

ZXID Reference

Generated from comments in code.

February 25, 2010

See also

- `mod_auth_saml` Apache module documentation: SSO without programming.
- `zxid_simple()` Easy API for SAML
- ZXID Raw API: Program like the pros (and fix your own problems). See also Function Reference
- ZXID ID-WSF API: Make Identity Web Services Calls using ID-WSF
- ZXID Compilation and Installation: Compile and install from source or package.
- ZXID Configuration Reference: Nitty gritty on all options.
- ZXID Circle of Trust Reference: How to set up the Circle of Trust, i.e. the partners your web site works with.
- ZXID Logging Reference: ZXID digitally signed logging facility
- `javazxid`: Using ZXID from Java
- `Net::SAML`: Using ZXID from Perl
- `php_zxid`: Using ZXID from PHP
- `README.smime`: Crypto and Cert Tutorial

Contents

1 Reference

1.1 Structures

1.2 Important Functions

1.2.1 `chkuid()`

Apache hook. Called from `httpd-2.2.8/server/request.c`: `ap_process_request_internal()` `ap_run_check_user_id()`. Return value is processed in `modules/http/http_request.c` and redirect is in `ap_die()`, `http_protocol.c`: `ap_send_error_response()`

Source file: `mod_auth_saml.c`

1.2.2 `zx_strf()`

Construct `zx_str` given `sprintf(3)` format and grabbing memory from ZX memory allocator.

Source file: `zxlib.c`

1.2.3 `zxid_add_qs_to_ses()`

Add Attributes from Query String to Session attribute pool The `qs` argument is parsed according to the CGI Query String rules and the attributes are added to the session. If `apply_map` is 1, the INMAP configuration is applied. While this may seem a hassle, it allows for specification of the values as `safe_base64`, etc. If values are to be added verbatim, just specify 0 (all other values reserved). The input argument `qs` gets modified in-situ due to URL decoding and nul termination. Make sure to duplicate any string constant before calling. Returns 1 on success, 0 on failure (return value often not checked).

Source file: `zxidpool.c`

1.2.4 `zxid_az_cf()`

Call Policy Decision Point (PDP) to obtain an authorization decision about a contemplated action on a resource. The attributes from the session pool, as filtered by `PEPMAP` are fed to the PDP as inputs for the decision.

cf the configuration will need to have `PEPMAP` and `PDP_URL` options set according to your situation.

qs if non-null, will receive error and status codes

sid all attributes are obtained from the session. You may wish to add additional attributes that are not known by SSO. The session id, such as returned from SSO.

returns 0 on deny (for any reason, e.g. indeterminate), or string containing the obligations on permit.

For simpler API, see `zxid_az()` family of functions.

Source file: `zxidpep.c`

1.2.5 `zxid_az_cf_ses()`

Call Policy Decision Point (PDP) to obtain an authorization decision about a contemplated action on a resource. The attributes from the session pool, as filtered by `PEPMAP` are fed to the PDP as inputs for the decision.

- cf** the configuration will need to have `PEPMAP` and `PDP_URL` options set according to your situation.
- qs** if non-null, will receive error and status codes
- ses** all attributes are obtained from the session. You may wish to add additional attributes that are not known by SSO. The session object, e.g. from `zxid_get_ses()`
- returns** 0 on deny (for any reason, e.g. indeterminate), or string containing the obligations on permit.

For simpler API, see `zxid_az()` family of functions.

Source file: `zxidpep.c`

1.2.6 `zxid_call()`

Make a SOAP call given XML payload for SOAP `<e:Envelope>` or `<e:Body>` content, specified by the string. This is your WSC work horse for calling almost any kind of web service. Simple and intuitive specification of XML as string: no need to build complex data structures.

If the string starts by "`<e:Envelope`", then string should be a complete SOAP envelope including `<e:Header>` and `<e:Body>` parts. This allows caller to specify custom SOAP headers, in addition to the ones that the underlying `zxid_wsc_call()` will add. Usually the payload service will be passed as the contents of the body. If the string starts by "`<e:Body`", then the `<e:Envelope>` and `<e:Header>` are automatically added. If the string starts by neither of the above (be careful to use the "e:" as namespace prefix), then it is assumed to be the payload content of the `<e:Body>` and the rest of the SOAP envelope is added.

- cf** ZXID configuration object, see `zxid_new_conf()`
- ses** Session object that contains the EPR cache
- svctype** URI (often the namespace URI) specifying the kind of service we wish to call. Used for EPR lookup or discovery.
- url** (Optional) If provided, this argument has to match either the ProviderID, EntityID, or actual service endpoint URL.
- di_opt** (Optional) Additional discovery options for selecting the service, query string format
- az_cred** (Optional) Additional authorization credentials or attributes, query string format. These credentials will be populated to the attribute pool in addition to the ones obtained from SSO and other sources. Then a PDP is called to get an authorization decision (as well as obligations we pledge to support). See also `PEPMAP` configuration option. This implements generalized (application independent) Requestor Out and Requestor In PEPs. To implement application dependent PEP features you should call `zxid_az()` directly.
- env** XML payload

return SOAP Envelope of the response, as a string. You can parse this string to obtain all returned SOAP headers as well as the Body and its content.

Source file: `zxidwsc.c`

1.2.7 `zxid_check_fed()`

Check federation, create federation if appropriate.

Source file: `zxidpssso.c`

1.2.8 `zxid_chk_sig()`

Check single item signature on given Request, Response, or Assertion. Typical usage

```
if (!zxid_chk_sig(cf, cgi, ses, (struct zx_elem_s*)req,
                req->Signature, req->Issuer, "LogoutRequest"))
    return 0;
```

cf ZXID configuration and context object, used for settings and memory allocation

cgi cgi or invocation variables object. `cgi->sigval` and `cgi->sigmsg` will be altered, if there is any signature.

ses Session object. The `ses->sigres` will be altered to reflect result of verification, if there is signature.

elem Element that was signed, usually needs type cast.

sig Signature element within elem

issue_ent The EntityID `zx_str` of the signer (Issuer)

return 0 if sig check could not be made due to error, 1 if there was no signature to check, 2 if check was made, in which case the result is in `ses->sigres`, 3 if check was not possible (due to error), but sig was not configured to be required (NOSIG_FATAL option).

See also: Signature validation codes VVV in `zxid-log.pd`, section "ZXID Log Format". N.B: If the signature is over multiple references, you need to do many processing steps manually and then call `zxsig_validate()` with correctly populate refs array.

Source file: `zxidlib.c`

1.2.9 `zxid_decode_redir_or_post()`

Decode redirect or POST binding message. `zxid_saml2_redir_enc()` performs the opposite operation.

Source file: `zxiddec.c`

1.2.10 *zxid_fed_mgmt_cf()*

Generate Single Logout button and possibly other federation management buttons for use in logged in state of the app HTML GUI.

Either outputs the management screen to stdout or returns string of HTML (at specified automation level). If *res_len* is supplied, the string length is returned in *res_len*. Otherwise you can just run *strlen()* on return value.

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidsimp.c*

1.2.11 *zxid_fetch_ses()*

Allocate memory and get session object from the filesystem, populating attributes to pool so they are available for use. You must obtain session id from some source.

Source file: *zxidses.c*

1.2.12 *zxid_get_at()*

Given session object (see *zxid_simple_cf_ses()* or *zxid_fetch_ses()*), return *n*'th value (*ix=0* is first) of given attribute, if any, from the session common attribute pool. If *apply_map* is 0, the value is returned as is. If it is 1 then OUTMAP is applied (the attribute name is in the internal namespace). Other *apply_map* values are reserved.

Source file: *zxidpool.c*

1.2.13 *zxid_get_ent_ss()*

Get metadata for entity, either from cache or network (using WKL), depending on configuration options.

cf ZXID configuration object

eid Entity ID whose metadata is desired

return Entity data structure, including the metadata

Source file: *zxidmeta.c*

1.2.14 *zxid_get_epr()*

First search epr cache, and if miss, go discover an EPR over the net. This is the main work horse for WSCs wishing to call WSPs via EPR.

cf ZXID configuration object, also used for memory allocation

ses Session object in whose EPR cache the file will be searched

svc Service type (usually a URN)

url (Optional) If provided, this argument has to match either the ProviderID, EntityID, or actual service endpoint URL.

di_opt (Optional) Additional discovery options for selecting the service, query string format

action (Optional) The action, or method, that must be invocable on the service

n How many matching instances are returned. 1 means first

return EPR data structure on success, 0 on failure (no discovery EPR in cache, or not found by the discovery service). If more than one were found, a linked list of EPRs is returned.

Source file: `zxidepr.c`

1.2.15 `zxid_idp_list_cf_cgi()`

Generate IdP selection buttons (Login buttons) for the IdPs that are members of our Circle of Trust (CoT). This can be used as component for developing your application specific (HTML) login screen.

N.B. More complete documentation is available in `zxid-simple.pdf`

Source file: `zxidsimp.c`

1.2.16 `zxid_idp_select_zxstr_cf_cgi()`

Render entire IdP selection screen. You may use this code, possibly adjusted by some configuration options (see `zxidconf.h`), or you may choose to develop your own IdP selection screen from scratch.

N.B. More complete documentation is available in `zxid-simple.pdf`

Source file: `zxidsimp.c`

1.2.17 `zxid_idp_sso()`

Generate SSO assertion and ship it to SP by chosen binding.

Source file: `zxidpssoc.c`

1.2.18 `zxid_init_conf()`

Initialize configuration object, which must have already been allocated, to factory defaults (i.e. compiled in defaults, see `zxidconf.h`).

cf Pointer to previously allocated configuration object

path Since this configuration option is so fundamental, it can be supplied directly as argument.

return 0 on success (currently, 2008, this function can not fail - thus it is common to ignore the return value)

N.B. This function does NOT initialize the ZX context object although it is a field of this object. You MUST separately initialize the ZX context object, e.g. using `zx_reset_ctx()` or `zx_init_ctx()`, before you can use ZXID configuration object in any memory allocation prone activity (which is nearly every function in this API).

Source file: `zxidconf.c`

1.2.19 `zxid_mk_user_a7n_to_sp()`

Construct an assertion given user's attribute and bootstrap configuration. `bs_lvl::` 0: DI (do not add any bs), 1: add all bootstraps at sso level,

```
<= cf->bootstrap_level: add all bootstraps, > cf->bootstrap_level: only add di BS.
```

Source file: `zxidpssso.c`

1.2.20 `zxid_my_entity_id()`

Primary interface to our own Entity ID. While this would usually be automatically generated from URL configuration option so as to conform to the Well Known Location (WKL) metadata exchange convention [?], on some sites the entity ID may be different and thus everybody who does not know better should use this interface to obtain it.

cf ZXID configuration object, used to compute EntityID and also for memory allocation

return Entity ID as `zx_str`

Source file: `zxidmeta.c`

1.2.21 `zxid_parse_cgi()`

Parse query string or form POST and detect parameters relevant for ZXID. N.B. This CGI parsing is very specific for needs of ZXID. It is not generic.

cgi Already allocated CGI structure where results of this function are deposited. Note that this structure is not cleared. Thus it is possible to call `zxid_parse_cgi()` multiple times to accumulate results from multiple sources, e.g. foirst for query string, and then for form POST.

qs CGI formatted input. Usually query string or form POST content.

return 0 on success. Other values reserved. Usually return value is ignored as there really is no way for this function to fail. Unrecognized CGI arguments are simply ignored with assumption that some other processing layer will pick them up - hence no need to flag error.

Source file: `zxidcgi.c`

1.2.22 `zxid_parse_conf_raw()`

Parse partial configuration specifications, such as may occur on command line or in a configuration file.

Generally you should call first `zxid_new_conf()`, or at least `zxid_init_conf()`, and then call this function to apply modifications over the defaults. The configuration options are named after the config options that appear in `zxidconf.h`, except that prefix `ZXID_` is removed.

N.B. The `qs` memory must come from static or permanently allocated source as direct pointers to inside it will be taken. The memory will be modified to add nul terminations. Do not use stack based memory like local variable (unless local of `main()`). Do consider `strdup()` or similar before calling this function.

cf Previously allocated and initialized ZXID configuration object

qs_len Query String length. -1 means nul terminated C string

qs Configuration data in extended CGI Query String format. "extended" means new-line can be used as separator, in addition to ampersand ("&") This argument is modified in place, changing separators to nul string terminations.

return -1 on failure, 0 on success

Source file: `zxidconf.c`

1.2.23 `zxid_peg_az_soap()`

Call Policy Decision Point (PDP) to obtain an authorization decision about a contemplated action on a resource. The attributes from the session pool, as filtered by PEPMAP are fed to the PDP as inputs for the decision. The call is using XACML SAML profile over SOAP.

cf the configuration will need to have `PEPMAP` and `PDP_URL` options set according to your situation.

cgi if non-null, will receive error and status codes

ses all attributes are obtained from the session. You may wish to add additional attributes that are not known by SSO.

returns 0 on deny (for any reason, e.g. indeterminate), or string containing the obligations on permit.

For simpler API, see `zxid_az()` family of functions.

Source file: `zxidpeg.c`

1.2.24 *zxid_pool_to_ldif()*

Convert attributes from (session) pool to LDIF entry, applying OUTMAP. This is used by *zxid_simple()* SSO successful code to generate return value, but can also be used later to regenerate the LDIF given the pool. See *zxid_ses_to_pool()* for how to create the pool.

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidpool.c*

1.2.25 *zxid_saml2_post_enc()*

Encode (and sign if Simple Sign) a form according to SAML2 POST binding. *zxid_decode_redir_or_post()* performs the opposite operation.

cf ZXID configuration object, also used for memory allocation

field The name of the CGI variable, e.g. "SAMLRequest" or "SAMLResponse"

payload What should be encoded in the redirect URL. Effectively becomes the query string

relay_state Optional relay state argument. Ends up being encoded in the query string

sign Whether binding layer signature is to be applied: 0=no, 1=POST-Simple-Sign

url URL where the form should be posted

return Query string encoding of the request. The memory should be freed by the caller. 0 on failure.

Source file: *zxidlib.c*

1.2.26 *zxid_saml2_redir_enc()*

Encode and sign a URL according to SAML2 redirect binding. *zxid_decode_redir_or_post()* performs the opposite operation.

