zvelo
We categorize the Web

zveloDB™ API

TABLE OF CONTENTS

DOCUMENT REVISIONS

Revision	Date	Initials	Description
2.1	3/15/2008	JD	Updates to include database updates.
2.1.1	4/30/2008	JD	Added url_engine_version, removing custom entries, detail about adding categories and url_debug.
2.1.2	5/19/2008	JD	Added url_categories_info. Extended url_init, documented zveloNET and how to implement direct zveloNET lookups.
2.1.2.1	5/29/2008	JD	Added category id to category_info struct.
2.1.2.2	6/16/2008	JD	Added url_deinit. Updated url_init.
2.1.2.3	7/9/2008	JD	Correctly documented port separator encodings.
2.1.2.4	7/17/2008	JD	Modified returned error codes in zveloNET cloud lookup to work around DNS that manipulate return codes.
2.1.2.5	9/5/2008	JD	Formatting change. Added NOTE to section 5.1 regarding checking cert.
2.1.3	10/23/2008	JAB	Added reference code section.
2.1.4	1/13/2009	JD	Updated perl example to strip all trailing slashes, not just one.
2.1.5	2/11/2009	JD	Show example of label separator.
2.1.6	3/1/2009	JD	Updated category names to be more descriptive.
2.1.7	4/6/2009	JD	Added encoding methods.
2.1.8	5/12/2009	JAB	
2.1.9	6/8/2009	JAB	Added url_read_custom_category_file API docs and updated url_reload docs.
2.4	7/22/2009	JAB	Added category mapping APIs.
3	12/1/2009	JAB	Format changes. Updates to url_remove_custom_url.
3.1	4/8/2011	JAB	Refer to the readme.txt file for details.

1. Introduction

Thank you for considering the zvelo offering. This guide is intended to be used by zvelo's OEM Partners who are integrating the zveloDB offering with their product. zvelo is committed to providing the best URL database accuracy, coverage and malicious website detection in the market.

2. zveloDB – URL Database

zveloDB consists of tens of millions of URLs classified into 141 unique categories. zvelo's automatic categorization systems and manual classification personnel update zveloDB continuously. The database is delivered through two deployment options. The first is the optional database stored locally (urldb) and is designed for fast database lookups. The urldb is updated daily and is available through both full and incremental updates. The second mechanism is through zveloNET. This optional service allows classifications of new/unseen URLs to be reported and classified on-the-fly as well as protection against new malicious websites between full database updates. Both mechanisms are covered in this document.

There are two options for zveloDB–Optimized and Full. The default option is the Optimized zveloDB, utilizing database size, URL popularity and blockable category URLs to provide a highly efficient and targeted database for on-disk deployment. The Full zveloDB downloads the entire database for local deployment and is typically utilized in service provider environments where extensive coverage and performance are key requirements. Please note that the Full zveloDB requires 64-bit systems.

3. zveloNET

zveloNET is zvelo's distributed network for the real-time detection, inspection, categorization and verification of ActiveWeb* sites collected from the global user community. Please refer to the Getting Started Guide for a detailed illustration of zveloNET.

**ActiveWeb – those sites visited by actual users*

4. zveloDB Categories

zveloDB consists of 141 different categories with the ability to add up to 10 custom categories. The built-in categories are defined in the zveloDB Categories document.

5. zveloDB Dependencies

The following sections describe the APIs needed to implement zveloDB on Linux, BSD and Windows. The zveloDB SDK depends on certain libraries and command line utilities. These must be installed on the system running the zveloDB SDK.

5.1. gzip

The full versions of the urldb database are gzip compressed. To uncompress these files the zveloDB SDK uses the gzip program.

5.2. xdelta

For applying the urldb incremental updates, the zveloDB SDK uses the xdelta program. Note that this is xdelta version 1, not version 3.

5.3. cURL

The zveloDB SDK uses cURL to download updates. The zveloDB SDK may use libcurl linked to libsitefilter, or it may use the independent cURL program. This depends on which type of zveloDB SDK an OEM Partner is using.

6. Using the zveloDB API

6.1. Setting the zveloDB Configuration Options

int url_option(const char *option, const char *value)
option - name of a zveloDB SDK option.
value - string with the value for the option.

int url_option_file(const char *fname)
fname - passing NULL will make it read .sitefilterrc in $HOME or in the current directory.

