Encryption

NetYCE uses encryption in several places of the application:

- User authentication passwords
- Application Login forms
- Distribution files
- Customer database backups (archives)
- Selected columns in the database

User passwords

User passwords (local accounts) are not encrypted but hashed. The difference is that the hashing process is one-way, the user password cannot be regenerated from the hash.

Currently the user password (and the password history as well) are stored in hashed format using the MD5 algorithm. We do not hash the user password by itself, but is concatenated with the userid and a 'secret' realm-string before hashing. The format for the concatenation is `userid:realm:userpassword`. The realm is the string “CRC” by default and is defined in the database (Lookup.Variable = 'Realm').

The purpose of the concatenation and realm usage is that the customer can define its own secret-string (the realm) and that a pre-md5-hashed password-dictionary approach will not work.

A future implementation of the user-password storage will replace the MD5 algorithm with a stronger SHA512 hash, but will continue to use the concatenation and realm usage. The impact of requiring all users to redefine their password and losing the password history prevented us from giving it a high priority.

Login

The login form accepts a user password for validation. This password may not be sent 'over-the-line' in clear-text. The original solution was to calculate the MD5 hash locally at the browser prior to transmission. To do so safely, a negotiation system was constructed in which a `sessionid` was generated between browser and server as the basis for MD5 generation.

Later, when LDAP support was added, the user password in clear-text was required at the server for authentication with the LDAP server. The MD5 hash could not provide this so an additional step was created in which the user password got encrypted with the 'sessionid' at browser side and so could be decrypted at the server. The algorithm for this scheme is based on `Tea_JS`, the “Tiny Encryption Algorithm”, that is available in Perl and JavaScript.

Unfortunately, The current implementation lacks the TEA encryption due to an oversight, but is will be reinstated with the next release as part of the addition of Active Directory support.

Database archives

The NetYCE database from any customer contains all relevant networking information, including device accounts and passwords, enable strings, routing passwords and practically all ip-ranges and addresses. Sufficient to prevent this information to become public. Therefore, the downloadable NetYCE database archives are always stored encrypted.

The encryption key used for this purpose is taken from the NetYCE license file `/opt/yce/etc/yce_license` and is unique for the customer. To retrieve this key the command `/opt/yce/system/ck_license -y` will extract it from the license file. When restoring a database, the decryption is automatic, but failing this, the use is prompted to provide this key. the key for the NetYCE demo and download systems is `39BC-2124-E8AB-40D2-0B08`.

NetYCE database archives are created and restored using the front-end tool 'DB archives'. Also a Manual database restore procedure is available should the situation demand it. This procedure deals with the decryption of the archive in detail.

Distribution files

NetYCE distribution files are encrypted using a Triple-DES algorithm. This is to achieve two purposes. First the software consists of many scripted source files and only a few compiled daemons. To prevent our software to show up on some free internet share, encryption is used.

Second, the NetYCE software uses a licensing schema to selectively install the appropriate software. By using encryption, only those packages licensed by a customer can be decrypted and installed. This way we can create a single distribution file containing all packages without risk.

Column encryption

Design

Selected database columns are stored encrypted. These columns will be decrypted automatically when using the application (front-end and back-end) but are inaccessible using other means.

These encrypt and decrypt functions will have to be available to different components of NetYCE: front-end, patch-files, db-restore, key-update-tool, back-end. The back-end includes the configuration-generator, supporting tools of the job-tools (e.g. command job), custom reporting and the various daemons.

Due to this diversity, we choose not to use the standard encryption functions build in the MySQL / MariaDB databases. This choice safeguards us also from the risk of MySQL or MariaDB support or vulnerabilities or switching to other databases.

So we created our own `yencrypt()` and `ydecrypt` routines and make them available in `Common.pm`. These routines will use the file `/opt/yce/etc/crypt.keys` to retrieve the key.

The encryption algorithm of choice is AES256. This algorithm requires both a 'key' and an 'initialization vector' (IV). Research learned that to change the IV per table or even per row (using the row keys as IV) is unmaintainable when such row-keys might change at the same time as the encryption 'key'. Therefore we use a salted approach where a random salt is used to calculate the IV

and hash the passphrase for its encryption.

