A DETAILED DESCRIPTION OF THE CEPHX AUTHENTICATION PROTOCOL

Peter Reiher 7/13/12

This document provides deeper detail on the Cephx authorization protocol whose high level flow is described in the memo by Yehuda (12/19/09). Because this memo discusses details of routines called and variables used, it represents a snapshot. The code might be changed subsequent to the creation of this document, and the document is not likely to be updated in lockstep. With luck, code comments will indicate major changes in the way the protocol is implemented.

INTRODUCTION

The basic idea of the protocol is based on Kerberos. A client wishes to obtain something from a server. The server will only offer the requested service to authorized clients. Rather than requiring each server to deal with authentication and authorization issues, the system uses an authorization server. Thus, the client must first communicate with the authorization server to authenticate itself and to obtain credentials that will grant it access to the service it wants.

Authorization is not the same as authentication. Authentication provides evidence that some party is who it claims to be. Authorization provides evidence that a particular party is allowed to do something. Generally, secure authorization implies secure authentication (since without authentication, you may authorize something for an imposter), but the reverse is not necessarily true. One can authenticate without authorizing. The purpose of this protocol is to authorize.

The basic approach is to use symmetric cryptography throughout. Each client C has its own secret key, known only to itself and the authorization server A. Each server S has its own secret key, known only to itself and the authorization server A. Authorization information will be passed in tickets, encrypted with the secret key of the entity that offers the service. There will be a ticket that A gives to C, which permits C to ask A for other tickets. This ticket will be encrypted with A’s key, since A is the one who needs to check it. There will later be tickets that A issues that allow C to communicate with S to ask for service. These tickets will be encrypted with S’s key, since S needs to check them. Since we wish to provide security of the communications, as well, session keys are set up along with the tickets. Currently, those session keys are only used for authentication purposes during this protocol and the handshake between the client C and the server S, when the client provides its service ticket. They could be used for authentication or secrecy throughout, with some changes to the system.

Several parties need to prove something to each other if this protocol is to achieve its desired security effects.

1. The client C must prove to the authenticator A that it really is C. Since everything is being done via messages, the client must also prove that the message proving authenticity is fresh, and is not being replayed by an attacker.

2. The authenticator A must prove to client C that it really is the authenticator. Again, proof that replay is not occurring is also required.

3. A and C must securely share a session key to be used for distribution of later authorization material between them. Again, no replay is allowable, and the key must be known only to A and C.

4. A must receive evidence from C that allows A to look up C’s authorized operations with server S.

5. C must receive a ticket from A that will prove to S that C can perform its authorized operations. This ticket must be usable only by C.

6. C must receive from A a session key to protect the communications between C and S. The session key must be fresh and not the result of a replay.

GETTING STARTED WITH AUTHORIZATION

When the client first needs to get service, it contacts the monitor. At the moment, it has no tickets. Therefore, it uses the “unknown” protocol to talk to the monitor. This protocol is specified as `CEPH_AUTH_UNKNOWN`. The monitor also takes on the authentication server role, A. The remainder of the communications will use the cephx protocol (most of whose code will be found in files in `auth/cephx`). This protocol is responsible for creating and communicating the tickets spoken of above.

Currently, this document does not follow the pre-cephx protocol flow. It starts up at the point where the client has contacted the server and is ready to start the cephx protocol itself.

Once we are in the cephx protocol, we can get the tickets. First, C needs a ticket that allows secure communications with A. This ticket can then be used to obtain other tickets. This is phase I of the protocol, and consists of a send from C to A and a response from A to C. Then, C needs a ticket to allow it to talk to S to get services. This is phase II of the protocol, and consists of a send from C to A and a response from A to C.

PHASE I:

The client is set up to know that it needs certain things, using a variable called need, which is part of the `AuthClientHandler` class, which the `CephxClientHandler` inherits from. At this point, one thing that's encoded in the need variable is `CEPH_ENTITY_TYPE_AUTH`, indicating that we need to start the authentication protocol from scratch. Since we're always talking to the same authorization server, if we've gone through this step of the protocol before (and the resulting ticket/session hasn't timed out), we can skip this step and just ask for client tickets. But it must be done initially, and we'll assume that we are in that state.