Option Name	Option Description
path	The directory path where the urldb files are stored. If this is set before calling url_init, the path argument to url_init may be NULL.
dia-host	The host name used in zveloNET DNS queries. If this is set before calling url_init, the dia_host argument may be NULL.
dia-serial	The zveloNET serial number. If this is set before calling url_init, the dia_serial argument may be NULL.
dia-timeout	The number of seconds for url_lookup to wait for each zveloNET query response. The default value is '0' which means url_lookup does not wait at all, but returns Uncategorized or categories inherited from the parent.
default-mapping	The name of the category mapping to use when returning category results.
update-url	The URL to be used when doing url_update. If this is set before calling url_update, the url argument may be NULL.
proxy-url	The URL of the proxy to be used when doing url_update.
proxy-user	The name of the proxy user when authentication is needed.
proxy-pass	The proxy password when authentication is needed.
proxy-method	The proxy authentication method.
debug-level	The debug level. May be set from 0 to 9.
update-info	A key and a value. Must be separated by one space, which means the key cannot have any spaces and the value may have as many as it likes. This option may be used multiple times which will add new keys or overwrite keys of the name name. The list of update-info keys and values will be appended to the url_update download URLs.
tools-path	An absolute path of the directory containing xdelta, curl and any other external programs needed by the zveloDB SDK. If this is not set then normal lookup rules are used.
category-file	The path to the categories.txt file. If it is an absolute path it may be in any directory. If it is just a filename it should be located in the urldb directory path.

ca-path	The path to the directory containing curl-ca-bundle.crt. If this is not set then the file should be in the DLL directory (on Windows) or the system ca-bundle files will be used.
dia-noresult	May be set to unknown (the default) or error. When zveloNET is used in asynchronous mode this controls what is returned when the SDK is awaiting a response from zveloNET. The error setting will make url_lookup return - URL_ERR_DIA_NORESULT instead of 0.
update-type	This controls which type of database is downloaded during url_update. It should be left unset or set to cdb1. cdb1 is the complete zvelo database and is much larger than the default database.

6.2. Initializing zveloDB

int url_init(const char * path_to_db_files, int verify_db, const char * dia_dns_host, char * serial_number)

path_to_db_files - absolute path containing the zveloDB and configuration files without trailing slash.

verify_db - specifies whether an OEM Partner wants the zveloDB API to verify the main database. 0 = no, 1= yes, 2 = extra verification. Verification takes additional time. It is always performed on customdb and diadb.

dia_dns_host - specifies the hostname used for the zveloNET real-time lookup interface. If NULL, zveloNET live lookup will not be used.

***NOTE**: You must store zveloNET key in file path_to_db_files/dia.key. dia.key will be automatically retrieved during url_update.*

serial_number - specifies the unit serial number used for licensing. If NULL, zveloNET live lookup will use standard lookup and licensing. This is used in per-device license enforcement of zveloNET lookups.

url_init returns 0 on success, non-zero on error.

***NOTE**: The database will be checked for expiration during url_init. The expiration time is 60 days. If url_update has not been used to download a new database in that time, the main database can no longer be used.*

6.3. Shutting Down zveloDB

int url_deinit(void)

url_deinit - returns 0 on success, non-zero on error.

6.4. Reloading zveloDB

int url_reload(void)

Returns 0 on success or non-zero on error.

Reinitialize the zveloDB API library and move any *.new files into place. Use this function after downloading updates or after manually placing new update files in place with *.new names.

The reload is thread-safe and affects all threads in the process. It will also be detected and url_reload will be done by other processes accessing the same database directory.

6.5. QUERYING ZVELODB

int url_lookup(const char * url, int result[5])

int url_lookup_match(const char * url, int result[5], char * match, size_t match_len)

int url_lookup_cache(const char * url, int result[5], char * match, size_t match_len)

int url_lookup_map(const char * url, int result[5], int map_id)

int url_lookup_match_map(const char * url, int result[5], char * match, size_t match_len, int map_id)

int url_lookup_cache_map(const char * url, int result[5], char * match, size_t match_len, int map_id)

The url_lookup function takes a pointer to the URL string and returns up to 5 result categories using the int[5] array pointer.

The url_lookup_match function also takes the pointer and length of a character buffer and returns the best match found in the URL database. For example, a lookup of:

www.zvelo.com/images/header/quote_header4.jpg

Returns zvelo.com in the match buffer. A leading dot on this string indicates that the URL database has more entries available for that URL path.