1. Compress payload
2. Base64 encode payload
3. URL encode and concatenate RelayState (if any)
4. Sign the URL encoded form (SimpleSign signs message prior to base64 and URL encodings)
5. Base64 encode the sig and concatenate to the URL

cf ZXID configuration object, also used for memory allocation

- field** The name of the CGI variable, e.g. "SAMLRequest=" or "SAMLResponse="
- payload** What should be encoded in the redirect URL. Effectively becomes the query string
- relay_state** Optional relay state argument. Ends up being encoded in the query string
- return** Query string encoding of the request. The memory should be freed by the caller.

Source file: `zxidlib.c`

1.2.27 `zxid_ses_to_pool()`

Process attributes from the AttributeStatements of the session SSO Assertion and insert them to the pool. NEED, WANT, and INMAP are applied. The pool is suitable for use by PEP or eventually rendering to LDIF (or JSON).

Source file: `zxidpool.c`

1.2.28 `zxid_simple_ab_pep()`

Postprocessing of SSO: Attribute Broker handles attributes and PEP/PDP decide on authorization.

Source file: `zxidsimp.c`

1.2.29 `zxid_simple_cf_ses()`

Simple handler that assumes the configuration has already been read in. The memory for result is grabbed from `ZX_ALLOC()`, usually `malloc(3)` and is "given" away to the caller, i.e. caller must free it. The return value is LDIF (or JSON or query string, if configured) of attributes in success case. `res_len`, if non-null, will receive the length of the response.

The major advantage of `zxid_simple_cf_ses()` is that the session stays as binary object and does not need to be recreated / reparsed from filesystem representation. The object can be directly used for PEP calls (but see inline PEP call enabled by PDPURL) and WSC.

cf Configuration object

qs_len Length of the query string. -1 = use `strlen()`

qs Query string (or POST content)

ses Session object

res_len Result parameter. If non-null, will be set to the length of the returned string

auto_flags Automation flags, see `zxid-simple.pd` for documentation

return String representing protocol action or SSO attributes

N.B. More complete documentation is available in `zxid-simple.pd`

Source file: `zxidsimp.c`

1.2.30 `zxid_soap_call_raw()`

Send SOAP request and wait for response. Send the message to the server using Curl. Return the parsed XML response data structure. This call will block while the HTTP request-response is happening.

cf ZXID configuration object, also used for memory allocation

url Where the request will be sent

data Serialized XML data to be sent

return XML data structure representing the response, or 0 upon failure

The underlying HTTP client is libcurl. While libcurl is documented to be "entirely thread safe", one limitation is that curl handle can not be shared between threads. Since we keep the curl handle a part of the configuration object, which may be shared between threads, we need to take a lock for duration of the curl operation. Thus any given configuration object can have only one HTTP request active at a time. If you need more parallelism, you need more configuration objects.

Source file: `zxidcurl.c`

1.2.31 `zxid_sp_deref_art()`

Dereference an artifact to obtain an assertion. This is the last step in artifact SSO profile and involved making a SOAP call to the IdP. The artifact is received in `saml_art` CGI field, `??` where SAMLart query string argument is parsed.

Source file: `zxidssso.c`

1.2.32 `zxid_sp_slo_redir()`

Send Single Logout to IdP using redirect binding. This function generates the URL encapsulating the request. You need to pass this URL to the appropriate function in your environment to provoke an HTTP 302 redirect.

cf ZXID config object, also used for memory allocation

cgi Data parsed from POST or query string. Provides parameters to determine details of the SLO request

ses Session object. Used to determine session index (`ses_ix`) and name id, among others

return location string if successful. `* ERR` upon failure.

Source file: `zxidslo.c`

1.2.33 *zxid_sp_slo_soap()*

SOAP client for sending Single Logout to IdP. The SOAP call is made using CURL HTTP Client and will block until response is received.

return 1 if successful. 0 upon failure.

Source file: `zxidslo.c`

1.2.34 *zxid_sp_sso_finalize()*

zxid_sp_sso_finalize() gets called irrespective of binding (POST, Artifact) and validates the SSO a7n, including the authentication statement. Then, it creates session and optionally user entry.

cf Configuration object, used to determine time slops, potentially memalloc via `cf->ctx`

cgi CGI object. `sigval` and `sigmsg` may be set.

ses Session object. Will be modified according to new session created from the SSO assertion.

a7n Single Sign-On assertion

return 0 for failure, otherwise some success code such as `ZXID_SSO_OK`

Source file: `zxidsso.c`

1.2.35 *zxid_start_sso_url()*

Generate an authentication request and make a URL out of it. `cf`:: Used for many configuration options and memory allocation `cgi`:: Used to pick the desired SSO profile based on hidden fields or user input. `return`:: Redirect URL as `zx_str`. Caller should eventually free this memory.

Source file: `zxidsso.c`

1.2.36 *zxid_validate_cond()*

Validates conditions required by Liberty Alliance SAML2 conformance testing.

May eventually validate additional conditions as well (this is the right place to add them). N.B. It is not an error if a condition is missing, or there is no Conditions element at all.

cf Configuration object, used to determine time slops. Potentially used for memory allocation via `cf->ctx`.

cgi Optional CGI object. If non-NULL, `sigval` and `sigmsg` will be set.

- ses** Optional session object. If non-NULL, then sigres code will be set.
- a7n** Assertion whose conditions are checked.
- myentid** Entity ID used for checking audience restriction. Typically from *zxid_my_entity_id(cf)*
- ourts** Timestamp for validating NotOnOrAfter and NotBefore.
- err** Result argument: Error letter (as may appear in audit log entry). The returned string will be a constant and MUST NOT be freed by the caller.
- return** 0 (ZXSIG_OK) if validation was successful, otherwise a ZXSIG error code.

Source file: `zxidssso.c`

1.2.37 *zxid_wsc_call()*

zxid_wsc_call() implements the main low level ID-WSF web service call logic, including preparation of SOAP headers, use of sec mech (e.g. preparation of wsse:Security header and signing of appropriate components of the message), and sequencing of the call. In particular, it is possible that WSP requests user interaction and thus the caller web application will need to perform a redirect and then later call this function again to continue the web service call after interaction.

env (rather than *Body*) is taken as argument so that caller can prepare additional SOAP headers at will before calling this function. This function will add Liberty ID-WSF specific SOAP headers.

Source file: `zxidwsc.c`

1.2.38 *zxid_wsf_decor()*

zxid_wsf_decor() implements the main low level ID-WSF web service call logic, including preparation of SOAP headers, use of sec mech (e.g. preparation of wsse:Security header and signing of appropriate components of the message), and sequencing of the call. In particular, it is possible that WSP requests user interaction and thus the caller web application will need to perform a redirect and then later call this function again to continue the web service call after interaction.

env (rather than *Body*) is taken as argument so that caller can prepare additional SOAP headers at will before calling this function. This function will add Liberty ID-WSF specific SOAP headers.

Source file: `zxidwsp.c`

1.2.39 *zxid_wsp_decorate()*

Add ID-WSF (and TAS3) specific headers and signatures to web service response. Simple and intuitive specification of XML as string: no need to build complex data structures.

If the string starts by "<e:Envelope", then string should be a complete SOAP envelope including <e:Header> and <e:Body> parts. This allows caller to specify custom SOAP headers, in addition to the ones that the underlying `zxid_wsc_call()` will add. Usually the payload service will be passed as the contents of the body. If the string starts by "<e:Body", then the <e:Envelope> and <e:Header> are automatically added. If the string starts by neither of the above (be careful to use the "e:" as namespace prefix), then it is assumed to be the payload content of the <e:Body> and the rest of the SOAP envelope is added.

cf ZXID configuration object, see `zxid_new_conf()`

ses Session object that contains the EPR cache

az_cred (Optional) Additional authorization credentials or attributes, query string format. These credentials will be populated to the attribute pool in addition to the ones obtained from token and other sources. Then a PDP is called to get an authorization decision (generating obligations). See also PEPMAP configuration option. This implements generalized (application independent) Responder Out PEP. To implement application dependent PEP features you should call `zxid_az()` directly.

env XML payload

return SOAP Envelope of the response, as a string, ready to be sent as HTTP response.

Source file: `zxidwsp.c`

1.2.40 `zxid_wsp_validate()`

Validate SOAP request (envelope), specified by the string.

If the string starts by "<e:Envelope", then string should be a complete SOAP envelope including <e:Header> (and <e:Body>) parts.

cf ZXID configuration object, see `zxid_new_conf()`

ses Session object that contains the EPR cache

az_cred (Optional) Additional authorization credentials or attributes, query string format. These credentials will be populated to the attribute pool in addition to the ones obtained from token and other sources. Then a PDP is called to get an authorization decision (matching obligations we support to those in the request, and obligations pledged by caller to those we insist on). See also PEPMAP configuration option. This implements generalized (application independent) Responder In PEP. To implement application dependent PEP features you should call `zxid_az()` directly.

env Entire SOAP envelope as a string

return idpnid of target identity of the request (rest of the information is populated to the session object, from where it can be retrieved). NULL if the validation fails. The target identity is still retrievable from the session, should there be desire to process the message despite the validation failure.

Source file: `zxidwsp.c`

1.2.41 *zxlog_path()*

Log to activity and/or error log depending on `res` and configuration settings. This is the main audit logging function you should call. Please see `zxid-log.pd` for detailed description of the log format and features. See `zxid-conf.pd` for configuration options governing the logging.

Proper audit trail is essential for any high value transactions based on SSO. Also some SAML protocol Processing Rules, such as duplicate detection, depend on the logging.

cf (1) ZXID configuration object, used for configuration options and memory allocation

ourts (2) Timestamp as observed by localhost. Typically the wall clock time. See *gettimeofday(3)*

srcts (3) Timestamp claimed by the message to which the log entry pertains

ippport (4) IP address and port number from which the message appears to have originated

entid (5) Entity ID to which the message pertains, usually the issuer. Null ok.

msgid (6) Message ID, can be used for correlation to establish audit trail continuity from request to response. Null ok.

a7nid (7) Assertion ID, if message contained assertion (outermost and first assertion if there are multiple relevant assertions). Null ok.

nid (8) Name ID pertaining to the message

sigval (9) Signature validation letters

res (10) Result letters

op (11) Operation code for the message

arg (12) Operation specific argument

fmt, ... Free format message conveying additional information

return 0 on success, nonzero on failure (often ignored as *zxlog()* is very robust and rarely fails - and when it does situation is so hopeless that you would not be able to report its failure anyway)

Source file: `zxlog.c`

1.2.42 *zxsig_sign()*

Sign, using XML-DSIG, some XML data in the `sref` array. The XML data is canonicalized and the signature is generated and returned. Typically the caller will then insert the signature to the original data structure and canonicalize for transport.

c ZX context. Used for memory allocation.

n Number of elements in the `sref` array

sref An array of (reference, xml data structure) tuples that are to be signed

cert Certificate (public key) used for signing

priv_key Private key used for signing

return Signature as XML data, or 0 if failure.

Steps

1. Canon *tag(s)* to sign (done by caller), pass as sig refs
2. Sha1 each sig ref
3. Construct the Signature element
4. Attach signature to the element (done by caller)

