which the corresponding ID is maybe gilbert@YU.YALE.EDU.

When Kerberos is installed by itself, then there is either a single institutional realm or any additional realms are introduced only in response to carefully engineered enterprise requirements. However, Active Directory creates a realm for every migrated domain, and in many institutions there is a domain for each major department. Although Kerberos has always supported cross-realm authentication, it has traditionally been a exotic option that is seldom used. Until Microsoft came along and made cross-realm, within Active Directory, the primary method of doing business.

The collection of all interconnected Kerberos realms in Active Directory is called the "Forest". Because the particular realm in this entire tree that happens to be home to "gilbert" is an local convention, Microsoft wanted to create a "Univeral Principal Name" (UPN) that provided a Principal identifier that was Forest-wide. The UPN could be "Howard.Gilbert@yale.edu" or "gilbert@people.yale.edu". It requires an individual identifier, an "@" separator, and then any DNS domain suffix that is within the scope of the Forest.

When Microsoft releases its cross-institutional Active Directory Federation Service as part of the "R2" refresh of Windows 2003 Server, probably in the Summer of 2005, then the UPN will be the global identifier by which a user at Yale would identify himself to services at federated institutions. Presenting "Howard.Gilbert@yale.edu" to an Active Directory at Rutgers would, given the additional configuration services of R2, allow the two institutional Forests to map out a string of realm connections and allow someone logged on as principal-realm "gilbert@YU.YALE.EDU" to obtain a string of cross-realm service tickets ending at a server computer in the rutgers.edu Forest.

Unfortunately, there is an entirely separate category of unique principal identifiers that emerge from the X.500 family of protocols including LDAP and X.509 Certificates. This is the "Distinguished Name" or DN. In the original X.500 standard, the DN contained geographic components with a world wide scope. Thus, yale.edu would be described as Organization "Yale University", Locality "New Haven", State "CT", Country "US". In practice, we generally assume that there is only one "Yale University" in the US and shorten the DN to

O=Yale University, C=US

This naming convention predates the wide use of Internet technologies and has been rendered obsolete by the now universally administered DNS namespace. An alternate standard for forming DN strings from DNS names produces

DC=yale, DC=edu

as the X.500-ish way of saying "yale.edu". The problem is that all this syntax is optional and there are no firm standards. So if Rutgers is trying to find something in a Yale LDAP directory, it has to know how many locality elements are included or omitted or whether the DNS alternate form is used. Not only is there no global standard, but each LDAP server at Yale uses an entirely different format.

The individual name part of a DN is the Common Name or CN. Unfortunately, the standard simply says that the Common Name is whatever name is commonly used. In different contexts for different directories, it could (and does) have forms like:

CN=gilbert
CN=Howard Gilbert
CN=Gilbert, Howard K.

So in a Windows Active Directory, which tries to combine legacy protocols and disparate current standards, the Principal identifier string for the same account can have several forms:

"gilbert" as a legacy host logon ID, which also happens to be the Netid
"gilbert@YU.YALE.EDU" as a Kerberos principal-realm where the realm cannot be dropped because the entire system is inherently cross-realm
"Howard.Gilbert@yale.edu" as the Universal Principal Name, a Forest-wide and, after R2, truely universal identifier
"CN=gilbert, CN=users, DC=yu, DC=yale, DC=edu" the AD form of DN, though the decision to use Netid as the common name is a Yale convention and would not have been the default if we had simply taken the default provided by the Windows administrative tools.
Unfortunately, you get a different form of Principal identifier with each different type of Credentials. Worse, although an X.509 certificate comes

with a DN, if the certificate comes from a Windows Certificate Server integrated with Active Directory it will have the format shown immediately above, while if the Certificate is vended by some other instutional CA it would probably have one of the "o=Yale University, c=US" formats.

This means that any authentication mechanism such as CAS that wants to accept various different forms of authentication has to accept that Principal is not one simply universally accepted string. There are a variety of string identifier formats used for different purposes, although the network should provide directory or database services that allow the institution to convert any of these formats to all of the others.

Long Term Persistant Cookie

The use of long term persistent Cookies for authentication is strongly discouraged. It is adequate for a level of access that requires attribution (who are you) but not authorization. For example, through cookies a Web based vendor may identify me and present recommendations based on buying patterns or a particular pricing level. However, when I actually order something the vendor requires me to logon with a real password.

In the University environment, some services may be restricted to on-campus users and behave differently for undergraduates, staff, or faculty. However, this may just be a convenience feature and not an access control mechanism. If an institution chooses to put this type of authentication behind CAS, then long term persistent Cookies are not an implausible badge of authority.

This somewhat complicates the CAS 3 architecture, because it implies that there are different reliability levels to different authentication method. Cookies are the weakest form, passwords are better, Smartcards are best. A CAS user may have logged on using a weak identification method, but may then be forced to relogon with a stronger method to use subsequent applications.

CAS uses cookies internally, but only in-memory cookies vended over SSL that are discarded when the browser session ends and timeout after a configured period. The kinds of cookies referred to here are persistent, stored on disk, and presented in subsequent browser sessions that may occur weeks after real authentication.

CAS should discourage this type of use. Fortunately, should it need to be supported the walkthrough is fairly simple. The Cookie variablename and value are extracted from the HTTP header at the Web interface layer, and they become the credential to the AuthHandler.

X.509 Certificate

A user can install a personal X.509 Certificate in a Web Browser. During SSL negotiation, the browser and server exchange certificate information. The Certificate is then available to CAS through the predefined ServletRequest attribute named "javax.net.ssl.peer_certificates". The attribute value is an array of javax.security.cert.X509Certificate objects. Presumably this array would be the primary credential passed across the interface from the Web layer to the AuthHandler.