The url_lookup_cache function also takes the pointer and length of a character buffer and returns the best URL found for caching the result. For example, a lookup of:

www.zvelo.com/images/header/quote_header4.jpg

Returns zvelo.com in the match buffer. There is no leading dot because the URL database does not contain any entries for child URLs of zvelo.com. With intel.com, for example, zvelo's database also includes entries for intel.com/reseller and intel.com/jobs. A lookup of:

www.intel.com/reseller/images/hdr_mapreseller.gif

Results in a cache URL of intel.com/reseller. A lookup of:

www.intel.com/notexist/images/hdr_mapreseller.gif

Results in intel.com/notexist even though the /notexist path is not in the zveloDB. That extra path is added in order to avoid using .intel.com as the cache. The leading dot shows that intel.com cannot be relied on to categorize intel.com/reseller or any other child item. The cache items intel.com/reseller and intel.com/notexist have no leading dot and are reliable because there are no URL database child items of those URLs.

The url_lookup_map and url_lookup_match_map and url_lookup_cache_map functions also take a map_id argument which changes the mapping of the category IDs returned in the result argument. Use the url_get_mapping_id function to get a map_id value from a mapping name.

A positive number returned is the number of categories. A negative return indicates an error. A zero return indicates no categories.

6.6. Decoding Errors Received From the zvelo API

```
const char *url_errstr(int errorcode)
```

The url_errstr function takes an error code as returned from url_init or url_lookup and returns a character string with a short description of the error.

6.7. Retrieving the zveloDB Version

```
int url_db_version (void)
```

The url_db_version function returns the version of the initialized database. The function returns a positive integer representing the version of the database and a "-1" on error or if the database is uninitialized.

6.8. Retrieving the zveloDB SDK Version

```
int url_engine_version (char *version, size_t length)
```

The url_engine_version functions returns an integer representing the major and minor version of the engine. The major version is represented above one thousand and the minor version is represented below one thousands. For example, 2005 is major version 2 and minor version 005 or version 2.005.

version - string containing the version.

length - length of version string.

6.9. Choosing Proxy Authentication

```
enum url_update_proxy_auth_method {
        URL_PROXY_DEFAULT,
        URL_PROXY_BASIC,
        URL_PROXY_DIGEST,
        URL_PROXY_NTLM,
        URL_PROXY_ANY
};

int url_update_proxy_auth(const char *user, const char *pass, int method);
```

The url_update_proxy_auth function changes the proxy authentication used by cURL in the url_update process. The user name and password will be overridden by any user name and password provided in the proxy URL. NULL values or empty strings may be passed for both user and password.

The default authentication method first tries Basic, then Any if Basic fails. The Any method first tries the proxy without any authentication and examines the results, then uses the most secure of the proxy's

supported options. Older versions of the cURL library have bugs in their NTLM authentication. cURL 7.15 is known to work.

6.10. UPDATING ZVELODB

```
struct extra_info {
        const char *key;
        const char *value;
};

int url_update(const char *download_url, const char *proxy_url, int method, struct extra_info[])
```

extra_info - an array of struct. The last struct needs to have NULL in the key and value fields. It passes data in GET or POST parameters depending on method used for vendor tracking.

Optional parameter keys include:

- product_name (vendor)
- product_version (vendor)
- os
- user_count
- serial_number

And any other parameters an OEM Partner deems appropriate.

download_url - encodes username/password info, protocol, etc. download_url is the parent directory for the required files. Supported authentication methods: Auth-Basic.

proxy_url - can encode the username/password, host and path into url format. Leave unset or NULL if not needed.

method - GET=0, POST=1

***NOTE**: Requires url_init even if url_init returned error. Requires url_reload after download succeeds.*

6.11. MONITORING THE UPDATE PROCESS

```
struct url_status_info {
        time_t start_time;
        time_t estimated_time;
        double current_bytes;
        double total_bytes;
        time_t op_start_time;
        char *current_op;
        char *current_file;
        double current_file_bytes;
        double total_file_bytes;
        char buffer[128];
};
```

int url_update_status(struct url_status_info *info);

The url_update_status function fills in an information structure with data about the currently running update process.

This function can be called once every second or two to update a progress display. For an event driven interface, place a file update monitor on the file named "update- status.txt" in the database path.

All of the byte counts refer to bytes processed, not only downloaded. The update process has many steps. Some of the most time-consuming steps may be the incremental patching, so those get byte counts of their own.