Typical XML-DSIG Signature

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
    <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
    <ds:Reference URI="#CREdM7unLxp2sOXQYfDR8E4F">
      <ds:Transforms>
        <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        <ec:InclusiveNamespaces
          xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"
          PrefixList="xasa" /></ec>
        <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <ds:DigestValue>I2wmlQullnvfSepvzor29kAZwAo=</ds:DigestValue>
      </ds:Reference>
    <ds:SignatureValue>
      FK6X9qO8qZntp3CeFbA7gpG9n9rWyJWlzsXy0vKNspwMGdl8HPfOGcXEs2Ts=</ds:SignatureValue>
  </ds:SignedInfo>
</ds:Signature>
```

Source file: `zxsig.c`

1.2.43 `zxsig_validate()`

Validate XML-DSIG signature over XML data found in `sref` array. Signature is validated against provided certificate, which must have been previously looked up, usually using Issuer field of message and metadata of the signing party. Trust in the certificate must have been established by other means.

c ZX context. Used for memory allocation.

- cert** Signing party's certificate (public key), typically from metadata
- sig** Parsed XML-DSIG data structure
- n** Number of elements in the sref array
- sref** An array of (reference, xml data structure) tuples that are referenced by the signature
- return** ZXSIG value. 0 (ZXSIG_OK) means success. Any other value is some soft of failure

Source file: `zxsig.c`

1.3 Other Functions

1.3.1 *close_file()*

Close a file rather than just any file descriptor and check error return. It is important that it is a file since on MS Windows closing files is different from closing descriptors. Checking error return from close is important because in NFS environments you may not know that your write has failed until you actually attempt to close the file.

Source file: `zxutil.c`

1.3.2 *dirconf()*

Create default configuration in response for Apache `<Location>` or `<Directory>` directives. This is then augmented by ZXIDConf directives. This code may run twice: once for syntax check, and then again for production use. Curently we just redo the work.

This is considered internal function to `mod_auth_saml`. Do not call directly.

Source file: `mod_auth_saml.c`

1.3.3 *hexdump()*

Outout a hexdump to `stderr`. Used for debugging purposes.

Source file: `zxutil.c`

1.3.4 *name_from_path()*

Generate formatted file name path.

Source file: `zxutil.c`

1.3.5 *open_fd_from_path()*

Open a file with formatted file name path.

Source file: `zxutil.c`

1.3.6 *pool2apache()*

Convert session attribute pool into Apache execution environment that will be passed to CGI, `mod_php`, `mod_perl`, and other Apache modules.

OUTMAP will be applied to decide which attributes to pass to the environment and to rename them.

This is considered internal function to `mod_auth_saml`, called by `chkuid()`. You should not call this directly, unless you know what you are doing.

Source file: `mod_auth_saml.c`

1.3.7 *read_all()*

Read all data from a file at formatted file name path.

maxlen Length of buffer

buf Result parameter. This buffer will be populated with data from the file.

return actual total length. The buffer will always be nul terminated.

Source file: `zxutil.c`

1.3.8 *read_all_fd()*

Low level function that keeps on sucking from a file descriptor until want is satisfied or error happens. May block (though usually will not if the file is in cache or local disk) in process. Buffer `p` must have been allocated. Return value reflects last got, i.e. what last `read(2)` system call returned. `got_all` reflects the total number of bytes received.

Source file: `zxutil.c`

1.3.9 *read_post()*

Read POST input, Apache style

This is considered internal function to `mod_auth_saml`, called by `chkuid()`. You should not call this directly, unless you know what you are doing.

Source file: `mod_auth_saml.c`

1.3.10 *reghk()*

Register Apache hook for mod_auth_saml

This is considered internal function to mod_auth_saml. Do not call directly.

Source file: mod_auth_saml.c

1.3.11 *send_res()*

Send Apache response.

This is considered internal function to mod_auth_saml, called by *chkuid()*. You should not call this directly, unless you know what you are doing.

Source file: mod_auth_saml.c

1.3.12 *set_debug()*

Process ZXIDDebug directive in Apache configuration file.

This is considered internal function to mod_auth_saml. Do not call directly.

Source file: mod_auth_saml.c

1.3.13 *set_zxid_conf()*

Process ZXIDConf directive in Apache configuration file. Can be called any number of times to set additional parameters.

This is considered internal function to mod_auth_saml. Do not call directly.

Source file: mod_auth_saml.c

1.3.14 *sha1_safe_base64()*

The out_buf should be 28 chars in length. The buffer is not automatically nul terminated. There will be 27 characters of payload, plus one termination character "." (which you can overwrite with nul if you like).

out_buf Buffer where result will be written. It must be 28 characters long and already allocated.

len Length of data

data Data to be digested

return Pointer one past last character written

Source file: zxutil.c

1.3.15 *unbase64_raw()*

Raw version. Can use any encoding table and arbitrary line length. Known bug: `line_len` is not fully respected on last line - it can be up to 3 characters longer than specified due to padding. Every three chars (from alphabet of 256) of input map to four chars (from alphabet of 64) of output. See also *SIMPLE_BASE64_LEN()*.

p input

len length of input

r Output buffer. Will not be NUL terminated.

basis_64 The 64 character alphabet to be used, such as `std_basis_64` or `safe_basis_64`

line_len Length of each line. 76 is customary. Or use very large value to avoid any line breaks

eol_len Length of End-of-Line string.

eol End-of-Line string, inserted every `line_len`.

eq_pad Padding character, usually equals (=)

return Pointer one past last byte written in `r`. This function never fails.

Source file: `zxutil.c`

1.3.16 *vname_from_path()*

Generate formatted file name path.

Source file: `zxutil.c`

1.3.17 *vopen_fd_from_path()*

Open a file with formatted file name path.

Source file: `zxutil.c`

1.3.18 *write2_or_append_lock_c_path()*

Write or append all data to a file at the formatted path. Will perform file locking to ensure consistent results. Returns 1 on success, 0 on err

Source file: `zxutil.c`

1.3.19 *write_all_fd()*

Low level function that keeps writing data to a file descriptor until everything is written. It may block in the process.

Source file: `zxutil.c`

1.3.20 *write_all_path_fmt()*

Write all data to a file at the formatted path. The *buf* is used for formatting data. The *path_fmt* can have up to two *%s* specifiers, which will be satisfied by *prepath* and *postpath*. Return 1 on success, 0 on fail.

Source file: *zxutil.c*

1.3.21 *zx_add_xmlns_if_not_seen()*

For WO encoder the order of *xmlns* declarations is not known at compile time. Thus we first add them to the *pop_seen* list using insertion sort (*pop_seen* is smallest and prefixes grow from there) and then later render the list using *zx_enc_seen()*.

Source file: *zxns.c*

1.3.22 *zx_alloc()*

ZX memory allocator that does not zero the buffer. Allocation is potentially done relative to ZX context *c*, though actual (2008) implementation simply uses *malloc(3)*.

Rather than reference this function directly, you should use the *ZX_ALLOC()* macro as much as possible.

Some implementations may take *c->mx* mutex lock. However, they will do so such that no deadlock will result even if already taken.

Source file: *zxlib.c*

1.3.23 *zx_alloc_sprintf()*

sprintf(3) implementation that will grab its memory from ZX memory allocator.

Source file: *zxlib.c*

1.3.24 *zx_alloc_vasprintf()*

vasprintf(3) implementation that will grab its memory from ZX memory allocator.

Source file: *zxlib.c*

1.3.25 *zx_date_time_to_secs()*

Convert a date-time format timestamp into seconds since Unix epoch. Format is as follows

```
01234567890123456789
yyyy-MM-ddThh:mm:ssZ
```

Source file: *zxutil.c*

1.3.26 *zx_dup_cstr()*

ZX verion of *strdup()*.

Source file: *zxlib.c*

1.3.27 *zx_dup_len_str()*

Construct *zx_str* by duplication of raw string data of given length.

Source file: *zxlib.c*

1.3.28 *zx_dup_str()*

Construct *zx_str* by duplication of C string.

Source file: *zxlib.c*

1.3.29 *zx_free()*

ZX memory free'er. Freeing is potentially done relative to ZX context *c*, though actual (2008) implementation simply uses *free(3)*.

Rather than reference this function directly, you should use the *ZX_FREE()* macro as much as possible.

Source file: *zxlib.c*

1.3.30 *zx_get_rsa_pub_from_cert()*

Obtain RSA public key from X509 certificate. The certificate must have been previously read into a data structure. See *zxid_read_cert()* and *zxid_extract_cert()*

Source file: *zxcrypto.c*

1.3.31 *zx_hexdec()*

Especially useful as *yubikey_modhex_decode()* replacement. Supports inplace conversion. Does not nul terminate.

Source file: *zxutil.c*

1.3.32 *zx_init_ctx()*

Allocate new ZX object and initialize it in standard way, i.e. use *malloc(3)* for memory allocation.

Source file: *zxidconf.c*

1.3.33 *zx_is_ns_prefix()*

Given known namespace, does the prefix refer to it, either natively or through an alias.

Source file: `zxns.c`

1.3.34 *zx_len_so_common()*

Render the unknown attributes list. CSE for almost all tags.

Source file: `zxlib.c`

1.3.35 *zx_md5_crypt()*

Compute MD5-Crypt password hash (starts by 1)

pw Password in plain

salt 0-8 chars of salt. Preceding 1 is automatically skipped. Salt ends in *ornul*.

buf must be at least 120 chars

return buf, nul terminated

Source file: `zxcrypto.c`

1.3.36 *zx_memmem()*

ZX implementation of *memmem(3)* for platforms that do not have this.

Source file: `zxlib.c`

1.3.37 *zx_new_len_str()*

Newly allocated string (node and data) of specified length, but uninitialized

Source file: `zxlib.c`

1.3.38 *zx_new_simple_elem()*

Construct new simple element from `zx_str` by referencing, not copying, it.

Source file: `zxlib.c`

1.3.39 *zx_prefix_seen_whine()*

zx_prefix_seen_whine() is a good place to detect, and add stub for, wholly unknown prefixes.

Source file: `zxns.c`

1.3.40 *zx_prepare_dec_ctx()*

Prepare a context for decoding XML. N.B. Often you would wrap this in locks, like

```
LOCK(cf->ctx->mx, "valid");
zx_prepare_dec_ctx(cf->ctx, zx_ns_tab, ss->s, ss->s + ss->len);
r = zx_DEC_root(cf->ctx, 0, 1);
UNLOCK(cf->ctx->mx, "valid");
```

Source file: `zxlib.c`

1.3.41 *zx_push_seen()*

See if prefix has been seen, and in the same meaning. If not, allocate a new node and push or add it to the doubly linked list as well as to the `pop_seen` list. Returns 0 if no addition was done (i.e. ns had been seen already).

Source file: `zxns.c`

1.3.42 *zx_rand()*

ZXID centralized hook for obtaining random numbers. This backends to OpenSSL random number generator and seeds from `/dev/urandom` where available. If you want to use `/dev/random`, which may block, you need to recompile with `ZXID_TRUE_RAND` set to true.

Source file: `zxcrypto.c`

1.3.43 *zx_ref_len_simple_elem()*

Construct new simple element by referencing, not copying, raw string data of given length.

Source file: `zxlib.c`

1.3.44 *zx_ref_len_str()*

Construct `zx_str` from length and raw string data, which will be referenced, not copied.

Source file: `zxlib.c`

1.3.45 *zx_ref_simple_elem()*

Construct new simple element by referencing, not copying, C string.

Source file: `zxlib.c`

1.3.46 *zx_ref_str()*

Construct *zx_str* from C string, which will be referenced, not copied.

Source file: *zxlib.c*

1.3.47 *zx_report_openssl_error()*

Walk through the OpenSSL error stack and dump it to the *stderr*.

logkey Way for caller to indicate what the OpenSSL errors are all about

return Number of open SSL errors processed, or 0 if none. Often ignored.

Source file: *zxsig.c*

1.3.48 *zx_reset_ctx()*

Reset the seen doubly linked list to empty and initialize memory allocation related function pointers to system *malloc(3)*. Without such initialization, any memory allocation activity as well as any XML parsing activity is doomed to segmentation fault.

Source file: *zxidconf.c*

1.3.49 *zx_rsa_priv_dec()*

RSA private key decryption. See *zxid_read_private_key()* and *zxid_extract_private_key()* for ways to read in the private key data structure. N.B. This function *only* does the private key part. It does not perform combined dec-session-key-with-priv-key-and-then-data-with-session-key operation, though this function could be used as a component to implement such a system.

This is considered a low level function. See *zxenc_privkey_dec()* for a higher level solution.

Source file: *zxcrypto.c*

1.3.50 *zx_rsa_priv_enc()*

RSA private key encryption. See *zxid_read_private_key()* and *zxid_extract_private_key()* for ways to read in the private key data structure.

Source file: *zxcrypto.c*

1.3.51 *zx_rsa_pub_dec()*

RSA public key decryption. See *zx_get_rsa_pub_from_cert()* for a way to obtain public key data structure.

Source file: *zxcrypto.c*

1.3.52 `zx_rsa_pub_enc()`

RSA public key encryption. See `zx_get_rsa_pub_from_cert()` for a way to obtain public key data structure. N.B. This function *only* does the public key part. It does not perform combined enc-session-key-with-pub-key-and-then-data-with-session-key operation, though this function could be used as a component to implement such a system.

This is considered a low level function. See `zxenc_pubkey_enc()` for a higher level solution.

Source file: `zxcrypto.c`

1.3.53 `zx_scan_xmlns()`

When trying to scan an element, an annoying feature of XML namespaces is that the namespace may be declared in a `xmlns` attribute within the element itself. Thus at the time of scanning the `<ns:element` part we can't know its namespace. What a lousy design. In order to handle this we need to either backtrack or make a special case forward scan for `xmlns` attributes (which is redundant with normal attribute scanning). It seems simpler to do the latter, so here goes...

The return value represents the list of namespaces that were newly declared at this level, i.e. pushed to the seen stacks. This list is used to pop the seen stacks after we are through with the element.

Source file: `zxns.c`

1.3.54 `zx_str_conv()`

`zx_str_conv()` helps SWIG typemaps to achieve natural conversion to native length + data representations of scripting languages. Should not need to use directly.

Source file: `zxlib.c`

1.3.55 `zx_str_ends_in()`

Check if string ends in suffix

Source file: `zxlib.c`

1.3.56 `zx_str_free()`

Free both the `zx_str` node and the underlying string data

Source file: `zxlib.c`

1.3.57 `zx_str_to_c()`

Convert `zx_str` to C string. The ZX context will provide the memory.

Source file: `zxlib.c`

1.3.58 `zx_tok_by_ns()`

Disambiguate token by considering its namespace. See `zx_attr_lookup()`, `zx_elem_lookup()`
For attributes the namespaceless case is considered.

Source file: `zxns.c`

1.3.59 `zx_url_encode()`

Perform URL encoding on buffer. New output buffer is allocated. The low level work
is performed by `zx_url_encode_raw()`.

N.B. For `zx_url_decode()` operation see `URL_DECODE()` macro in `errmac.h`

Source file: `zxutil.c`

1.3.60 `zx_url_encode_len()`

Compute length of the URL encoded string. The encoding is done to characters listed
in `URL_BAD()` macro in `zxutil.c`. return: Required buffer size, including nul term.
Subtract 1 for string length.

Source file: `zxutil.c`

1.3.61 `zx_url_encode_raw()`

URL encode input into output. The encoding is done to characters listed in `URL_BAD()`
macro in `zxutil.c`. The output must already have been allocated to correct length
(which can be obtained from `zx_url_encode_len()` function). `zx_url_encode()` is higher
level function that does just that. Raw version does not nul terminate. Returns pointer
one past last byte written.

Source file: `zxutil.c`

1.3.62 `zx_xmlns_decl()`

Process XML namespace declaration, trying to match it by its declared namespace
URI. Should this fail, we will attempt to match by customary (at least in our opinion)
namespace prefixes. If deprecated namespaces are detected, they are handled as aliases.

Source file: `zxns.c`

1.3.63 `zx_zalloc()`

ZX memory allocator that zeroes the buffer. Allocation is potentially done relative to
ZX context `c`, though actual (2008) implementation simply uses `malloc(3)`.

Rather than reference this function directly, you should use the `ZX_ALLOC()` macro as
much as possible.

Source file: `zxlib.c`

1.3.64 *zx_zlib_raw_deflate()*

Compress data using zlib-deflate (RFC1951). The deflated data will be in new buffer, which is returned. `out_len` will indicate the length of the compressed data. Since the compressed data will be binary, there is no provision for nul termination. Caveat: RFC1951 is not same as gzip.

Source file: `zxutil.c`

1.3.65 *zx_zlib_raw_inflate()*

Decompress zlib-deflate (RFC1951) compressed data. The decompressed data will be in a newly allocated buffer which is returned. The length of the decompressed data is returned via `out_len`. The buffer will always be at least one byte longer than indicated by `out_len` - this should allow safe nul termination (but the decompressed data itself may contain any number of nuls). Caveat: RFC1951 is not same as gzip.

Source file: `zxutil.c`

1.3.66 *zxenc_privkey_dec()*

Private key decryption using XML-ENC. The encryption algorithm is auto-detected from the XML-ENC data. The private key is looked up from the configuration object.

cf ZXID configuration object, used for memory allocation

ed Encrypted data as XML data structure

ek Symmetric encryption key data structure. If not supplied, the EncryptedKey element from EncryptedData is used

return Decrypted data as `zx_str`. Caller should free this memory.

Source file: `zxsig.c`

1.3.67 *zxenc_pubkey_enc()*

Public key encryption using XML-ENC. The encryption algorithm is auto-detected from the XML-ENC data.

cf ZXID configuration object, used for memory allocation

data Data blob to encrypt. Typically serialized XML

ekp Result parameter. XML data structure corresponding to the `<EncryptedKey>` element will be returned. This is the encrypted symmetric key (which is pseudo-random generated inside this function)

cert Certificate containing the public key used to encrypt the symmetric key

idsuffix Use to generate XML Id attributes for `<EncryptedKey>` and `<EncryptedData>`

return Encrypted data as XML data structure. Caller should free this memory.

Source file: `zxsig.c`

1.3.68 *zxenc_symkey_dec()*

Symmetric key decryption using XML-ENC. The encryption algorithm is auto-detected from the XML-ENC data.

cf ZXID configuration object, used for memory allocation

ed Encrypted data as XML data structure

symkey Symmetric key used for decryption

return Decrypted data as *zx_str*. Caller should free this memory.

Source file: *zxsig.c*

1.3.69 *zxenc_symkey_enc()*

Symmetric key encryption using XML-ENC. The encryption algorithm is auto-detected from the XML-ENC data.

cf ZXID configuration object, used for memory allocation

data Data blob to encrypt. Typically serialized XML

ed_id The value of the *Id* XML attribute of the `<EncryptedData>` element

symkey Raw symmetric key used for encryption

symkey_id The value of the *Id* XML attribute of the `<EncryptedKey>` element

return Encrypted data as XML data structure. Caller should free this memory.

Example of XML-ENC encrypted data