The key file, `/opt/yce/etc/crypt.keys`, holds one key per line using the format <key-id> <key-string>. The <key-id> is an hex number of two characters (00-ff) where the highest id forms the current active key. The key-id minus one the previous key and so on. By maintaining a history of keys, a 'forgotten' re-cryption (decrypt with old, encrypt with new key) will always be decodable. The default encryption <key> is extracted from the user's license file. Its values are similar to `39BC-2124-E8AB-40D2-0B08`.

The database archives (backups) include the list of encryption keys found in `crypt.keys` in its manifest file. When a database is restored, the `crypt.keys` file belonging to that database gets rebuild as well. For obvious reasons the encryption keys stored in the active manifest are encrypted.

> A blank <key-string> after a valid <key_id> indicates no encryption should take place, any existing encryptions will be updated to clear-text.

To identify which key was used to encrypt a given value, the key-id is included in the encrypted value. Also, to detect if a value string was already encrypted or not, the encrypted string will include a checksum. A string not matching a valid key-id and has an invalid checksum must therefore be a true value requiring encryption.

Table modifications were required to store the encrypted value of the column since they are generally much longer. Encrypted values will NOT be stored binary but base64 encoded. This to allow convenient SQL (debug) logging and reliable export and import functions.

The stored format for the encrypted string therefore consists of a concatenation of:

```
<key-id><encrypted-base64-encoded-string><checksum>
```

The <key-id> can be '00 thru 'ff',
the <encrypted> string is base64 encoded for storing and in- and export purposes,
and the <checksum> is calculated as a two-byte value, stored in 4-byte hex.

Practically ALL database access is handled by the `qdata()` function in `Common.pm`. This routine is well positioned to make on-the-fly decryption possible. It uses a data structure where Column names are available (as generated by the SQL SELECT statement).

This imposes limitations on the SQL SELECT statement the programmer may use:

- encrypted column names must be unique - the col name identifies the value to be decrypted
- encrypted column names must not be renamed in the SQL using AS or functions altering the SQL col name returned.

Further:

- the list of column names to de decrypted must be manageable (not to long to become slow)
- the list of column names to de decrypted must be externally defined, not hardcoded.

As for the non-select SQL statements INSERT and UPDATE, the on-the-fly encryption is much harder since is must rely on SQL parsing to deduce the column name. Initial research code demonstrated that this parsing is too unreliable for production purposes. Alternatively, modifying existing SQL statements and code to perform the encryption requires key-management in all parts of the

application.

Instead, after we realized that the encryption does not have to take place immediately with the INSERT or UPDATE, a simpler solution presented itself: The task of (ensuring) the selected columns are stored properly encrypted is performed by a background process, the `yce_skulker`. It will periodically (every 5 minutes) examine all tables with encrypted columns for update-timestamps that are more recent that the last examination. By reviewing the values of these selected rows, the non-encrypted values can be encrypted.

Care must be taken that in database-replication setups (using master-master) only ONE of the masters may execute this background process. If both were to modify the same fields simultaneously, race-conditions could cause a (brief) replication loop.

Another consequence of using the Update-timestamp() feature of the table is that the encryption record update triggers an update of the timestamp. Although this would result in a no-update at the second run, this trigger consumes resources. By setting the update-timestamp with the original value the trigger can be cleared.

To identify what tables and columns to examine for unencrypted values, an external definition file is required. The XML-formatted file `/opt/yce/etc/crypt_cols.xml` serves that purpose. This xml file can be modified by the customer to include or exclude encrypted fields. The file `/opt/yce/system/crypt_cols.xml` will be copied to `/opt/yce/etc` in case it is missing.