Although the certificate may come from Verisign or some other universally accepted authority, in common use it may be a local certificate vended by a self-signed CA. For example, any organization running Windows Active Directory can create a self-signed domain CA and configure Group Policy to push a client certificate to each desktop user when he logs onto the domain. At that point the desktop user can authentication with Kerberos 5 to services that support that protocol, NTLM to services that support that protocol, and X.509 Certificates to other services (mostly SSL Web Servers).

The Principal identifier in the X.509 Certificate is in the form of a Distinguished Name. It may or may not contain the institutional Netid as a component. If not, some external lookup service must be provided (typically an LDAP server which natively organizes information in DN format) to return the Netid for a given DN value.

An X.509 Certificate can also be used by XML Signature to sign or encrypt part of a SOAP Web Services payload. Since this is not part of the HTTP protocol, is is not handled automatically by Servlet Container. However, Web Services support is typically structured as a processing pipeline in which each SOAP header is processed in turn by a class pushed onto the processing stack. Depending on how this is structured, the work might already be done by the WS stack. If not, then this is a standard API of the Apache XML Security jar file. So the Web interface part of CAS would capture the data, and the AuthHandler would use XML Security to verify the signature.

This sweeps under the rug a rather massive mound of crap involving different issues of Certificate validation. This problem is, in general terms, unsolvable. However, in practical deployment an institution can make some reasonable assumptions and develop a workable approach.

NTLM

Microsoft's Internet Explorer Browser has extensions that permit it to proxy the user's existing Windows Domain logon to the Web Server through a challenge-response protocol. The only 100% certain support for this protocol is when IE talks directly to a Microsoft IIS system. However, pieces of the protocol have either slipped out or been reversed engineered. Mozilla claims NTLM client support, and the SAMBA people provide a library (JCIFS) that supposedly understands the NTLM server side.

Getting this to actually work will be a problem. Fortunately, this paper only has to propose it and describe how it would work if it did work and make sure it fits in the CAS 3 architecture.

In the simplest case, NTLM is handled by the JCIFS Filter. CAS is passive and gets its information from the getRemoteUser() and similar methods of a wrapped ServletRequest.

Alternately, a 100% pure Java CAS may have to Redirect the request to a hybrid CAS running in a Microsoft IIS environment. It gets back a ticket, and obtains the Principal name when it validates the ticket.

Passport

Passport is only supported by IIS, so this is a Redirect handoff to IIS similar to the second NTLM case.

Kerberos

Microsoft has proposed an extension to the HTTP Browser authentication techniques that allows the use of real Kerberos tickets. Of course, any Windows XP or 2000 desktop computer logged into an Active Directory has use of Kerberos 5, and the protocol can be supported by IE. There appear to be versions of the protocol ported to older versions of Mozilla on Linux (the Negotiateauth plugin project) and Apache (mod_auth_kerb), but their real state of development and deployment is unclear.

There is no evidence of a server-side Java implementation available now. It may be possible to configure a working version of this running CAS under Tomcat under Apache, but the installation looks so complex and limited that it is an exercise for someone else who really wants it. It would appear, however, that this is a design where, from the CAS perspective, Authentication is performed by the Filter/Container and CAS would passively receive the results and require no significant code.

In Web Services, Kerberos has been added to later versions of the OASIS standards. The client obtains a Service Ticket and uses his key to encrypt or digitally sign part of the SOAP message. The Server verifies the signature with standard Kerberos libraries. The probable implementation would be in the Web Servers support layer and should be transparent to CAS. By the time the SOAP message arrives at the end of the pipeline and comes to CAS, the signature should have been validated and some part of the API should expose the Principal name. Details, however, depend on the implementation of WS support.

Shibboleth

CAS 3 would use Shibboleth to authenticate a user from another institution. The human interface would be determined by decisions at each CAS deployer. In one model, when a service running at Yale Redirects the Browser to Yale CAS, and the user types into the Userid field of the form a name of the form "somebody@rutgers.edu", then CAS could determine that this request be redirected to Shibboleth at Rutgers.

As it happens, Rutgers also uses CAS for its authentication. However, Yale CAS doesn't need to depend on this. It redirects the Browser to the URL configured (in the Shibboleth or SAML "Metadata") as the ID Provider for the Entity "rutgers.edu". That triggers cross-institutional exchange of authentication and attribute information in Shibboleth format.

When Yale CAS gets control back from this process, Rutgers has forwarded authentication and attribute information to the Yale Shibboleth Service Provider component. This information has a key (the "SessionID") which CAS can use to fetch it. CAS then internally generates a Ticket in which the Principal object is backed by information obtained from Shibboleth rather than information obtained from local authentication mechanisms and databases.

Shibboleth can operate in a sequence of steps that correlate one-for-one with CAS processing. Just as when one CAS server has to direct to a second CAS server, CAS hands the request off to Shibboleth and gets back something that looks a lot like a Ticket. Validating the Ticket produces authentication and attribute information. CAS now generates its own Ticket backed by the information it was just fed, and returns this secondary ticket to the original application.

In a slightly more direct mode, Shibboleth can act more like a CAS 3 AuthHandler. In this model, the Shibboleth ID Provider at Rutgers builds a set of Credentials in a Form and, using a small Javascript program, automatically submits the Form to Yale CAS. Instead of the Userid and Password Form that CAS normally processes, this Form contains the Bin64 encoded version of a SAML Authentication Response. So what. The CAS Web layer can extract it into a Credentials object, and it can be passed to a Shibboleth AuthHandler. It can then be passed through the API from a Resource Manager (the Shibboleth version of a CAS "service") to the Shibboleth classes. Shibboleth processes the Credentials and obtains attributes. This information can then be fed back to CAS through API queries.

Either programming model works, and the choice between them is mostly a matter of aesthetics.