The message C sends to A in phase I is build in `CephxClientHandler::build_request()` (in `auth/cephx/CephxClientHandler.cc`). This routine is used for more than one purpose. In this case, we first call `validate_tickets()` (from routine `CephXTicektManager::validate_tickets()` which lives in `auth/cephx/CephxProtocol.h`). This code runs through the list of possible tickets to determine what we need, setting values in the need flag as necessary. Then we call `ticket.get_handler()`. This routine (in `CephxProtocol.h`) finds a ticket of the specified type (a ticket to perform authorization) in the ticket map, creates a ticket handler object for it, and puts the handler into the right place in the map. Then we hit specialized code to deal with individual cases. The case here is when we still need to authenticate to A (the `if (need & CEPH_ENTITY_TYPE_AUTH)` branch).

We now create a message of type `CEPH_AUTH_UNKNOWN`. We need to authenticate this message with C's secret key, so we fetch that from the local key repository. (It's called a key server in the code, but it's not really a separate machine or processing entity. It's more like the place where locally used keys are kept.) We create a random challenge, whose purpose is to prevent replays. We encrypt that challenge. We already have a server challenge (a similar set of random bytes, but created by the server and sent to the client) from our pre-cephx stage. We take both challenges and our secret key and produce a combined encrypted challenge value, which goes into `req.key`.

If we have an old ticket, we store it in `req.old_ticket`. We're about to get a new one.

The entire `req` structure, including the old ticket and the cryptographic hash of the two challenges, gets put into the message. Then we return from this function, and the message is sent.

We now switch over to the authenticator side, A. The server receives the message that was sent, of type `CEPH_AUTH_UNKNOWN`. The message gets handled in `prep_auth()`, in `mon/AuthMonitor.cc`, which calls `handle_request()` is `CephxServiceHandler.cc` to do most of the work. This routine, also, handles multiple cases.

The control flow is determined by the `request_type` in the `cephx_header` associated with the message. Our case here is `CEPH_AUTH_UNKNOWN`. We need the secret key A shares with C, so we call `get_secret()` from out local key repository to get it. We should have set up a server challenge already with this client, so we make sure we really do have one. (This variable is specific to a `CephxServiceHandler`, so there is a different one for each such structure we create, presumably one per client A is dealing with.) If there is no challenge, we'll need to start over, since we need to check the client's crypto hash, which depends on a server challenge, in part.

We now call the same routine the client used to calculate the hash, based on the same values: the client challenge (which is in the incoming message), the server challenge (which we saved), and the client's key (which we just obtained). We check to see if the client sent the same thing we expected. If so, we know we're talking to the right client. We know the session is fresh, because it used the challenge we sent it to calculate its crypto hash. So we can give it an authentication ticket.

We fetch C's `eauth` structure. This contains an ID, a key, and a set of caps (capabilities).

The client sent us its old ticket in the message, if it had one. If so, we set a flag, `should_enc_ticket`, to true and set the global ID to the global ID in that old ticket. If the attempt to decode its old ticket fails (most probably because it didn't have one), `should_enc_ticket` remains false. Now we set up the new ticket, filling in timestamps, the name of C, the global ID provided in the method call (unless there was an old ticket), and his `auid`, obtained from the `eauth` structure obtained above. We need a new session key to help the client communicate securely with us, not using its permanent key. We set the service ID to `CEPH_ENTITY_TYPE_AUTH`, which will tell the client C what to do with the message we send it. We build a cephx response header and call `cephx_build_service_ticket_reply()`.

`cephx_build_service_ticket_reply()` is in `auth/cephx/CephxProtocol.cc`. This routine will build up the response message. Much of it copies data from its parameters to a message structure. Part of that information (the session key and the validity period) gets encrypted with C's permanent key. If the `should_encrypt_ticket` flag is set, encrypt it using the old ticket's key. Otherwise, there was no old ticket key, so the new ticket is not encrypted. (It is, of course, already encrypted with A's permanent key.) Presumably the point of this second encryption is to expose less material encrypted with permanent keys.