The default crypt_cols.xml file:

```
<crypt_columns>
  <table name="Domain" db="YCE">
    <column name="Default_enable_secret" />
    <column name="Snmp_comm_ro" />
    <column name="Snmp_comm_ro_upd" />
    <column name="Snmp_comm_rw" />
    <column name="Snmp_comm_rw_upd" />
    <column name="Snmp_trap_community" />
    <column name="Tacacs_key" />
    <column name="If_ser_chap_passwd" />
    <column name="Rme_passwd" />
    <column name="Rme_passwd_upd" />
    <column name="Local_passwd" />
    <column name="Local_passwd_upd" />
    <column name="Maint_passwd" />
    <column name="Fallback_ppp_passwd" />
    <column name="Fallback_telnet_passwd" />
    <column name="Isdn_passwd" />
    <column name="Rtg_bgp_nb_passwd" />
    <column name="Rtg_ospf_mdk" />
    <column name="Hsrp_passwd" />
    <column name="Mpls_fingerprint" />
  </table>
  <table name="SiteRouter" db="YCE">
    <column name="Enable_secret" />
  </table>
```

```
  <table name="Par_vals" db="YCE">
    <column name="Var_value" />
  </table>
</crypt_columns>
```

The reference to the table `Par_vals` is included to indicate custom variables of the PASSWORD type are encrypted. It is handled as a special case to prevent all custom parameter values from being encrypted.

Two entries in the Lookup are used for the encryption administration. First there is the variable `Crypt_update`. It has the latest timestamp (in epoch format) when the yce_skulker examined any database rows for encryption. To prevent several skulkers the Str_value has the server name of the active skulker that is performing the encryption. An operator is allowed to reset this timestamp to '0' to force the yce_skulker into examining all rows in the tables. The server name acts as a lock for other skulkers. If no activity for two hours, the lock is broken and another skulker may take over.

The second lookup variable is 'Crypt_id' which indicates the current active key-id, the key-id used to 'encrypt' (the decryption key-id is included in the stored value). Normally this value should not be modified.

Procedures

Remove encryption

To remove the column encryption of all fields currently in `Crypt_cols.xml` these steps must be executed in sequence:

1. Edit the `crypt.keys` file and append a new line with the next key-id but no key-value. So, for a file with one entry with key-id `00`, add a new line with key-id '01'. Remember that the key-id uses a HEX format.
2. Kill the `yce_skulker` process (using the front-end system tool or from the command line). The process is restarted automatically within 20 seconds, then the skulker will remove the encryption.

Add new encrypted column

Edit the `Crypt_cols.xml` in `/opt/yce/etc` to include the extra column. If the Table already is defined, append it to the column list, otherwise create both the table and column definition.

If the `Crypt_cols.xml` file gets corrupted or should be defaulted, it is sufficient to delete it. The file will be automatically recreated using the `Crypt_cols.xml` found in `opt/yce/system`.

Remove an encrypted column

It is NOT sufficient to remove a column from the `Crypt_cold.xml`. The existing encryption should first be removed, then redefined and applied. The following steps should be executed:

1. Edit the `crypt.keys` file and append a new line with the next key-id but no key-value. So, for a file with one entry with key-id `00`, add a new line with key-id `01`. Remember that the key-id uses a two-digit HEX format.
2. Kill the `yce_skulker` process (using the front-end system tool or from the command line). The process is restarted automatically within 20 seconds, then the skulker will remove the encryption.
3. Modify the `Crypt_cols.xml` file to remove the intended column(s).
4. Edit the `crypt.keys` file again and remove the blank key-id line.
5. Open in the front-end the 'Admin - Lookup' form. Select the `Internal` class.
6. Locate the `Crypt_id` variable and modify the 'String' value to the desired key-id. Click Apply
7. Locate the `Crypt_update` variable and modify the 'Num' value to '0'. Modify the String value to the local server if needed. Click Apply
8. Kill the `yce_skulker` process again (using the front-end system tool or from the command line). The process is restarted automatically within 20 seconds, then the skulker will re-apply the (new) key.

Changing the active encryption key

The default encryption key is derived from the customer's unique license file. This value will not change for that customer unless NetYCE issued a license file to a different contract owner name. To allow customers to change the encryption key manually, new keys can be added to the `crypt.keys` file.

The procedure to realise the activation of a new key:

1. Edit the `crypt.keys` file and append a new line with the next key-id but no key-value. So, for a file with one entry with key-id `00`, add a new line with key-id `01`. Remember that the key-id uses a two-digit HEX format.
2. Kill the `yce_skulker` process (using the front-end system tool or from the command line). The process is restarted automatically within 20 seconds, then the skulker will decrypt with the old and re-encrypt with the new key.