The byte counts are in floating point to avoid 32-bit integer overflow. An incremental patch update of five or six patches may use byte counts of five gigabytes and more.

All of the strings are stored in buffer and pointers to the individual strings are provided for easy access.

start_time - the time when the update process started.

estimated_time - the approximate time the update will be finished. This estimate is not very accurate and can vary widely with changes in download speed and is especially variable when doing incremental updates.

current_bytes - the number of bytes already processed.

total_bytes - the number of bytes in the entire process. This is an estimate because the final uncompressed sizes of patched or downloaded databases is unknown.

op_start_time - the time when the current operation started.

current_op - a C string containing a short description of the current operation such as "downloading", "unzipping", "patching."

current_file - a C string with the name of the current file being operated on.

current_file_bytes - the number of bytes in the current file.

total_file_bytes - the expected number of bytes in the current file when it is complete. This is an estimate.

buffer - should not be read from, as it is the space used for storing string data.

6.12. Adding Custom Sites to zveloDB From Files

There are cases when users will want to add their own sites and classifications or will want to override classification from the main database. Adding custom sites allows users to perform both of these functions.

The first step to adding custom sites is to write those sites and categories to /var/lib/zveloDB/your_custom_list.urls in the following format:

url <TAB> cat_id,…,cat_id<LF> (up to 5 category ids – 3 standard and 2 custom)

Once the file is written, the zveloDB API can be instructed to load the file using the following API:

int url_read_custom_files(void)

url_read_custom_files - reads all files with *.urls extension from /var/lib/zveloDB to incorporate into the custom database.

/var/lib/zveloDB/custom.urls - used to store entries added with url_add_custom_url. Do not use this file name to write your own entries.

/var/lib/zveloDB/customdb - the binary database generated from the text files and used by the engine for fast queries of custom entries.

6.13. Adding Custom Sites to zveloDB One at a Time

int url_add_custom_url(const char * url, const int categories[5])

url - full url including, protocol, domain, port and path to be added to the custom list database.

categories - array of up to 5 categories (3 standard and 2 custom) to be associated with url. Use 0 for unused category array items.

/var/lib/zveloDB/custom.urls - used to store entries added with url_add_custom_url

***NOTE**: url_reload is not required.*

6.14. Adding Custom Site Lists

int url_read_custom_category_file(const char *file, const int categories[5])

file - the name of a file containing one URL per line. If the file name is not absolute (does not start with / or \ or X:/) then the file should be in the directory provided to url_init.

categories - array of up to 5 categories to be associated with the URL. Use 0 for unused category array items.

6.15. Removing Entries From Custom Site Files

To remove entries from custom sites, remove the entry from all of the *.urls files in path_to_db_files, such as custom.urls, and call url_read_custom_files.

6.16. Removing Custom Sites One at a Time

int url_remove_custom_url(const char *url)

url - full URL including protocol, domain, port and path to be removed from the custom URL database.

This will remove one entry from the binary custom site database but not from the *.urls files. This means that if the binary database becomes corrupt or if read_custom_files is called, the entry will be added back.

6.17. Associating Category Identifiers With Category Names

int url_categories_name(const int categories[5], const char *delimiter, size_t delim_length, const char *catnames, size_t *len_catnames)

categories - array of categories to convert to names.

delimiter - character(s) used to delimit category names in returned string.

delim_length - length of delimiter used above.

catnames - string that will contain the returned list of category names.

len_catnames - size of the catnames string buffer. It will be modified to contain the returned string size.

Category names are read from /var/lib/zveloDB/category.txt, which is in the following format:

Cat_id<TAB>category_name<TAB>category description<LF>

It returns 0 on success, non-zero on error.

6.18. Retrieving All Categories

```
struct category_info {
        int id;
        const char *name;
        const char *description;
};
char* url_categories_info(struct category_info **info, size_t *info_count)
```

info - pointer to pointer to array of category_info structures, allocated by the function. Pass it a pointer to a pointer in the calling function.

info_count - pointer to the number of category_info records.

Call this function and it will allocate the necessary memory and fill it with category_info records. When complete, call url_categories_info_free on the returned pointer.

6.19. Adding Custom Categories

Adding custom categories requires modifying /var/lib/zveloDB/categories.txt using the following format:

cat_id<TAB>category_name<TAB>category description<LF>

Custom cat_id must be between the numbers of 101 and 110.