```
<sa:EncryptedID>
  <e:EncryptedData
    xmlns:e="http://www.w3.org/2001/04/xmlenc#"
    Id="ED38"
    Type="http://www.w3.org/2001/04/xmlenc#Element">
    <e:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <ds:RetrievalMethod
        Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"
        URI="#EK38"/></ds:RetrievalMethod>
      # N.B. hash
    <e:CipherData>
      <e:CipherValue>FWfOV7aytBE2xIMe...YTA3ImLf9JCM/vdLIMizMf1</e:CipherValue>
    </e:EncryptedData>
  </sa:EncryptedID>
  <e:EncryptedKey xmlns:e="http://www.w3.org/2001/04/xmlenc#" Id="EK38">
    <e:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
```

```
<ds:X509Data>
  <ds:X509Certificate>***</></></>
<e:CipherData>
  <e:CipherValue>xf5HkmQM68t...7zRbxbkqtniIVnxBHjKA=</></>
<e:ReferenceList>
  <e:DataReference URI="#ED38"/></></></>
```

N.B. hash

Source file: zxsig.c

1.3.70 *zxid_OK()*

Create SAML <Status> element indicating success.

Source file: zxidmk.c

1.3.71 *zxid_ac_desc()*

Generate Assertion Consumer Service (SSO) Descriptor metadata fragment [?].

Source file: zxidmeta.c

1.3.72 *zxid_add_a7n_at_to_pool()*

Add Attribute Statements of an Assertion to pool, applying NEED, WANT, and IN-MAP

Source file: zxidpool.c

1.3.73 *zxid_add_at_values()*

Add values, applying NEED, WANT, and INMAP

Source file: zxidpool.c

1.3.74 *zxid_add_attr_to_ses()*

Add simple attribute to pool, applying NEED, WANT, and INMAP

Source file: zxidpool.c

1.3.75 *zxid_add_fed_tok_to_epr()*

Consider an EPR and user and generate the necessary access credential (SAML a7n). The EPR is modified in place. Returns logging keyword indicating which kind of token was issued.

Source file: zxidpssso.c

1.3.76 *zxid_add_header_refs()*

For purposes of signing, add references and canon forms of all known SOAP headers

Source file: `zxidwsc.c`

1.3.77 *zxid_add_ldif_attrs()*

Parse LDIF format and insert attributes to linked list. Return new head of the list. *** illegal input causes corrupt pointer. For example query string input causes corruption.

Source file: `zxidpssso.c`

1.3.78 *zxid_alloc_ses()*

Allocate memory for session object. Used with *zxid_simple_cf_ses()*.

Source file: `zxidsses.c`

1.3.79 *zxid_an_page_cf()*

Generate IdP Authentication Page.

Either outputs the authentication screen to stdout or returns string of HTML (at specified automation level). If `res_len` is supplied, the string length is returned in `res_len`. Otherwise you can just run *strlen()* on return value.

The *ssoreq* parameter is used for conveying the original AuthnReq from the SP, encoded in SAMLRequest, for processing after the authentication step.

N.B. More complete documentation is available in `zxid-simple.pd`

Source file: `zxidsimp.c`

1.3.80 *zxid_anoint_a7n()*

Helper function to sign, if needed, and log the issued assertion. Checks for Assertion ID duplicate and returns 0 on failure (i.e. duplicate), 1 on success.

Source file: `zxidpssso.c`

1.3.81 *zxid_anoint_sso_resp()*

Helper function to sign, if needed, and log the issued response. Checks for message ID duplicate and returns 0 on failure (i.e. duplicate), or the canonicalized response message string on success. This string may be useful for caller to send further and should be freed by the caller.

Source file: `zxidpssso.c`

1.3.82 *zxid_ar_desc()*

Generate Artifact Resolution (AR) Descriptor idp metadata fragment [?].

Source file: `zxidmeta.c`

1.3.83 *zxid_as_call_ses()*

Authentication Service Client See also: *zxid_idp_as_do()*

Source file: `zxidssso.c`

1.3.84 *zxid_az()*

See *zxid_az_cf()* for description. Only difference is that the configuration is accepted as a string instead of an object.

Source file: `zxidpep.c`

1.3.85 *zxid_cache_epr()*

Serialize EPR data structure to XML and write it to session's EPR cache under file name that is both unique and indicates the service type.

cf ZXID configuration object, also used for memory allocation

ses Session object in whose EPR cache the file will be located

epr XML data structure representing the EPR

return 1 on success, 0 on failure

Source file: `zxidepr.c`

1.3.86 *zxid_callf()*

Call web service, printf style. See *zxid_call()* for more documentation.

Source file: `zxidwsc.c`

1.3.87 *zxid_cdc_check()*

Process second part of Common Domain Cookie redirection. See *zxid_cdc_read()* for first part.

The SAML CDC is a standards based method for SSO IdP discovery.

Source file: `zxidcdc.c`

1.3.88 *zxid_cdc_read()*

Read Common Domain Cookie and formulate HTTP redirection to pass it back.

Limitations In its current form (2008) this function only works for CGI scripts.

The SAML CDC is a standards based method for SSO IdP discovery.

Source file: `zxidcdc.c`

1.3.89 *zxid_conf_to_cf_len()*

Convert configuration string `conf` to configuration object `cf`.

Source file: `zxidsimp.c`

1.3.90 *zxid_curl_read_data()*

Call back used by Curl to move data from application buffer to Curl internal send buffer. Internal. Do not use directly.

Source file: `zxidcurl.c`

1.3.91 *zxid_curl_write_data()*

Call back used by Curl to move received data to application buffer. Internal. Do not use directly.

Source file: `zxidcurl.c`

1.3.92 *zxid_date_time()*

Format a date-time string as usually used in XML, SAML, and Liberty. Apparently there are two ways to format this: with or with-out milliseconds. ZXID accepts either form as input, as they are both legal, but will only generate the without milliseconds form. Some other softwares are buggy and fail to accept the without milliseconds form. You can change the format at compile time.

Source file: `zxidlib.c`

1.3.93 *zxid_dec_a7n()*

Extract an assertion, decrypting EncryptedAssertion if needed.

Source file: `zxidsp.c`

1.3.94 *zxid_decrypt_nameid()*

Given NameID or <EncryptedID>, return Name ID. Typically used by SSO and SLO. If unencrypted NameID is available, then decryption will not be attempted. This facilitates code that handles either encrypted or non-encrypted case in one line:

```
req->NameID = zxid_decrypt_nameid(cf, req->NameID, req->EncryptedID);
```

cf ZXID configuration object, also used for memory allocation

nid XML data structure for Name ID. Possibly 0 (NULL). In that case *encid* should be specified.

encid XML Data Structure for Encrypted Name ID. If no *nid* is specified, this structure is decrypted and its contents returned as the Name ID

return XML data structure corresponding to (possibly decrypted) Name ID

Source file: *zxidlib.c*

1.3.95 *zxid_decrypt_newnym()*

Given new nym or <NewEncryptedID>, return Name ID. Typically used by Name ID Management

cf ZXID configuration object, also used for memory allocation

newnym XML data structure for new Name ID. Possibly 0 (NULL). In that case *encid* should be specified.

encid XML Data Structure for Encrypted Name ID. If no *newnym* is specified, this structure is decrypted and its contents returned as the Name ID

return XML data structure corresponding to (possibly decrypted) new Name ID

Source file: *zxidlib.c*

1.3.96 *zxid_del_ses()*

Delete, or archive, session object from file system. Assertion, if any, is not deleted. This is called upon explicit logout events. However, in reality many sessions are simply abandoned, thus a deploying site should implement some mechanism, such as a *cron(8)* job to remove or archive expired sessions.

Source file: *zxidses.c*

1.3.97 *zxid_di_query()*

Server side Discovery Service Query processing. See also *zxid_gen_bootstraps()*

Source file: *zxiddi.c*

1.3.98 *zxid_epr_path()*

Compute (and fold) unique EPR name according to `/var/zxid/ses/SESID/SVC,SHA1`

This name format is designed to ensure unique name for each EPR, while also making it easy to determine the service type from the name. This is useful in the common case where WSC wants to call a specific type of web service.

cf ZXID configuration object, also used for memory allocation

dir Directory, such as "ses/"

sid Session ID whose EPR cache the file *is/will* be located

buf result parameter. The buffer, which must have been allocated, will be modified to have the path. The path will be nul terminated.

buf_len The length of the buf (including nul termination), usually *sizeof(buf)*

svc Service name

cont content of EPR, used to compute sha1 hash that becomes part of the file name

return 0 on success (the real return value is returned via buf result parameter)

N.B. This function relies on specific, ANSI documented, functioning of *snprintf(3)* library function. Unfortunately, it has been found that on some platforms this function only works correctly in the 'C' locale. If you suspect this to be the case, you may want to try

```
export LANG=C
```

especially if you get errors about multibyte characters.

Source file: `zxidepr.c`

1.3.99 *zxid_extract_body()*

Extract from a string representing SOAP envelope, the payload part in the body.

Source file: `zxidlib.c`

1.3.100 *zxid_extract_cert()*

Extract a certificate from PEM encoded string.

Source file: `zxidconf.c`

1.3.101 *zxid_extract_issuer()*

Look for issuer in all messages we support.

Source file: `zxiddec.c`

1.3.102 *zxid_extract_private_key()*

Extract a private key from PEM encoded string.

Source file: `zxidconf.c`

1.3.103 *zxid_find_at()*

Check whether attribute is in pool.

Source file: `zxidconf.c`

1.3.104 *zxid_find_attribute()*

Look into attribute *statement(s)* of an assertion and scan for *n*th occurrence of named attribute. Ordering of attributes is according to their occurrence in attribute statement, or more broadly according to ordering of the attribute statements themselves.

- NULL or zero length *nfmt* (name format) will match any
- NULL or zero length *name* will match any
- NULL or zero length *friendly* (name) will match any
- minus one (-1) as either length field will cause *strlen()* to be done
- the index *n* is one based

Arguments

a7n Assertion data structure, obtained from XML parsing

nfmt_len Length of the name format, or 0 if no matching by name format is desired

nfmt name format to match (or 0)

name_len Length of the attribute name, or 0 if no matching by attribute name is desired

name attribute name to match (or 0)

friendly_len Length of the friendly name, or 0 if no matching by friendly name is desired

friendly friendly name to match (or 0)

n Howmany instance of the matching attribute is desired. 1 means first.

return Data structure representing the matching attribute.

Source file: `zxida7n.c`

1.3.105 *zxid_find_cstr_list()*

Check whether name is in the list. Used for Local PDP wite and black lists.

Source file: `zxidconf.c`

1.3.106 *zxid_find_epr()*

Search the EPRs cached under the session for a match. First directory is searched for files whose name starts by service type. These files are opened and parsed as EPR and further checks are made. The nth match is returned. 1 means first. Typical name: `/var/zxid/ses/SESID/SVCTYPE,SHA1`

cf ZXID configuration object, also used for memory allocation

ses Session object in whose EPR cache the file is searched

svc Service type (usually a URN)

url (Optional) If provided, this argument has to match either the ProviderID, EntityID, or actual service endpoint URL.

di_opt (Optional) Additional discovery options for selecting the service, query string format

action (Optional) The action, or method, that must be invocable on the service

n How manieth matching instance is returned. 1 means first

return EPR data structure (or linked list of EPRs) on success, 0 on failure

Source file: `zxidepr.c`

1.3.107 *zxid_find_map()*

Check whether attribute is in a (needed or wanted) list. Just a linear scan as it is simple and good enough for handful of attributes.

Source file: `zxidconf.c`

1.3.108 *zxid_find_ses()*

Find a session object by a number of criteria.

cf ZXID configuration object

ses Result parameter. Must have been previously allocated. This will be modified to match the found session.

ses_ix Session Index, usually from SSO asserion or from SLO request. If not supplied (i.e. 0), the **nid** MUST be supplied and will be used as sole basis for deleting the session.

nid The idp assigned Name ID associated with the session. If supplied as 0, then `ses_ix` MUST be supplied and will be used to determine which session is deleted.

return 0 unknown session or error, 1 session found successfully

Source file: `zxidses.c`

1.3.109 `zxid_fold_svc()`

Fold service type (or any URN or URL) to file name.

Source file: `zxidepr.c`

1.3.110 `zxid_gen_boots()`

Process `.bs` directory. See also `zxid_di_query()`

Source file: `zxidpssso.c`

1.3.111 `zxid_get_ent()`

Wrapper for `zxid_get_ent_ss()`, which see.

Source file: `zxidmeta.c`

1.3.112 `zxid_get_ent_by_sha1_name()`

Given `sha1_name`, check in memory cache and if not, the disk cache. Do not try net (WKL).

Source file: `zxidmeta.c`

1.3.113 `zxid_get_ent_by_succinct_id()`

In artifact profile concept of "succinct id" appears. If you have one of those, you can use this function to fetch the entity metadata. Only in-memory and disk caches will be tried. No network connection (WKL) will be initiated.

Source file: `zxidmeta.c`

1.3.114 `zxid_get_ent_from_cache()`

Compute `sha1_name` for an entity and then read the metadata from the CoT metadata cache directory, e.g. `/var/zxit/cot`.

Source file: `zxidmeta.c`

1.3.115 *zxid_get_ent_from_file()*

Read metadata from a file.

Usually the file will be named according to "sha1 name", which is safe base64 encoded SHA1 digest hash over the EntityID. This is used to ensure unique file name for each entity. However, this function will in fact read from any file name supplied.

See also *zxid_get_ent_from_cache()* which will compute the sha1_name and then read the metadata.

Source file: `zxidmeta.c`

1.3.116 *zxid_get_epr_a7n()*

Accessor function for extracting endpoint's SAML2 assertion token.

Source file: `zxidepr.c`

1.3.117 *zxid_get_epr_address()*

Accessor function for extracting endpoint address URL.

Source file: `zxidepr.c`

1.3.118 *zxid_get_epr_desc()*

Accessor function for extracting endpoint Description (Abstract).

Source file: `zxidepr.c`

1.3.119 *zxid_get_epr_entid()*

Accessor function for extracting endpoint ProviderID.

Source file: `zxidepr.c`

1.3.120 *zxid_get_meta()*

Send HTTP request for metadata using Well Known Location (WKL) method and wait for response. Send the message to the server using Curl. Return the metadata as parsed XML for the entity. This call will block while the HTTP request-response is happening.

cf ZXID configuration object, also used for memory allocation

url Where the request will be sent, i.e. the WKL

return XML data structure representing the entity, or 0 upon failure

The underlying HTTP client is libcurl. While libcurl is documented to be "entirely thread safe", one limitation is that curl handle can not be shared between threads. Since we keep the curl handle a part of the configuration object, which may be shared between threads, we need to take a lock for duration of the curl operation. Thus any given configuration object can have only one HTTP request active at a time. If you need more parallelism, you need more configuration objects.

Source file: `zxidcurl.c`

1.3.121 `zxid_get_meta_ss()`

Wrapper for `zxid_get_meta()` so you can provide the URL as `zx_str`.

Source file: `zxidcurl.c`

1.3.122 `zxid_get_ses()`

Get simple session object from the filesystem. This just gets the nameid and reference to the assertion. Use `zxid_get_ses_sso_a7n()` to actually load the assertion, if needed. Or `zxid_ses_to_pool()` if you need attributes as well. Returns 1 if session gotten, 0 if fail.

Source file: `zxidses.c`

1.3.123 `zxid_get_ses_idp()`

Get the IdP entity associated with the session. Generally this is figured out from the Issuer field of the SSO assertion that started the session.

Source file: `zxidses.c`

1.3.124 `zxid_get_ses_sso_a7n()`

When session is loaded, we only get the reference to assertion. This is to avoid parsing overhead when the assertion really is not needed. But when the assertion is needed, you have to call this function to load it from file (under `/var/zxid/log/rely/EID/a7n/AID`) and parse it.

Source file: `zxidses.c`

1.3.125 `zxid_get_sid_from_cookie()`

Try to extract session ID from a cookie. The extracted value, if any, will be deposited in `cgi->sid`. If no session ID is found, then `cgi->sid` is not modified. The name of the cookie is determined by configuration option `SES_COOKIE_NAME` (see `zxidconf.h`).

For original Netscape cookie spec see: http://curl.haxx.se/rfc/cookie_spec.html (Oct2007)

Example