Then we call the key server's `get_service_caps()` routine on the entity name, with a flag `CEPH_ENTITY_TYPE_MON`, and capabilities, which will be filled in by this routine. The use of that constant flag means we're going to get the client's caps for A, not for some other data server. The ticket here is to access the authorizer A, not the service S. The result of this call is that the caps variable (a parameter to the routine we're in) is filled in with the monitor capabilities that will allow C to access A's authorization services.

handle_request() itself does not send the response message. It builds up the result_bl, which basically holds that message's contents, and the capabilities structure, but it doesn't send the message. We go back to prep_auth(), in mon/AuthMonitor.cc, for that. This routine does some fiddling around with the caps structure that just got filled in. There's a global ID that comes up as a result of this fiddling that is put into the reply message. The reply message is built here (mostly from the response_bl buffer) and sent off.

This completes Phase I of the protocol. At this point, C has authenticated itself to A, and A has generated a new session key and ticket allowing C to obtain server tickets from A.

PHASE II

This phase starts when C receives the message from A containing a new ticket and session key. The goal of this phase is to provide C with a session key and ticket allowing it to communicate with S.

The message A sent to C is dispatched to build_request() in CephxClientHandler.cc, the same routine that was used early in Phase I to build the first message in the protocol. This time, when validate_tickets() is called, the need variable will not contain CEPH_ENTITY_TYPE_AUTH, so a different branch through the bulk of the routine will be used. This is the branch indicated by if (need). We have a ticket for the authorizer, but we still need service tickets.

We must send another message to A to obtain the tickets (and session key) for the server S. We set the request_type of the message to CEPHX_GET_PRINCIPAL_SESSION_KEY and call ticket_handler.build_authorizer() to obtain an authorizer. This routine is in CephxProtocol.cc. We set the key for this authorizer to be the session key we just got from A,and create a new nonce. We put the global ID, the service ID, and the ticket into a message buffer that is part of the authorizer. Then we create a new CephXAuthorize structure. The nonce we just created goes there. We encrypt this CephXAuthorize structure with the current session key and stuff it into the authorizer's buffer. We return the authorizer.

Back in build_request(), we take the part of the authorizer that was just built (its buffer, not the session key or anything else) and shove it into the buffer we're creating for the message that will go to A. Then we delete the authorizer. We put the requirements for what we want in req.keys, and we put req into the buffer. Then we return, and the message gets sent.

The authorizer A receives this message which is of type CEPHX_GET_PRINCIPAL_SESSION_KEY. The message gets handled in prep_auth(), in mon/AuthMonitor.cc, which again calls handle_request() in CephxServiceHandler.cc to do most of the work.

In this case, handle_request() will take the CEPHX_GET_PRINCIPAL_SESSION_KEY case. It will call cephx_verify_authorizer() in CephxProtocol.cc. Here, we will grab a bunch of data out of the input buffer, including the global and service IDs and the ticket for A. The ticket contains a secret_id, indicating which key is being used for it. If the secret ID pulled out of the ticket was -1, the ticket does not specify which secret key A should use. In this case, A should use the key for the specific entity that C wants to contact, rather than a rotating key shared by all server entities of the same type. To get that key, A must consult the key repository to find the right key. Otherwise, there's already a structure obtained from the key repository to hold the necessary secret. Server secrets rotate on a time expiration basis (key rotation is not covered in this document), so run through that structure to find its current secret. Either way, A now knows the secret key used to create this ticket. Now decrypt the encrypted part of the ticket, using this key. It should be a ticket for A.

The ticket also contains a session key that C should have used to encrypt other parts of this message. Use that session key to decrypt the rest of the message.

Create a CephXAuthorizeReply to hold our reply. Extract the nonce (which was in the stuff we just decrypted), add 1 to it, and put the result in the reply. Encrypt the reply and put it in the buffer provided in the call to cephx_verify_authorizer() and return to handle_request(). This will be used to prove to C that A (rather than an attacker) created this response.

Having verified that the message is valid and from C, now we need to build it a ticket for S. We need to know what S it wants to communicate with and what services it wants. Pull the ticket request that describes those things out of its message. Now run through the ticket request to see what it wanted. (He could potentially be asking for multiple different services in the same request, but we will assume it's just one, for this discussion.) Once we know which service ID it's after, call build_session_auth_info().

build_session_auth_info() is in CephxKeyServer.cc. It checks to see if the secret for the service_ID of S is available and puts it into the subfield of one of the parameters, and calls the similarly named _build_session_auth_info(), located in the same file. This routine loads up the new auth_info structure with the ID of S, a ticket, and some timestamps for that ticket. It generates a new session key and puts it in the structure. It then calls get_caps() to fill in the info.ticket caps field. get_caps() is also in CephxKeyServer.cc. It fills the caps_info structure it is provided with caps for S allowed to C.