***NOTE**: categories.txt must be in either ASCII , UTF8 or ISO 8859 format for url_categories_name to return a category name.*

6.20. Reporting Miscategorized URLs Through zveloNET

int url_dia_report(const char *url, int suggested_cats[3])

url - domain and path to be submitted.

suggested_cats - array of category id to be suggested (up to 3 standard categories).

Occasionally, end-users may disagree with a classification of a URL. In those cases, zvelo has supplied the means to allow users to submit those suggestions directly to zvelo through zveloNET. The reports are sent directly to manual classifiers to be reviewed.

Submissions do not guarantee change.

This requires the dia_dns_host parameter to be set and valid in url_init and it requires a valid zveloNET key.

If the OEM Partner has arranged with zvelo to use their own choice of host name for dia_dns_host then the submission data will appear to go to the vendor and not zvelo.

6.21. Enabling Debug Services

void url_debug(int level)

url_debug - used to set the level of debug message sent to STDERR or a custom output destination.

0 (default) is no messages sent to console.
1 is error messages only (recommended).
…
4 includes detail about the update process.
…
7 includes details about zveloNET lookups.
…
9 is all messages sent to console.

6.22. Changing Debug Output Destination

void url_debug_function(void (*f)(const char*, size_t size))

This function accepts the address of another function which takes a character string pointer and a length. That function may do whatever it likes with the debug information that will be sent to it. Debug information will usually be sent one line at a time with newlines included, but that is not guaranteed.

The buffer is not NULL terminated. Use the size parameter to determine where to stop reading from the buffer.

The default function writes debugging output to STDERR.

6.23. Using an Alternate Category Mapping

int url_get_mapping_id(const char *map_name)

This function returns a map_id value for use with url_lookup_map and url_lookup_match_map. If there is an error, it returns a negative error code.

6.24. Selecting Default Category Mapping

int url_default_mapping(const char *map_name)

This function sets the default category mapping to the chosen map. These maps are found in the database directory with names ending in .map. The map_name is the file name, not including .map. For example, the sitefilter-v2.map file provides the sitefilter-v2 mapping.

It returns either a negative error code or the previous default map_id. It can be called with a NULL pointer as map_name in order to get the current map_id without changing it.

The default category map affects all API functions that do not use a map_id argument. This includes zveloNET submissions and all custom URL functions.

6.25. Flushing the zveloCACHE

int url_dia_cache_flush(void)

This removes all of the URLs received through zveloNET and cached by the zveloDB library. This also happens whenever url_reload is called after a new urldb file is downloaded by url_update.

7. zveloNET for Direct Cloud Queries

zveloNET is the system used for delivering URL categorizations in real-time from the cloud. It is also used to collect uncategorized and miscategorized URLs and for automatically categorizing URLs.

The URL lookup component of zveloNET uses the DNS protocol to fetch the categories for a given URL. This allows the lookups to be made through firewalls, to use built-in caching across the Internet, and to use light-weight UDP packets for communications. The DNS lookups are protected by a rotating key as explained below.

Users of the zveloDB SDK do not need to implement or understand this protocol as it is incorporated into the zveloDB SDK and is automatic if enabled.

7.1. zveloNET DNS Rotating Secret Key Retrieval

OEMs will use this method to fetch the secret key and will provide this key to their devices.

https://subscriptions.esoft.com/api/GetRBLKey.php?OEMKey=<oemid>

OEMKey - a key provided by zvelo to OEM Partners for access to zveloNET. The key will rotate weekly and must be updated weekly on the devices. zveloNET will store the previous 3 weeks of keys to ensure clients that fail to retrieve a key will have time to re-request a valid key before service is disabled.

oemid - will be provided to OEM Partners by zvelo, Inc.

GetRBLKey.php - returns a string containing the rotating secret key or "Error: <reason>".

Check that the result is a number before using it as a zveloNET key.

***NOTE**: Be sure to enforce certificate validation checks on https calls to subscriptions.esoft.com. Man in the middle attacks could result in theft of your OEMKey.*

7.2. USING ZVELONET TO RETRIEVE CATEGORIES

URL categorizations are retrieved using a standard DNS TXT query. The query will return a list of category ids and the suggested URL for caching purposes if the URL is categorized, a domain not found error (NXDOMAIN) if the site is not yet categorized, or REFUSED if the system is not authorized to use the system. Uncategorized sites are pushed into the auto-classification system and can be categorized and ready for the next customer within minutes.