```
ONE_COOKIE=aaa; ZXIDSES=S12cvd324; SOME_OTHER_COOKIE=...
```

Source file: `zxidcgi.c`

1.3.126 `zxid_get_user_nameid()`

Locate user file using a NameID, which may be old or current. If old, chase the MNIptr fields until current is found. Mainly used to support MNI.

Source file: `zxiduser.c`

1.3.127 `zxid_idp_as_do()`

ID-WSF Authentication Service: check password and emit *bootstrap(s)* To generate the data, use:

```
perl -MMIME::Base64 -e 'print encode_base64("\0user\0pw\0")'
perl -MMIME::Base64 -e 'print encode_base64("\0tastest\0tas123\0")'
```

See also: `zxid_as_call_ses()`

Source file: `zxidpssso.c`

1.3.128 `zxid_idp_dispatch()`

Dispatch redirect and post binding requests.

return a string (such as Location: header) and let the caller output it.

Source file: `zxididpx.c`

1.3.129 `zxid_idp_loc()`

SAML2 service locator. Given desired service, like SLO or MNI, and possibly binding, locate the appropriate service descriptor from the IdP metadata.

cf ZXID configuration object, used for preferences and for memory allocation

cgi May contain CGI variables that further indicate preference. Often specified as 0 (no preference).

ses Session object, which may be used to remember historical events, such as binding of SSO transaction, that may act as preferences for binding. The session MUST have assertion.

idp_meta Metadata for the IdP

svc_type The desired service, indicated as URN

binding preferred binding URN, or 0 if no preference. In that case the built in preference is used, or if that is indifferent, then first applicable metadata item is picked. If IdP only supports one binding 0 will match that. If nonzero, then the IdP metadata MUST have exactly matching entry or else 0 is returned.

return URL for accessing the service or 0 upon failure

Limitation: If binding is not specified, it may be ambiguous what binding the returned URL relates to. Generally the decision will have been taken prior to calling this function.

Source file: `zxidloc.c`

1.3.130 `zxid_idp_loc_raw()`

Raw computation of IdP URL given service type, binding, and whether operation is a request. See `zxid_idp_loc()` for full description.

Source file: `zxidloc.c`

1.3.131 `zxid_idp_slo_do()`

Process IdP SLO request. The IdP SLO Requests are complicated by the need to log the user out of other SPs as well, if they belong to same session. Part of the complication is figuring out what constitutes "same session". Finally, the redirect profiles may be "hairy" to handle if some SP does not collaborate in the SLO. For SOAP similar problem exists, but it should be manageable.

Source file: `zxidslo.c`

1.3.132 `zxid_idp_soap()`

Determine URL for SOAP binding to given service and perform a SOAP call.

cf ZXID configuration object

cgi CGI variables that may influence determination of end point. Or 0 if no preference.

ses Session information that may influence the choice of the end point. The session MUST have assertion.

idp_meta Metadata for the IdP

svc_type The desired service, indicated as URN

body XML data structure for the SOAP call <Body> element payload

return XML data structure for Body element of the SOAP call response.

Source file: `zxidloc.c`

1.3.133 *zxid_idp_soap_dispatch()*

SOAP dispatch can also handle requests and responses received via artifact resolution. However only some combinations make sense. Return 0 for failure, otherwise some success code such as ZXID_SSO_OK *** NOT CALLED FROM ANYWHERE. See *zxid_sp_soap_dispatch()* for real action

Source file: `zxididpx.c`

1.3.134 *zxid_idp_soap_parse()*

Return 0 for failure, otherwise some success code such as ZXID_SSO_OK

Source file: `zxididpx.c`

1.3.135 *zxid_idp_sso_desc()*

Generate IdP SSO Descriptor metadata fragment [?].

Source file: `zxidmeta.c`

1.3.136 *zxid_init_conf_ctx()*

If `zxid_path` is supplied as NULL, then a minimal initialization of the context is performed. Certificate and key operations as well as CURL initialization are omitted. However the `zx_ctx` is installed so that memory allocation against the context should work.

Source file: `zxidconf.c`

1.3.137 *zxid_ins_xacml_az_stmt()*

Create Authorization Decision

Source file: `zxidsp.c`

1.3.138 *zxid_is_needed()*

Check whether attribute is in a (needed or wanted) list. Just a linear scan as it is simple and good enough for handful of attributes.

Source file: `zxidconf.c`

1.3.139 *zxid_issuer()*

Generate Issuer value. Issuer is often same as Entity ID, but sometimes it will be affiliation ID. This function is a low level interface. Usually you would want to use *zxid_my_issuer()*.

Source file: `zxidmeta.c`

1.3.140 *zxid_key_desc()*

Generate key descriptor metadata fragment given X509 certificate [?].

Source file: `zxidmeta.c`

1.3.141 *zxid_key_info()*

Generate XML-DSIG key info given X509 certificate.

Source file: `zxidmeta.c`

1.3.142 *zxid_lazy_load_sign_cert_and_pkey()*

Lazy load signing certificate and private key. This reads them from disk if needed. If they do not exist and `auto_cert` is enabled, they will be generated on disk and the read. Once read from disk, they will be cached in memory.

Source file: `zxidconf.c`

1.3.143 *zxid_lecp_check()*

Check for ECP indications in HTTP request headers and initiate PAOS based Single Sign On, i.e AuthnRequest. This is part of the SAML2 Enhanced Client Proxy profile.

Limitation Current (2008) code only works in CGI environment due to reliance on environment variables.

If you do not know what PAOS, ECP or LECP means, you should read [?] specification.

Source file: `zxidecp.c`

1.3.144 *zxid_load_atsrc()*

Parse ATTRSRC specification and add it to linked list namespaceA, BweightaccessparamURLAAPMLrefotherLimext;

Source file: `zxidconf.c`

1.3.145 *zxid_load_cot_cache()*

Usually you will want to use the *get_ent()* methods if you need only specific entities. Loading the entire cache is expensive and only useful if you really need to enumerate through all available entities. This may be the case when rendering login buttons for all IdPs in a user interface.

cf ZXID configuration object

return Linked list of Entity objects (metadata) for CoT partners

Source file: `zxidmeta.c`

1.3.146 *zxid_load_cstr_list()*

Parse ATTRSRC specification and add it to linked list namespaceA, BweightaccessparamURLAAPMLrefotherLimext;

Source file: zxidconf.c

1.3.147 *zxid_load_map()*

Parse map specification and add it to linked list srcnsArulebext;srcArulebext;...

Source file: zxidconf.c

1.3.148 *zxid_load_need()*

Parse need specification and add it to linked list A,Busageretentionobligext;A,Busageretentionobligext;...

Source file: zxidconf.c

1.3.149 *zxid_localpdp()*

Local Policy Decision Point - decide on role and idpnid. Return: 0 for Deny and 1 for Permit.

Source file: zxidsimp.c

1.3.150 *zxid_map_sec_mech()*

Try to map security mechanisms across different frame works. Low level function.

Source file: zxidwsc.c

1.3.151 *zxid_map_val()*

Transform content according to map. The returned zx_str will be nul terminated.

Source file: zxidlib.c

1.3.152 *zxid_mk_Status()*

Create SAML protocol <Status> element, given various levels of error input.

Source file: zxidmk.c

1.3.153 *zxid_mk_a7n()*

Constructor for Assertion

Source file: zxidmk.c

1.3.154 *zxid_mk_addr()*

Low level constructor for WSA <Address>.

Source file: `zxidmkwsf.c`

1.3.155 *zxid_mk_an_stmt()*

Construct AuthnStatement

Source file: `zxidmk.c`

1.3.156 *zxid_mk_art_deref()*

Make the body for the ArtifactResolve SOAP message, signing it if needed.

Source file: `zxidmk.c`

1.3.157 *zxid_mk_attribute()*

Construct SAML SAML Attribute

Source file: `zxidmk.c`

1.3.158 *zxid_mk_authn_req()*

Interpret ZXID standard form fields to construct a XML structure for AuthnRequest

Source file: `zxidmk.c`

1.3.159 *zxid_mk_az()*

Construct XACMLAuthzDecisionQuery

Source file: `zxidmk.c`

1.3.160 *zxid_mk_az_cd1()*

Construct XACMLAuthzDecisionQuery according to Commitee Draft 1

Source file: `zxidmk.c`

1.3.161 *zxid_mk_dap_query()*

Low level constructor for <dap:Query>.

Source file: `zxidmkwsf.c`

1.3.162 *zxid_mk_dap_query_item()*

Low level constructor for <dap:QueryItem>.

Source file: zxidmkwsf.c

1.3.163 *zxid_mk_dap_resquery()*

Low level constructor for <dap:ResultQuery>.

Source file: zxidmkwsf.c

1.3.164 *zxid_mk_dap_select()*

Low level constructor for <dap>Select>.

Source file: zxidmkwsf.c

1.3.165 *zxid_mk_dap_subscription()*

Low level constructor for <dap:Subscription>.

Source file: zxidmkwsf.c

1.3.166 *zxid_mk_dap_test_item()*

Low level constructor for <dap:TestItem>.

Source file: zxidmkwsf.c

1.3.167 *zxid_mk_dap_testop()*

Low level constructor for <dap:TestOp>.

Source file: zxidmkwsf.c

1.3.168 *zxid_mk_di_query()*

Low level constructor for discovery <di:Query>.

Source file: zxidmkwsf.c

1.3.169 *zxid_mk_di_req_svc()*

Low level constructor for discovery <di:RequestedService>.

Source file: zxidmkwsf.c

1.3.170 *zxid_mk_ecp_Request_hdr()*

Generate headers for use with Liberty ID-FF 1.2 LECP carried AuthnRequest.

If you do not know what PAOS, ECP or LECP means, you should read [?] specification.

Source file: `zxidecp.c`

1.3.171 *zxid_mk_enc_a7n()*

Create EncryptedAssertion given normal A7N and metadata of destination. Encryption will be done using encryption certificate of the receiver identified by the metadata.

Source file: `zxidmk.c`

1.3.172 *zxid_mk_enc_id()*

Create EncryptedID given normal NameID and metadata of destination. Encryption will be done using encryption certificate of the receiver identified by the metadata.

Source file: `zxidmk.c`

1.3.173 *zxid_mk_id()*

Generate pseudorandom or statistically unique identifier of given length. The unique identifier will be safe base64 encoded.

cf Configuration object, used for memory allocation.

prefix A prefix string, usually used to distinguish classes of unique ids.

bits Number of pseudorandom bits in the unique ID. For best results, bits should be multiple of 24 (3 bytes expands to 4 safe base64 chars)

return The identifier as `zx_str`. Caller should eventually free this memory.

Source file: `zxidlib.c`

1.3.174 *zxid_mk_idp_list()*

Build IDPList of IDPEntry(s) from the IdPs know to us at the moment (our CoT). Can be used for ECP and IdP proxying.

cf ZXID configuration object, used to locate the CoT directory (PATH configuration option) and for memory allocation

binding The SSO protocol binding the qualifying IdPs MUST support, or 0 if anything goes

return IdP list data structure or 0 on failure

Source file: `zxidecp.c`

1.3.175 *zxid_mk_logout()*

Create XML data structure for <LogoutRequest> element. Low level API.

Source file: `zxidmk.c`

1.3.176 *zxid_mk_logout_resp()*

Create XML data structure for <LogoutResponse> element. Low level API.

Source file: `zxidmk.c`

1.3.177 *zxid_mk_lu_Status()*

Create ID-WSF protocol <lu:Status> element, given various levels of error input.

Source file: `zxidmkwsf.c`

1.3.178 *zxid_mk_mni()*

Change SPNameID (newnym supplied), or Terminate federation (newnym not supplied). Create XML data structure for <ManageNameIDRequest> element. Low level API.

Source file: `zxidmk.c`

1.3.179 *zxid_mk_mni_resp()*

Create XML data structure for <ManageNameIDResponse> element. Low level API.

Source file: `zxidmk.c`

1.3.180 *zxid_mk_paos_Request_hdr()*

Generate SOAP headers for use with PAOS carried SAML2 ECP profile AuthnRequest.

If you do not know what PAOS, ECP or LECP means, you should read [?] specification.

Source file: `zxidecp.c`

1.3.181 *zxid_mk_saml_resp()*

Construct SAML protocol Response (such as may be used to carry assertion in SSO)

Source file: `zxidmk.c`

1.3.182 *zxid_mk_self_sig_cert()*

Create Self-Signed Certificate-Private Key pair and Certificate Signing Request This function is invoked when AUTO_CERT is set and a certificate is missing. As this is not expected to be frequent, we are cavalier about releasing the memory needed for each intermediate step.

cf zxid configuration object, of wich cf->ctx will be used for memory allocation

buflen *sizeof(buf)*

buf Buffer used for rendering pem representations of the data

log key Who and why is calling

name Name of the certificate file to be created

See also: *keygen()* in *keygen.c*

Source file: *zxcrypto.c*

1.3.183 *zxid_mk_subj()*

Construct Subject, possibly with EncryptedID

Source file: *zxidmk.c*

1.3.184 *zxid_mk_transient_nid()*

Change NameID to be transient and record corresponding mapping.

Source file: *zxidpss.c*

1.3.185 *zxid_mk_xacml_resp()*

Construct XACML Response

Source file: *zxidmk.c*