Once build_session_auth_info() returns, A has a list of the capabilities allowed to C for S. We put a validity period based on the current TTL for this context into the info structure, and put it into the info_vec structure we are preparing in response to the message.

Now call build_cephx_response_header(), also in CephxServiceHandler.cc. Fill in the request_type, which is

`CEPHX_GET_PRINCIPAL_SESSION_KEY`, a status of 0, and the result buffer.

Now call `cephx_build_service_ticket_reply()`, which is in `CephxProtocol.cc`. The same routine was used towards the end of A's handling of its response in phase I. Here, the session key (now a session key to talk to S, not A) and the validity period for that key will be encrypted with the existing session key shared between C and A. The `should_encrypt_ticket` parameter is false here, and no key is provided for that encryption. The ticket in question, destined for S once C sends it there, is already encrypted with S's secret. So, essentially, this routine will put ID information, the encrypted session key, and the ticket allowing C to talk to S into the buffer to be sent to C.

After this routine returns, we exit from `handle_request()`, going back to `prep_auth()` and ultimately to the underlying message send code.

The client receives this message. The nonce is checked as the message passes through `Pipe::connect()`, which is in `msg/SimpleMessager.cc`. In a lengthy `while(1)` loop in the middle of this routine, it gets an authorizer. If the get was successful, eventually it will call `verify_reply()`, which checks the nonce. `connect()` never explicitly checks to see if it got an authorizer, which would suggest that failure to provide an authorizer would allow an attacker to skip checking of the nonce. However, in many places, if there is no authorizer, important connection fields will get set to zero, which will ultimately cause the connection to fail to provide data. It would be worth testing, but it looks like failure to provide an authorizer, which contains the nonce, would not be helpful to an attacker.

The message eventually makes its way through to `handle_response()`, in `CephxClientHandler.cc`. In this routine, we call `get_handler()` to get a ticket handler to hold the ticket we have just received. This routine is embedded in the definition for a `CephXTicketManager` structure. It takes a type (`CEPH_ENTITY_TYPE_AUTH`, in this case) and looks through the `tickets_map` to find that type. There should be one, and it should have the session key of the session between C and A in its entry. This key will be used to decrypt the information provided by A, particularly the new session key allowing C to talk to S.

We then call `verify_service_ticket_reply()`, in `CephxProtocol.cc`. This routine needs to determine if the ticket is OK and also obtain the session key associated with this ticket. It decrypts the encrypted portion of the message buffer, using the session key shared with A. This ticket was not encrypted (well, not twice - tickets are always encrypted, but sometimes double encrypted, which this one isn't). So it can be stored in a service ticket buffer directly. We now grab the ticket out of that buffer.

The stuff we decrypted with the session key shared between C and A included the new session key. That's our current session key for this ticket, so set it. Check validity and set the expiration times. Now return true, if we got this far.

Back in `handle_response()`, we now call `validate_tickets()` to adjust what we think we need, since we now have a ticket we didn't have before. If we've taken care of everything we need, we'll return 0.

This ends phase II of the protocol. We have now successfully set up a ticket and session key for client C to talk to server S. S will know that C is who it claims to be, since A will verify it. C will know it is S it's talking to, again because A verified it. The only copies of the session key for C and S to communicate were sent encrypted under the permanent keys of C and S, respectively, so no other party (excepting A, who is trusted by all) knows that session key. The ticket will securely indicate to S what C is allowed to do, attested to by A. The nonces passed back and forth between A and C ensure that they have not been subject to a replay attack. C has not yet actually talked to S, but it is ready to.

Much of the security here falls apart if one of the permanent keys is compromised. Compromise of C's key means that the attacker can pose as C and obtain all of C's privileges, and can eavesdrop on C's legitimate conversations. He can also pretend to be A, but only in conversations with C. Since it does not (by hypothesis) have keys for any services, he cannot generate any new tickets for services, though it can replay old tickets and session keys until S's permanent key is changed or the old tickets time out.

Compromise of S's key means that the attacker can pose as S to anyone, and can eavesdrop on any user's conversation with S. Unless some client's key is also compromised, the attacker cannot generate new fake client tickets for S, since doing so requires it to authenticate himself as A, using the client key it doesn't know.