7.2.1. Building the Query

The URL being queried must first be encoded as described below. The general form of DNS query looks like this:

<Category>.<EncodedURL>.<AuthString>.<OEMPrefix-SerialNumber>.<Server>

Example: 0.www.domain.com_-.index.html.01EF.aa-50534.url.esoft.com

Where 01EF is the AuthString, url.esoft.com is the lookup domain (may be branded for different OEMs), aa is the OEM's assigned prefix, 50534 is the unit's serial number, and www.domain.com_-.index.html is the encoded URL.

Detailed TXT Record Format:

<Category> = use 0 if the category of the URL is unknown

To report a miscategorized URL, put the suggested category ID here. Use underscores to separate multiple categories (up to 3).

Example: 3_7_9.<EncodedURL>.<AuthString>.<OEMPrefix-SerialNumber>.<Server>

<EncodedURL> = normalized/encoded URL (See section Encoding URLs)

<AuthString> = this string is used to authenticate the request using a combination of the serial number, a one-way hash of the URL, and the secret key downloaded by the OEM Partner from zveloNET using the interface specified above. The calculation of the AuthString is described in more detail below.

<OEMPrefix-SerialNumber> = The OEM Partner prefix is a two-character string, assigned by zvelo and associated with a given OEM Partner. This is used to avoid serial number collisions between vendors. The serial number is the local serial number of the appliance. An OEM Partner may choose to use a single serial number or maintain unique serial numbers for each device. zvelo will compare the serial number to a list of allowed serial numbers supplied by the OEM Partner and return a DNS REFUSED error if the serial number is not authorized.

<server> = url.esoft.com or a branded OEM Partner's domain (example: dia.oem.com), which points to zveloNET's servers as the authoritative server for that subdomain.

7.2.2. Encoding URLs

To encode URLs, the following rules must be used. The failure to properly encode URLs will lead to inability to submit URLs to zveloNET.

***NOTE**: code examples in Perl and C++ that implement the steps below are included in the zveloNET SDK.*

1. Remove all trailing slashes from URL.
2. Remove all characters passed as arguments. This involves removing all characters after a question mark "?" character or a hash "#" character in a URL.
3. Take care of "_-" character sequences by converting "_-" to "_-_-".
4. The path separator slash "/" character should be replaced with the legal DNS character string underscore, dash, dot "_-.".
5. Port numbers separator colon ":" character should be replaced with the legal DNS character string underscore, dash, dash "_--".
6. Encode all illegal DNS characters into their URL percent encoded representation. For example, a tilde "~" character needs to be percent encoded to "%7e". The percent character must be converted into underscore dash format to be submitted to zveloNET as described in the next point.
7. Encode characters used for URL encoding. This is done by converting "%" to underscore, dash "_-".
8. DNS prohibits two empty labels (two dots next to each other). If the URL ends in a dot, it must be percent-encoded.
9. DNS requires labels (sections between dots) to be less than 64 characters. If you have a label that exceeds 64 characters, OEM Partners must split the label with "_-0". This requires OEM Partners to check at 60 bytes to insert this encoding.
10. The length of the entire query must not exceed 255 characters. If the length of the encoded category + url + serialnumber + server + 7 > 255, the encoded URL must be truncated.

AuthString:
The hex representation of a 16 bit CRC-CCITT hash of the following concatenated strings:

"domain" + "dia_key" + "OEMprefix-serialnumber"

***NOTE**: domain contains the subdomain and domain portion of the URL without any path for the purpose of caching CRC calculations for future requests.*

Return Values:
NXDOMAIN if accepted but not categorized

ANSWER (see below)

Answer Format:

When DNS responds with an ANSWER, it returns a TXT record with one of the following formats:

Category found:

categoryId_categoryId<TAB>cache url<LF>

That is an underscore-separated list of category id's followed optionally by a tab character and a cache URL. This cache URL is returned when a local database can cache the current URL using a given cache URL and can assume that all URLs under that base URL are the same category. In the case where a top-level domain is categorized, but subdomains or subpaths have different categorizations, the base URL is returned with a leading dot (".") indicating only that exact URL can be cached and URLs under it should be queried.