1.3.186 *zxid_mni_desc()*

Generate Manage Name Id (MNI) Descriptor metadata fragment [?].

Source file: *zxidmeta.c*

1.3.187 *zxid_mni_do()*

Process <ManageNameIDRequest>, presumably received from IdP. This is very rarely used.

Source file: *zxidmni.c*

1.3.188 *zxid_mni_do_ss()*

Wrapper for *zxid_mni_do()*, which see.

Source file: `zxidmni.c`

1.3.189 *zxid_my_cdc_url()*

Dynamically determine our Common Domain Cookie (IdP discovery) URL.

Source file: `zxidmeta.c`

1.3.190 *zxid_my_issuer()*

Generate Issuer value for our entity. Issuer is often same as Entity ID, but sometimes it will be affiliation ID.

Source file: `zxidmeta.c`

1.3.191 *zxid_new_at()*

Create new (common pool) attribute and add it to a linked list

Source file: `zxidconf.c`

1.3.192 *zxid_new_conf()*

Allocate conf object, but do not actually initialize it with default config or config file. See: *zxid_new_conf_to_cf()* for a more complete solution.

Source file: `zxidconf.c`

1.3.193 *zxid_new_conf_to_cf()*

Create new ZXID configuration object given configuration string and possibly configuration file.

conf Configuration service

return Configuration object

Source file: `zxidsimp.c`

1.3.194 *zxid_nice_sha1()*

Compute (and fold) unique nice sha1 name according to NAME,SHA1

This name format is designed to ensure unique name, while maintaining human readability. This is useful in the common case where WSC wants to call a specific type of web service.

cf ZXID configuration object, also used for memory allocation

buf result parameter. The buffer, which must have been allocated, will be modified to have the path. The path will be nul terminated.

buf_len The length of the buf (including nul termination), usually *sizeof(buf)*

name Often Service name or SP Entity ID

cont content of EPR or the SP EntityID, used to compute sha1 hash that becomes part of the file name

ign_prefix How many characters to ignore from beginning of name: 0 or 7

return 0 on success (the real return value is returned via buf result parameter)

Source file: `zxidepr.c`

1.3.195 *zxid_parse_conf()*

Wrapper with initial error checking for *zxid_parse_conf_raw()*, which see.

Source file: `zxidconf.c`

1.3.196 *zxid_parse_meta()*

Parse Metadata, see [?]. This function is quite low level and assumes it is processing a buffer (which may contain multiple instances of various metadata).

cf ZXID configuration object, used here mainly for memory allocation

md Value-result parameter. Pointer to char pointer pointing to the beginning of the metadata. As metadata is scanned and parsed, this pointer will be advanced

lim End of the metadata buffer

return Entity data structure composed from the metadata.

Source file: `zxidmeta.c`

1.3.197 *zxid_parse_mni()*

Parse a line from `.mni` and form a NameID, unless there is `mniptr`

Source file: `zxiduser.c`

1.3.198 *zxid_pick_sso_profile()*

This function makes the policy decision about which profile to use. It is only used if there was no explicit specification in the CGI form (e.g. "Login (P)" button). Currently it's a stub that always picks the artifact profile. Eventually configuration options or cgi input can be used to determine the profile in a more sophisticated way. Often *zxid_mk_authn_req()* will override the return value of this function by its own inspection of the CGI variables.

Source file: `zxidssso.c`

1.3.199 *zxid_pool_to_json()*

Convert attributes from (session) pool to JSON, applying OUTMAP. *** Need to check escaping JSON values, e.g. " or

Source file: `zxidpool.c`

1.3.200 *zxid_pool_to_qs()*

Convert attributes from (session) pool to query string, applying OUTMAP. *** Need to check multivalue handling. Now all values are simply blurted

out as separate name=value pairs.

*** Need to figure out how to distinguish query string return from

other returns, like redirect. Perhaps arrange dn field always first?

Source file: `zxidpool.c`

1.3.201 *zxid_process_keys()*

Process certificates (public keys) from a metadata for entity. Since one entity can be both IdP and SP, this function may be called twice per entity, with different kd argument.

Source file: `zxidmeta.c`

1.3.202 *zxid_protocol_binding_map_saml2()*

Map SAML protocol binding URN to form field.

Source file: `zxidssso.c`

1.3.203 *zxid_put_ses()*

Create new session object in file system. The assertion must have been created separately.

cf Configuration object

ses Pointer to previously allocated and populated session object

return 1 upon success, 0 on failure.

Source file: `zxidses.c`

1.3.204 *zxid_put_user()*

Create new user object in file system.

Source file: `zxiduser.c`

1.3.205 *zxid_pw_authn()*

Locally authenticate user. If successful, create a session. Expects to get username and password in `cgi->au` and `cgi->ap` respectively. User authentication is done against local database or by default using `/var/zxid/uid/uid/.pw` file. When filesystem backend is used, for safety reasons the uid (user) component can not have certain characters, such as slash (/) or sequences like "..". See also: `zxpaswd.c`

return 0 on failure and sets `cgi->err`; 1 on success

Source file: `zxiduser.c`

1.3.206 *zxid_read_cert()*

Extract a certificate from PEM encoded file.

Source file: `zxidconf.c`

1.3.207 *zxid_read_private_key()*

Extract a private key from PEM encoded file.

Source file: `zxidconf.c`

1.3.208 *zxid_saml2_map_authn_ctx()*

Map authentication contest class ref form field to SAML specified URN string.

Source file: `zxidssoc.c`

1.3.209 *zxid_saml2_map_nid_fmt()*

Map name id format form field to SAML specified URN string.

Source file: `zxidsso.c`

1.3.210 *zxid_saml2_map_protocol_binding()*

Map protocol binding form field to SAML specified URN string.

Source file: `zxidsso.c`

1.3.211 *zxid_saml2_redir()*

SAMLRequest. Return the HTTP 302 redirect LOCATION header + CRLF2. You need to pass this to some application layer facility to effectuate the actual redirect. Wrapper for *zxid_saml2_redir_enc()*. This is different from *zxid_saml2_redir_url()* in that the entire Location header is returned, rather than just the url.

cf ZXID configuration object, also used for memory allocation

loc The URL up to query string

pay_load What should be encoded in the redirect URL. Effectively becomes the query string

relay_state Optional relay state argument. Ends up being encoded in the query string

return HTTP Location header as `zx_str`. The memory should be freed by the caller.

Source file: `zxidlib.c`

1.3.212 *zxid_saml2_redir_url()*

SAMLRequest. Return the URL needed for redirect. You need to pass this to some application layer facility to effectuate the actual redirect. Wrapper for *zxid_saml2_redir_enc()*. This function is different from *zxid_saml2_redir()* in that only the URL is returned, not the complete Location header.

cf ZXID configuration object, also used for memory allocation

loc The URL up to query string

pay_load What should be encoded in the redirect URL. Effectively becomes the query string

relay_state Optional relay state argument. Ends up being encoded in the query string

return URL suitable for redirection as `zx_str`. The memory should be freed by the caller.

Source file: `zxidlib.c`

1.3.213 *zxid_saml2_resp_redir()*

SAMLResponse. Return the HTTP 302 redirect LOCATION header + CRLF2. You need to pass this to some application layer facility to effectuate the actual redirect. Wrapper for *zxid_saml2_redir_enc()*.

cf ZXID configuration object, also used for memory allocation

loc The URL up to query string

pay_load What should be encoded in the redirect URL. Effectively becomes the query string

relay_state Optional relay state argument. Ends up being encoded in the query string

return HTTP Location header as `zx_str`. The memory should be freed by the caller.

Source file: `zxidlib.c`

1.3.214 *zxid_saml_ok()*

Check status codes in SAML response to verify that request was completed OK.

cf ZXID configuration object, also used for memory allocation

cgi CGI variables decoded from the query string. `err` field of the CGI object will be set upon failure.

st The SAML `<Status>` element from the response, as XML data structure

what Explanatory string used in error and log messages

return 1 if SAML message is OK, 0 if message is not OK.

Source file: `zxidlib.c`

1.3.215 *zxid_send_sp_meta()*

Generate our SP metadata and send it to remote partner.

Limitation This function only works with CGI as it will print the serialized metadata straight to stdout. There are other methods for getting metadata without this limitation, e.g. *zxid_sp_meta()*

Source file: `zxidmeta.c`

1.3.216 *zxid_ses_to_json()*

Convert attributes from session to JSON, applying OUTMAP.

Source file: `zxidpool.c`

1.3.217 *zxid_ses_to_ldif()*

Convert attributes from session to LDIF, applying OUTMAP.

Source file: `zxidpool.c`

1.3.218 *zxid_ses_to_qs()*

Convert attributes from session to query string, applying OUTMAP.

Source file: `zxidpool.c`

1.3.219 *zxid_set_opt()*

Set obscure options of ZX and ZXID layers. Used to set debug options. Generally setting these options is not supported, but this function exists to avoid uncontrolled access to global variables. At least this way the unsupported activity will happen in one controlled place where it can be ignored, if need to be. You have been warned.

Source file: `zxidconf.c`

1.3.220 *zxid_set_opt_cstr()*

Set obscure options of ZX and ZXID layers. Used to set debug options. Generally setting these options is not supported, but this function exists to avoid uncontrolled access to global variables. At least this way the unsupported activity will happen in one controlled place where it can be ignored, if need to be. You have been warned.

Source file: `zxidconf.c`

1.3.221 *zxid_show_conf()*

Generate our SP CARML and return it as a string.

Source file: `zxidconf.c`

1.3.222 *zxid_show_cstr_list()*

Pretty print cstr list as used in local PDP.

Source file: `zxidconf.c`

1.3.223 *zxid_show_map()*

Pretty print map chain.

Source file: `zxidconf.c`

1.3.224 *zxid_show_need()*

Pretty print need or want chain.

Source file: `zxidconf.c`

1.3.225 *zxid_sigres_map()*

Map ZXSIG constant to letter for log and string message.

Source file: `zxidssoc.c`

1.3.226 *zxid_simple()*

Main simple interface. C string nul termination is assumed. Really just a wrapper for *zxid_simple_cf()*.

N.B. More complete documentation is available in `zxid-simple.pdf`

Source file: `zxidsimp.c`

1.3.227 *zxid_simple_cf()*

Allocate simple session and then call simple handler. Strings are length + pointer (no C string nul termination needed). A wrapper for *zxid_simple_cf()*.

cf Configuration object

qs_len Length of the query string. -1 = use *strlen()*

qs Query string (or POST content)

res_len Result parameter. If non-null, will be set to the length of the returned string

auto_flags Automation flags, see `zxid-simple.pdf` for documentation

return String representing protocol action or SSO attributes

N.B. More complete documentation is available in `zxid-simple.pdf`

Source file: `zxidsimp.c`

1.3.228 *zxid_simple_idp_an_ok_do_rest()*

Process IdP side after successful authentication. If IdP was invoked with AuthnReq (in SAMLRequest) then `op=='F'` as set in *zxid_simple_idp_pw_authn()* which will trigger the rest of the SSO protocol in *zxid_simple_ses_active_cf()*. Otherwise just show the IdP management screen.

Source file: `zxidsimp.c`

1.3.229 *zxid_simple_idp_pw_authn()*

Process password authentication form and, if ssoreq (ar=) is present (see *zxid_simple_idp_show_an()* for how it is embedded to hidden form field), proceed to federated SSO. If login fails, redisplay the authentication page.

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidsimp.c*

1.3.230 *zxid_simple_idp_show_an()*

Show Authentication screen. Generally this will be in response to the SP having sent used via redirect carrying AuthnRequest encoded in SAMLRequest query string parameter, per SAML redirect binding [?]. We must preserve SAMLRequest as hidden field in the page for later processing once the authentication step has been taken care of. It will also be passed on the query string to external authentication page if any was configured with AN_PAGE directive.

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidsimp.c*

1.3.231 *zxid_simple_len()*

Process simple configuration and then call simple handler. Strings are length + pointer (no C string nul termination needed). a wrapper for *zxid_simple_cf()*.

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidsimp.c*