***NOTE**: In case of error, a negative value is returned for a category ID.*

Invalid AuthString:
FAILURE: Invalid authorization

Invalid SerialNumber:
REFUSED: Unlicensed serial number

***NOTE**: Additional lines returned are reserved for future use. Develop software to only use first line in response and ignore all additional lines.*

Example of transmission:

Retrieve secret key from zveloNET (the key only needs to be retrieved once per week):

https://subscriptions.esoft.com/api/GetRBLKey.php?OEMKey=<key>

Returns: 523008891

Assume a local OEM Partner prefix of "aa" and a local serial number of "12345".

Submit the following URL to zveloNET:

http://www.domain.com/Site/Resources/Check_something.asp?arg1=127.0.0.1

First, encode the URL:

www.domain.com_-.Site_-.Resources_-.Check_something.asp

Next, take the CRC-CCITT value of the concatenated string:

0x12AC = crcccitt(www.domain.com523008891aa-12345)

To query this URL, use a command like this:

dig 0.www.domain.com_-.Site_-.Resources_-.Check_something.asp.12AC.aa-12345.url.esoft.com

8. Direct zveloNET Query Sample Code

OEM Partners who choose to implement the zveloNET query method would still greatly benefit by the use of an in-memory cache. Repeating queries for the same URL wastes bandwidth and increases latency, both of which can be reduced with the use of in-memory cache.

The zveloNET SDK includes sample code that implements a memory-efficient cache intended for small devices.

8.1. Using the C++ Sample Code

Below is a description of each main section of the sample code. It covers zveloNET lookups, caching objects, cache memory limits, encoding and decoding and cleaning the cache.

***NOTE**: The sample software package includes a sample program named sample.cpp which shows in detail how the following is used.*

8.1.1. zveloNET Lookup and Cache Object

The first decision to be made is picking one of the following two classes: cached_dia_async or cached_dia_sync. Both of these classes inherit from cached_dia. The difference between the two options is how each class implements the lookup function. The async class does not wait on responses before returning a category value, while the sync class does.

Using the async class means that the category values returned for the first URL lookups are likely to be uncategorized. If uncategorized URLs are allowed through the system, the effect would be that a user could load parts of a malicious, pornography or gambling page before the blocking system receives the page categories. After that further access to that URL would be blocked.

The advantage in using the async class is that lookup does not slow down the system and does not require multiple processes or threading.

Using the sync class means that the lookup will wait for a response from zveloNET before continuing. All requests to zvelo's data centers will have a 5 second (configurable) timeout. If it expires, the lookup function will return a value of 0 or uncategorized.

In order to use either class, you will need values for zveloNET key (see section 8.1), serial number and DNS host (provided by zvelo).

Create an object from one of these classes near the beginning of your program. For example:

cached_dia_async dia(key, serial, host);

8.1.2. zveloCACHE Memory Limits

Choosing a memory limit is based on the constraints of the system. For example, on a gateway appliance with minimal memory, OEM Partners may elect to limit total memory to 1MB while on a PC with more available memory choosing a higher limit will provide performance gains.

To set memory bounds, set a limit value for the memory_limiter class. This tracks and limits how much memory the cache is using. This is accomplished with a customized allocator that is passed to the STL containers making up the cache.

Near the beginning of the program, set the limit value. For example:

```
memory_limiter::limit(1*1024*1024);
```

8.1.3. URL Lookup and Category Decoding

Use the cached_dia object to look up URL categories. The lookup is invoked like this:

```
uint32_t info = dia->lookup(url);
```

Then use the category_decoder class to pull out the individual categories:

```
int cats[url_cache::max_categories] = {0};
generate_n(
        cats,
        url_cache::max_categories,
        category_decoder(info)
);
```

8.1.4. Cleaning the zveloCACHE

If the cache fills up, the zveloNET class will automatically make room in the cache by removing the oldest entries.

zveloNET's master database is updated continuously as old category information is replaced with newer, more accurate information. Therefore, cache entries more than a day old should be removed to make room for updated category information. Clearing more often ensures more up-to-date categorization, however, expiring too soon results in performance penalties. zvelo has found expiring after one day to be a good compromise, but each implementation may have different goals.

To expire old cache entries call the clean_expired function periodically.

The sample program does it once per hour.

***NOTE**: Do not call the clean function every time through the processing loop because it is time consuming.*

8.1.5. Waiting for Something to Happen

It is a good idea to keep running the DNS event loop while waiting for things to happen. When there is an outstanding DNS request, there are retransmit and query expiration timers to manage. The cached_dia::lookup function will run these events. When running with cached_dia_async, there can be time between lookup calls when DNS events need to run.