1.3.232 *zxid_simple_no_ses_cf()*

Subroutine of *zxid_simple_cf()* for the no session detected/active case.

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidsimp.c*

1.3.233 *zxid_simple_redir_page()*

Helper function to redirect according to auto flags.

Source file: *zxidsimp.c*

1.3.234 *zxid_simple_ses_active_cf()*

Subroutine of *zxid_simple_cf()* for the session active case.

NULL return means the "not logged in" processing is needed, see *zxid_simple_no_ses_cf()*

N.B. More complete documentation is available in *zxid-simple.pd*

Source file: *zxidsimp.c*

1.3.235 *zxid_simple_show_carmml()*

Emit CARML declaration for SP. Corresponds to "o=c" query string.

Source file: `zxidsimp.c`

1.3.236 *zxid_simple_show_conf()*

Dump internal info and configuration. Corresponds to "o=d" query string.

Source file: `zxidsimp.c`

1.3.237 *zxid_simple_show_idp_sel()*

Show IdP selection or login screen.

N.B. More complete documentation is available in `zxid-simple.pd`

Source file: `zxidsimp.c`

1.3.238 *zxid_simple_show_meta()*

Emit metadata. Corresponds to "o=B" query string.

N.B. More complete documentation is available in `zxid-simple.pd`

Source file: `zxidsimp.c`

1.3.239 *zxid_simple_show_page()*

Deal with the various methods of shipping the page, including CGI stdout, or as string with or without headers, as indicated by the `auto_flag`. The page is in `ss`.

Source file: `zxidsimp.c`

1.3.240 *zxid_slo_desc()*

Generate Single Logout (SLO) Descriptor metadata fragment [?].

Source file: `zxidmeta.c`

1.3.241 *zxid_slo_resp_redir()*

Generate SLO Response, SP or IdP variant. The actual session invalidation must be done somewhere else, i.e. this is just the final protocol phase of the SLO.

Source file: `zxidslo.c`

1.3.242 *zxid_snarf_eprs()*

Look into attribute statements of a SSO assertion and extract anything that looks like EPR, storing results in the session for later reference.

cf ZXID configuration object, also used for memory allocation

ses Session object in whose EPR cache will be populated

N.B. This approach ignores the official attribute names totally. Anything that looks like an EPR and that is structurally in right place will work. Typical name `/var/zxid/ses/SESID/SVCTYPE,SHA1`

Source file: `zxidepr.c`

1.3.243 *zxid_snarf_eprs_from_ses()*

Look into attribute statements of a SSO assertion and extract anything that looks like EPR, storing results in the session for later reference.

cf ZXID configuration object, also used for memory allocation

ses Session object in whose EPR cache will be populated

N.B. This approach ignores the official attribute names totally. Anything that looks like an EPR and that is structurally in right place will work. Typical name `/var/zxid/ses/SESID/SVCTYPE,SHA1`

Source file: `zxidepr.c`

1.3.244 *zxid_soap_call_body()*

Encode XML data structure representing SOAP envelope (request) and send the message to the server using Curl. Return the parsed XML response data structure. This call will block while the HTTP request-response is happening. To be called from SSO world. Wrapper for *zxid_soap_call_raw()*.

cf ZXID configuration object, also used for memory allocation

url The endpoint where the request will be sent

body XML data structure representing the SOAP body

return XML data structure representing the response

Source file: `zxidlib.c`

1.3.245 *zxid_soap_call_envelope()*

Encode XML data structure representing SOAP envelope (request) and send the message to the server using Curl. Return the parsed XML response data structure. This call will block while the HTTP request-response is happening. To be called from ID-WSF world. Wrapper for *zxid_soap_call_raw()*.

cf ZXID configuration object, also used for memory allocation

url The endpoint where the request will be sent

env XML data structure representing the request

return XML data structure representing the response

Source file: *zxidlib.c*

1.3.246 *zxid_soap_call_hdr_body()*

Encode XML data structure representing SOAP envelope (request) and send the message to the server using Curl. Return the parsed XML response data structure. This call will block while the HTTP request-response is happening. To be called from SSO world. Wrapper for *zxid_soap_call_raw()*.

cf ZXID configuration object, also used for memory allocation

url The endpoint where the request will be sent

hdr XML data structure representing the SOAP headers. Possibly 0 if no headers are desired

body XML data structure representing the SOAP body

return XML data structure representing the response

Source file: *zxidlib.c*

1.3.247 *zxid_soap_cgi_resp_body()*

Emit to stdout XML data structure representing SOAP envelope (request). Typically used in CGI environment.

cf ZXID configuration object, also used for memory allocation

body XML data structure representing the request

return 0 if fail, ZXID_REDIR_OK if success.

Source file: *zxidlib.c*

1.3.248 *zxid_sp_anon_finalize()*

Fake a login and generate a session. Used if SSO failure is configured to result anonymous session.

cf Configuration object, used to determine time slops, potentially memalloc via cf->ctx

cgi CGI object. signal and sigmsg may be set.

ses Session object. Will be modified according to new session created from the SSO assertion.

return 0 for failure, otherwise some success code such as ZXID_SSO_OK

Source file: zxidssso.c

1.3.249 *zxid_sp_carml()*

Generate our SP CARML and return it as a string.

Source file: zxidmeta.c

1.3.250 *zxid_sp_dig_sso_a7n()*

Extract an assertion from Request, decrypting EncryptedAssertion if needed, and perform SSO

Source file: zxidsp.c

1.3.251 *zxid_sp_dispatch()*

Dispatch redirect or post binding requests (and sometimes responses).

return a string (such as Location: header) and let the caller output it. Sometimes a dummy string is just output to indicate status, e.g. "O" for SSO OK, "K" for normal OK no further action needed, "M" show management screen, "I" forward to IdP dispatch, or "* ERR" for error situations. These special strings are allocated from static storage and MUST NOT be freed. Other strings such as "Location: ..." should be freed by caller.

Source file: zxidsp.c

1.3.252 *zxid_sp_loc()*

SAML2 service locator for SP. Given desired service, like SLO or MNI, and possibly binding, locate the appropriate service descriptor from the Sp metadata.

cf ZXID configuration object, used for preferences and for memory allocation

cgi May contain CGI variables that further indicate preference. Often specified as 0 (no preference).

ses Session object, which may be used to remember historical events, such as binding of SSO transaction, that may act as preferences for binding. The session MUST have assertion.

sp_meta Metadata for the Sp

svc_type The desired service, indicated as URN

binding preferred binding URN, or 0 if no preference. In that case the built in preference is used, or if that is indifferent, then first applicable metadata item is picked. If Sp only supports one binding 0 will match that. If nonzero, then the Sp metadata MUST have exactly matching entry or else 0 is returned.

return URL for accessing the service or 0 upon failure

Limitation: If binding is not specified, it may be ambiguous what binding the returned URL relates to. Generally the decision will have been taken prior to calling this function.

Source file: `zxidloc.c`

1.3.253 *zxid_sp_loc_raw()*

Raw computation of SP URL given service type, binding, and whether operation is a request. See *zxid_sp_loc()* for full description.

return URL for the protocol end point, or 0 on failure

Source file: `zxidloc.c`

1.3.254 *zxid_sp_meta()*

Generate our SP metadata and return it as a string.

Source file: `zxidmeta.c`

1.3.255 *zxid_sp_mni_redir()*

Change SPNameID (newnym supplied), or Terminate federation (newnym not supplied), using SAML2 HTTP redirect binding. This is the (SP) client side that contacts the IdP. Return the HTTP 302 redirect LOCATION header + CRLF2. Returns the URL as string to which the environment should cause the user (browser) to be redirected.

Source file: `zxidmni.c`

1.3.256 *zxid_sp_mni_soap()*

Change SPNameID (newnym supplied), or Terminate federation (newnym not supplied), using SAML2 SOAP binding. This is the (SP) client side that contacts the IdP.

Source file: `zxidmni.c`

1.3.257 *zxid_sp_slo_do()*

Process SP SLO request.

Source file: `zxidslo.c`

1.3.258 *zxid_sp_soap()*

Determine URL for SOAP binding to given service on SP and perform a SOAP call.

cf ZXID configuration object

cgi CGI variables that may influence determination of end point. Or 0 if no preference.

ses Session information that may influence the choice of the end point. The session MUST have assertion.

sp_meta Metadata for the Sp

svc_type The desired service, indicated as URN

body XML data structure for the SOAP call <Body> element payload

return XML data structure for Body element of the SOAP call response.

Source file: `zxidloc.c`

1.3.259 *zxid_sp_soap_dispatch()*

SOAP dispatch can also handle requests and responses received via artifact resolution. However only some combinations make sense. See `zxid/sg/wsf-soap11.sg` for the master SOAP dispatch from parsing perspective.

Return 0 for failure, otherwise some success code such as ZXID_SSO_OK

Source file: `zxidsp.c`

1.3.260 *zxid_sp_soap_parse()*

Return 0 for failure, otherwise some success code such as ZXID_SSO_OK

Source file: `zxidsp.c`

1.3.261 *zxid_sp_sso_desc()*

Generate SP SSO Descriptor metadata fragment [?].

Source file: `zxidmeta.c`

1.3.262 *zxid_sso_desc()*

Generate Single SignOn (SSO) Descriptor idp metadata fragment [?].

Source file: `zxidmeta.c`

1.3.263 *zxid_start_sso()*

Wrapper for *zxid_start_sso_url()*, used in CGI scripts.

Source file: `zxidssso.c`

1.3.264 *zxid_start_sso_location()*

Wrapper for *zxid_start_sso_url()*, used when Location header needs to be passed outside. return:: Location header as `zx_str`. Caller should eventually free this memory.

Source file: `zxidssso.c`

1.3.265 *zxid_url_set()*

Set the URL configuration variable. Usually you would use *zxid_parse_conf()* to manipulate this and some other options. This function exists for some special cases encountered in scripting language bindings.

Source file: `zxidconf.c`

1.3.266 *zxid_user_change_nameid()*

Change a NameID to newnym. Old NameID's user entry is rewritten to have `mniptr`

Source file: `zxiduser.c`

1.3.267 *zxid_version()*

Obtain the hex encoded version integer describing the libzxid. This can be used to effectuate a runtime version number check. For compile time you should check the value of the `ZXID_VERSION` macro.

Source file: `zxidlib.c`

1.3.268 *zxid_version_str()*

Obtain the version string describing the libzxid. This can be used for runtime version display. For compile time you should check the value of the `ZXID_VERSION` macro.

Source file: `zxidlib.c`

1.3.269 *zxid_write_ent_to_cache()*

Write metadata of an entity to the Circle of Trust (CoT) cache of the entity identified by `cf`. Mainly used by Auto-CoT.

Source file: `zxidmeta.c`

1.3.270 *zxid_wsp_decoratef()*

Create web service response, printf style. See *zxid_wsp_decorate()* for more documentation.

Source file: `zxidwsp.c`

1.3.271 *zxid_xacml_az_do()*

Process `<XACMLAuthzDecisionQuery>`. The response will have SAML assertion containing Authorization Decision Statement.

Source file: `zxidsp.c`

1.3.272 *zxlog_alloc_zbuf()*

Allocate memory for logging purpose. Generally memory allocation goes via *zx_alloc()* family of functions. However due to special requirements of cryptographically implemented logging, we maintain this special allocation function (which backends to *zx_alloc()*).

This function is considered internal. Do not use unless you know what you are doing.

Source file: `zxlog.c`

1.3.273 *zxlog_blob()*

Write a blob of content to log file according to `logflag` (see `zxidconf.h`). If the file already exists, i.e. there is a duplicate, the data is simply appended. When logging objects such as assertions, the duplicate check should be done as preprocessing step, see example below.

cf ZXID configuration object, used for memory allocation

logflag 0 if logging should not happen, 1 for normal logging, other values reserved

path Path where file is to be written, usually from *zxlog_path()*

blob The data to be logged.

lk Log key. Indicates which part of the program invoked the logging function.

return 0 if no log written (failure or logflag false), 1 if log written. Often ignored.

Example

```
logpath = zxlog_path(cf, issuer, a7n->ID, "rely/", "/a7n/", 1);
if (logpath) {
    if (zxlog_dup_check(cf, logpath, "SSO assertion")) {
        zxlog_blob(cf, cf->log_rely_a7n, logpath, zx_EASY_ENC_WO_sa_Assertion(cf->ctx, a7n), "E"
        goto erro;
    }
    zxlog_blob(cf, cf->log_rely_a7n, logpath, zx_EASY_ENC_WO_sa_Assertion(cf->ctx, a7n), "OK"
}
```

In the above example we determine the logpath and check for the duplicate and then log even if duplicate. The logic of this is that in case of duplicate, the audit trail captures both the original and the duplicate assertion (the logging is an append), which may have forensic value.

Source file: *zxlog.c*

1.3.274 *zxlog_dup_check()*

Check if file by path already exist. Since each uniquely ID'd object has unique path, mere existence of a file serves as duplicate ID check. This is used to satisfy some SAML processing rule requirements such as duplicate ID check for assertions.

cf ZXID configuration object, used for memory allocation

path Path where file is to be written, usually from *zxlog_path()*

logkey String that will help to identify reason of failure

return 0 if no duplicate (success), 1 if duplicate (failure)

Source file: *zxlog.c*

1.3.275 *zxlog_write_line()*

Write a line to a log, taking care of all formalities of locking and observing all special options for signing and encryption of the logs. Not usually called directly (but you can if you want to), this is the work horse behind *zxlog()*.

cf ZXID configuration object, used for memory allocation.

c_path Path to the log file, as C string

enclflags Encryption flags. See LOG_ERR or LOG_ACT configuration options in `zxidconf.h`

n length of log data

logbuf The data that should be logged

Source file: `zxlog.c`

1.3.276 `zxsig_data_rsa_sha1()`

Sign a blob of data using rsa-sha1 algorithm.

c ZX context. Used for memory allocation.

len Length of the raw data

data Raw data to sign

sig Result parameter. Raw binary signature data will be returned via this parameter.

priv_key Private key used for signing.

lk Log key. Used to make logs and error messages more meaningful.

return -1 on failure. Upon success the length of the raw signature data.

Source file: `zxsig.c`

1.3.277 `zxsig_verify_data_rsa_sha1()`

Verify a signature over a blob of data using rsa-sha1 algorithm.

len Length of the raw data

data Raw data to sign

siglen Length of the raw binary signature data

sig Raw binary signature data

cert Certificate used for signing

lk Log key. Used to make logs and error messages more meaningful

return ZX_SIG value. o (ZXSIG_OK) means success. Other values mean failure of some sort.

Source file: `zxsig.c`

Index

ap_die(), 1
ap_process_request_internal(), 1
ap_run_check_user_id(), 1
ap_send_error_response(), 1

bootstrap(), 41

chkuid(), 1, 18, 19
close_file(), 17
cron(), 34

dirconf(), 17

free(), 22

get_ent(), 44
gettimeofday(), 15

hexdump(), 17

keygen(), 50

malloc(), 10, 21, 22, 25, 27
memmem(), 23

name_from_path(), 17

open_fd_from_path(), 18

pool2apache(), 18

read(), 18
read_all(), 18
read_all_fd(), 18
read_post(), 18
reghk(), 19

SAML2bind, 44, 48, 49, 59
SAML2meta, 7, 30, 32, 43, 44, 50, 52, 60, 66
send_res(), 19
set_debug(), 19
set_zxid_conf(), 19
sha1_safe_base64(), 19
SIMPLE_BASE64_LEN(), 20
sizeof(), 35, 50, 52
snprintf(), 35
sprintf(), 2, 21
statement(), 36

strdup(), 8, 22
strlen(), 5, 10, 31, 36, 58

tag(), 16

unbase64_raw(), 20
URL_BAD(), 27
URL_DECODE(), 27

vasprintf(), 21
vname_from_path(), 20
vopen_fd_from_path(), 20

write2_or_append_lock_c_path(), 20
write_all_fd(), 20
write_all_path_fmt(), 21

yubikeymodhex_decode(), 22

zx_add_xmlns_if_not_seen(), 21
ZX_ALLOC(), 10, 21, 27
zx_alloc(), 21, 67
zx_alloc_sprintf(), 21
zx_alloc_vasprintf(), 21
zx_attr_lookup(), 27
zx_date_time_to_secs(), 21
zx_dup_cstr(), 22
zx_dup_len_str(), 22
zx_dup_str(), 22
zx_elem_lookup(), 27
zx_enc_seen(), 21
ZX_FREE(), 22
zx_free(), 22
zx_get_rsa_pub_from_cert(), 22, 25, 26
zx_hexdec(), 22
zx_init_ctx(), 7, 22
zx_is_ns_prefix(), 23
zx_len_so_common(), 23
zx_md5_crypt(), 23
zx_memmem(), 23
zx_new_len_str(), 23
zx_new_simple_elem(), 23
zx_prefix_seen_whine(), 23
zx_prepare_dec_ctx(), 24
zx_push_seen(), 24
zx_rand(), 24
zx_ref_len_simple_elem(), 24
zx_ref_len_str(), 24

zx_ref_simple_elem(), 24
zx_ref_str(), 25
zx_report_openssl_error(), 25
zx_reset_ctx(), 7, 25
zx_rsa_priv_dec(), 25
zx_rsa_priv_enc(), 25
zx_rsa_pub_dec(), 25
zx_rsa_pub_enc(), 26
zx_scan_xmlns(), 26
zx_str_conv(), 26
zx_str_ends_in(), 26
zx_str_free(), 26
zx_str_to_c(), 26
zx_strf(), 2
zx_tok_by_ns(), 27
zx_url_decode(), 27
zx_url_encode(), 27
zx_url_encode_len(), 27
zx_url_encode_raw(), 27
zx_xmlns_decl(), 27
zx_zalloc(), 27
zx_zlib_raw_deflate(), 28
zx_zlib_raw_inflate(), 28
zxenc_privkey_dec(), 25, 28
zxenc_pubkey_enc(), 26, 28
zxenc_symkey_dec(), 29
zxenc_symkey_enc(), 29
zxid_ac_desc(), 30
zxid_add_a7n_at_to_pool(), 30
zxid_add_at_values(), 30
zxid_add_attr_to_ses(), 30
zxid_add_fed_tok_to_epr(), 30
zxid_add_header_refs(), 31
zxid_add_ldif_attrs(), 31
zxid_add_qs_to_ses(), 2
zxid_alloc_ses(), 31
zxid_an_page_cf(), 31
zxid_anoint_a7n(), 31
zxid_anoint_sso_resp(), 31
zxid_ar_desc(), 32
zxid_as_call_ses(), 32, 41
zxid_az(), 2, 3, 8, 14, 32
zxid_az_cf(), 2, 32
zxid_az_cf_ses(), 2
zxid_cache_epr(), 32
zxid_call(), 3, 32
zxid_callf(), 32
zxid_cdc_check(), 32
zxid_cdc_read(), 32, 33
zxid_check_fed(), 4
zxid_chk_sig(), 4
zxid_conf_to_cf_len(), 33
zxid_curl_read_data(), 33
zxid_curl_write_data(), 33
zxid_date_time(), 33
zxid_dec_a7n(), 33
zxid_decode_redir_or_post(), 4, 9
zxid_decrypt_nameid(), 34
zxid_decrypt_newnym(), 34
zxid_del_ses(), 34
zxid_di_query(), 34, 38
zxid_epr_path(), 35
zxid_extract_body(), 35
zxid_extract_cert(), 22, 35
zxid_extract_issuer(), 35
zxid_extract_private_key(), 25, 36
zxid_fed_mgmt_cf(), 5
zxid_fetch_ses(), 5
zxid_find_at(), 36
zxid_find_attribute(), 36
zxid_find_cstr_list(), 37
zxid_find_epr(), 37
zxid_find_map(), 37
zxid_find_ses(), 37
zxid_fold_svc(), 38
zxid_gen_boots(), 38
zxid_gen_bootstraps(), 34
zxid_get_at(), 5
zxid_get_ent(), 38
zxid_get_ent_by_sha1_name(), 38
zxid_get_ent_by_succinct_id(), 38
zxid_get_ent_from_cache(), 38, 39
zxid_get_ent_from_file(), 39
zxid_get_ent_ss(), 5, 38
zxid_get_epr(), 5
zxid_get_epr_a7n(), 39
zxid_get_epr_address(), 39
zxid_get_epr_desc(), 39
zxid_get_epr_entid(), 39
zxid_get_meta(), 39, 40
zxid_get_meta_ss(), 40
zxid_get_ses(), 3, 40
zxid_get_ses_idp(), 40
zxid_get_ses_sso_a7n(), 40
zxid_get_sid_from_cookie(), 40
zxid_get_user_nameid(), 41
zxid_idp_as_do(), 32, 41
zxid_idp_dispatch(), 41
zxid_idp_list_cf_cgi(), 6
zxid_idp_loc(), 41, 42

zxid_idp_loc_raw(), 42
zxid_idp_select_zxstr_cf_cgi(), 6
zxid_idp_slo_do(), 42
zxid_idp_soap(), 42
zxid_idp_soap_dispatch(), 43
zxid_idp_soap_parse(), 43
zxid_idp_sso(), 6
zxid_idp_sso_desc(), 43
zxid_init_conf(), 6, 8
zxid_init_conf_ctx(), 43
zxid_ins_xacml_az_stmt(), 43
zxid_is_needed(), 43
zxid_issuer(), 43
zxid_key_desc(), 44
zxid_key_info(), 44
zxid_lazy_load_sign_cert_and_pkey(), 44
zxid_lecp_check(), 44
zxid_load_atsrc(), 44
zxid_load_cot_cache(), 44
zxid_load_cstr_list(), 45
zxid_load_map(), 45
zxid_load_need(), 45
zxid_localpdp(), 45
zxid_map_sec_mech(), 45
zxid_map_val(), 45
zxid_mk_a7n(), 45
zxid_mk_addr(), 46
zxid_mk_an_stmt(), 46
zxid_mk_art_deref(), 46
zxid_mk_attribute(), 46
zxid_mk_authn_req(), 46, 53
zxid_mk_az(), 46
zxid_mk_az_cd1(), 46
zxid_mk_dap_query(), 46
zxid_mk_dap_query_item(), 47
zxid_mk_dap_resquery(), 47
zxid_mk_dap_select(), 47
zxid_mk_dap_subscription(), 47
zxid_mk_dap_test_item(), 47
zxid_mk_dap_teststop(), 47
zxid_mk_di_query(), 47
zxid_mk_di_req_svc(), 47
zxid_mk_ecp_Request_hdr(), 48
zxid_mk_enc_a7n(), 48
zxid_mk_enc_id(), 48
zxid_mk_id(), 48
zxid_mk_idp_list(), 48
zxid_mk_logout(), 49
zxid_mk_logout_resp(), 49
zxid_mk_lu_Status(), 49
zxid_mk_mni(), 49
zxid_mk_mni_resp(), 49
zxid_mk_paos_Request_hdr(), 49
zxid_mk_saml_resp(), 49
zxid_mk_self_sig_cert(), 50
zxid_mk_Status(), 45
zxid_mk_subj(), 50
zxid_mk_transient_nid(), 50
zxid_mk_user_a7n_to_sp(), 7
zxid_mk_xacml_resp(), 50
zxid_mni_desc(), 50
zxid_mni_do(), 50, 51
zxid_mni_do_ss(), 51
zxid_my_cdc_url(), 51
zxid_my_entity_id(), 7, 13
zxid_my_issuer(), 43, 51
zxid_new_at(), 51
zxid_new_conf(), 3, 8, 14, 51
zxid_new_conf_to_cf(), 51
zxid_nice_sha1(), 52
zxid_OK(), 30
zxid_parse_cgi(), 7
zxid_parse_conf(), 52, 66
zxid_parse_conf_raw(), 8, 52
zxid_parse_meta(), 52
zxid_parse_mni(), 52
zxid_pep_az_soap(), 8
zxid_pick_sso_profile(), 53
zxid_pool_to_json(), 53
zxid_pool_to_ldif(), 9
zxid_pool_to_qs(), 53
zxid_process_keys(), 53
zxid_protocol_binding_map_saml2(), 53
zxid_put_ses(), 54
zxid_put_user(), 54
zxid_pw_authn(), 54
zxid_read_cert(), 22, 54
zxid_read_private_key(), 25, 54
zxid_saml2_map_authn_ctx(), 54
zxid_saml2_map_nid_fmt(), 55
zxid_saml2_map_protocol_binding(), 55
zxid_saml2_post_enc(), 9
zxid_saml2_redirect(), 55
zxid_saml2_redirect_enc(), 4, 9, 55, 56
zxid_saml2_redirect_url(), 55
zxid_saml2_resp_redirect(), 56
zxid_saml_ok(), 56
zxid_send_sp_meta(), 56
zxid_ses_to_json(), 56
zxid_ses_to_ldif(), 57

zxid_ses_to_pool(), 9, 10, 40
zxid_ses_to_qs(), 57
zxid_set_opt(), 57
zxid_set_opt_cstr(), 57
zxid_show_conf(), 57
zxid_show_cstr_list(), 57
zxid_show_map(), 57
zxid_show_need(), 58
zxid_sigres_map(), 58
zxid_simple(), 9, 58
zxid_simple_ab_pep(), 10
zxid_simple_cf(), 58, 59
zxid_simple_cf_ses(), 5, 10, 31
zxid_simple_idp_an_ok_do_rest(), 58
zxid_simple_idp_pw_authn(), 58, 59
zxid_simple_idp_show_an(), 59
zxid_simple_len(), 59
zxid_simple_no_ses_cf(), 59
zxid_simple_redir_page(), 59
zxid_simple_ses_active_cf(), 58, 59
zxid_simple_show_carm1(), 60
zxid_simple_show_conf(), 60
zxid_simple_show_idp_sel(), 60
zxid_simple_show_meta(), 60
zxid_simple_show_page(), 60
zxid_slo_desc(), 60
zxid_slo_resp_redir(), 60
zxid_snarf_eprs(), 61
zxid_snarf_eprs_from_ses(), 61
zxid_soap_call_body(), 61
zxid_soap_call_envelope(), 62
zxid_soap_call_hdr_body(), 62
zxid_soap_call_raw(), 11, 61, 62
zxid_soap_cgi_resp_body(), 62
zxid_sp_anon_finalize(), 63
zxid_sp_carm1(), 63
zxid_sp_deref_art(), 11
zxid_sp_dig_sso_a7n(), 63
zxid_sp_dispatch(), 63
zxid_sp_loc(), 64
zxid_sp_loc_raw(), 64
zxid_sp_meta(), 56, 64
zxid_sp_mni_redir(), 64
zxid_sp_mni_soap(), 65
zxid_sp_slo_do(), 65
zxid_sp_slo_redir(), 11
zxid_sp_slo_soap(), 12
zxid_sp_soap(), 65
zxid_sp_soap_dispatch(), 43, 65
zxid_sp_soap_parse(), 65
zxid_sp_sso_desc(), 66
zxid_sp_sso_finalize(), 12
zxid_sso_desc(), 66
zxid_start_sso(), 66
zxid_start_sso_location(), 66
zxid_start_sso_url(), 12, 66
zxid_url_set(), 66
zxid_user_change_nameid(), 66
zxid_validate_cond(), 12
zxid_version(), 66
zxid_version_str(), 67
zxid_write_ent_to_cache(), 67
zxid_wsc_call(), 3, 13, 14
zxid_wsf_decor(), 13
zxid_wsp_decorate(), 13, 67
zxid_wsp_decorateff(), 67
zxid_wsp_validate(), 14
zxid_xacml_az_do(), 67
zxlog(), 15, 68
zxlog_alloc_zbuf(), 67
zxlog_blob(), 67
zxlog_dup_check(), 68
zxlog_path(), 15, 68
zxlog_write_line(), 68
zxsig_data_rsa_shal(), 69
zxsig_sign(), 15
zxsig_validate(), 4, 16
zxsig_verify_data_rsa_shal(), 69