8.1.6. POSIX

If programming in a POSIX environment, OEM Partners can use the dia::wait function. For example:

```
struct pollfd pfd = {0, POLLIN, 0};
while (dia.wait(&pfd, 1, 10) >= 0 && pfd.revents == 0)
        /* do nothing */;
```

The above code will wait for a 10 second timeout or an input event on file descriptor 0 (standard input) while executing outstanding DNS events. After all DNS requests are answered or expire, dia::wait returns -1.

Afterwards, go into a blocking read on standard input or a command socket to wait for more work.

8.1.7. Windows

Set up a timer to generate an idle event every one or two seconds. Call dia::wait from the idle event handler.

8.1.8. Data Structures

A linked list or binary tree uses pointers and too much RAM. An array takes too long to insert new entries. A deque is a collection of arrays that acts like one large array. It is a good choice to balance both insert time and space efficiency.

***NOTE**: The domain and path hash values will be 32-bit words as a default, and may be adjusted smaller for more space efficiency or larger for lower collision chances.*

8.1.9. Modifying the Code

The sample code is licensed for use in OEM Partner's applications. Modifying it or rewriting it in Java, C# or Perl is acceptable.

Some suggestions of areas you may want to look at first:

In dia::retrieve there is a hard-coded value of 1000 entries to clean from the cache when an insert throws bad_alloc.

In cached_dia_sync::lookup the timeout for a synchronous lookup is set to 5 seconds.

8.1.10. Adding Thread Safety

OEM Partners may want a reader/writer lock for the cache insert, clean and lookup functions. It should be safe for any number of threads to look up values in the cache, but they should lock against writes and deletes done by the insert and clean operations.

The zveloNET lookup functions write into the dia::outstanding collection and create new entries for lookup in DNS. These need protection by a mutex or critical section.

The tracking_pool_allocator class uses a boost::fast_pool_allocator with a null_mutex template argument. This might have to change for use in a threaded application.

8.1.11. Debugging

The code has some simple debugging macros defined in dlog.h. Change #if 0 to #if 1 and recompile to enable these.

8.2. Processing Overview

8.2.1. Main Loop

Check timer and clean expired cache values if needed.
Get a URL (from input or a proxy or browser plugin).
Perform a category lookup on the URL.
Output the category.

8.2.2. Lookup

Get a request via lookup function call.
If in async mode:
    Gather outstanding zveloNET responses into cache.
Clean expired cache values.
Check the cache for the URL, return data if found.
If in async mode:
    Check list of outstanding requests (minus request filename).
    If it is in the list: return uncategorized or cache hint.
    Send URL to zveloNET (section 6.4.7).
If not in async mode:
    Loop until timeout:
        Gather zveloNET responses.
If URL is found: return data.
If timed out: return uncategorized or cache hint.

8.2.3. Gathering Responses into Cache

Poll all outstanding DNS requests for results.

For each result:
    Encode the returned category information into an info value.
    If the recommended cache domain had a leading dot:
        Insert the original request URL with a set authority bit.
        Clear the authority bit on the info value.
    Insert the recommended cache domain and path with the info value.
    If the domain or path insertion failed because of low memory:
        Free cache memory by freeing older entries and redo the insertion.

Do the same work for all of the recommended cache returns. Look for leading dots to determine the authority bit.

8.2.4. Cleaning the Cache

Build a temporary binary tree (a sorted list) of pairs of domain, path indexes sorted by timestamp.
Build a set of indexes for deletion in order of age.
Delete indexes in order, adjusting path index value for each deletion, or domain index when no paths remain.

8.2.5. Cleaning the Expired URLs in the Cache

Scan all domain and path entries for timestamps older than the cutoff time.
Delete expired path entries.
If all path entries deleted: delete domain entry.

8.2.6. Checking the Cache for URLs

Loop on the domain, remove another subdomain on each loop:
Hash the domain.
Binary search the domain deque for the hash value.
When domain hash is found, loop on the path, remove another path component on each loop:
Hash the path.
Binary search the domain's path deque for the hash value.
When the path is found, return the information value, which contains the categories and the authority bit.

8.2.7. Sending URLs to zveloNET

Create a new string with the last part (usually a file name) trimmed off.
Check this trimmed URL against the list of outstanding zveloNET requests.
If found in the list, do not send and return immediately.
Create a new zveloNET request using DNS.
Add the URL (minus file name) and DNS request object to the list of outstanding zveloNET requests.