*March 2017*

**Objective**

Implement use of Salted Password Hashing model for ANet passwords.

In simple terms, the requirements for generating a hashed password are:

1. Generate a long random salt using Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). Using a salt value in the hash protects against rainbow or dictionary attacks.
2. Use a standard password hashing function that supports a “key stretching” algorithm to generate the hashed value from a clear text password. Using an iteration count for key stretching helps protect against brute force attempts to hack password values.
3. It has been suggested that the resulting hash value should then be encrypted with a key.
4. Store the key stretching iteration count, the generated salt and the encrypted hashed password value in the DB.

Requirements for verifying that the user logon credentials match the stored password value:

1. Fetch the salt and hashed password values from the DB or cache
2. Hash the logon password (clear text) with the saved key stretching iteration and salt values, using the same password hashing function.
3. Compare that hashed value with the hashed password value from the DB. If they match then the user provided password is correct.

**ANet Passwords**

The following ANet passwords will use the new salted hash model:

1. System User (staff) password
2. Customer password (includes “temp” password generated for “forgot password” functionality)
3. Customer “touchtone” password (“password2”)
4. Pending Customer password
5. Team password
6. Optional “Activation password” for online fees

**Other ANet Encrypted Values**

ANet will continue to use RC4 encryption model for the following values:

1. ID values in emails, etc. (e.g. customer id, permit\_id, membership\_id, etc.). These values are encrypted based on OWASP security check (<http://jira.active.com/jira/browse/AN-20680>).
2. SSN
3. Customer Security Questions / Answers
4. Portal logon token
5. AMS Merchant name/password
6. FTP password for scheduled customer import. Cannot use salted hash value since we need to be able to decrypt this value to pass to FTP.
7. FTP password for scheduled “pass/fail” import. Cannot use salted hash value since we need to be able to decrypt this value to pass to FTP.
8. FTP password for scheduled report downloads. Cannot use salted hash value since we need to be able to decrypt this value to pass to FTP. Note: This ftp password isn’t currently encrypted in the DB. It should be!

**Servlet Implementation Details**

* Use Java package [java.security.SecureRandom](http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html) (CSPRNG) to generate the salt value. Per security recommendations that long salt values be used, the salt value will be 256 bits (32 bytes).
* A new salt value will be generated any time a password is created or modified. Salt values won’t be reused.
* The default ANet key stretching iteration count will default to <TBD>, but will be configurable in SDI.INI and subject to change based on performance testing. Target goal will be for password validation to take 1 second or less. If password validation takes too long the configured iteration count for the salted hash will be reduced. The higher the iteration count, the more secure the password will be.
* The preliminary plan is to use PBKDF2 to generate the hash values. But there are also Java implementations available for Argon2 or Bcrypt so one of those could be used instead if preferable.
* Hashing will be done by the servlet not the client or SQL Server.
* If it is necessary to encrypt the generated hash value with a key, then the hashed value will be encrypted in the same way that the existing strings are encrypted in ANet.
* The key stretching iteration count, the generated salt value AND the resulting hashed representation of the password will all be stored in the DB. Research indicates that the salt value is not considered to be a “secret” so storing it in the DB is acceptable. But, if necessary, a further level of security can be achieved by storing the salt value externally.
* Support will be removed for the current backdoor which allows successful one-time logon if a password in the DB is set to a clear text (unencrypted) value. The first time the user logs on with that password, if the clear text password provided by the user matches the DB password (unencrypted) then the servlet will encrypt the value and write it back to the DB. I don’t really know the history about why this is supported so we need to reach out to GTO/Support to confirm it won’t be a problem.
* The backdoor to allow a clear text password in the DB will optionally still be supported for non-production systems (DEV/QA) if a new SystemInfo option is enabled.
* Currently the password value in imported customer records is always (?) in clear text. A new feature was added for ANE-34725 to implement a new adminEncryptPasswords.sdi function that is meant to be executed by Consulting after an import for a new site to encrypt all clear text passwords in the DB. But I believe scheduled customer load will also load customer records with a clear text password. The adminEncryptPasswords.sdi implementation should be adapted to the new salted hash password method and should also be refined to execute periodically (hourly?) as a background thread to check for and hash any new clear text passwords.

**DB Schema Modifications  
  
Option #1:**

* Add new HASHED\_PASSWORDS table with columns: HASH, SALT, KEY\_STRETCH\_ITERATIONS
* Add new HASHED\_PASSWORD\_ID column to all tables that currently have an password column
* The old password columns will still exist in the old tables (e.g. customer, systemuser, etc.), since ANet doesn’t typically drop old columns, but as existing password values are converted to a hashed password, the data in the old password column will be cleared.
* I probably favor this approach unless there are any objections.

**Option #2:**

* Add new SALT and KEY\_STRETCH\_ITERATIONS columns to all existing tables that currently have a password column
* Increase size of all existing varchar password columns to accommodate maxed hashed password length (TBD)

**Migration Plan**

Existing password values need to be replaced with the hashed version of the same password. This should be transparent to the customer or user. The ANet servlet will handle converting the password values. The following is the plan for replacing the passwords:

* Password values maintained in ANet cached data will be updated using the ANet DBUpdateScript model. On startup, if the org DB “level” indicates that passwords haven’t been hashed yet then the password values will be hashed and updated in the DB as part of org initialization. The update happens before cached data is loaded from the DB by the normal initialization process. Passwords updated by this method will be:
  + System User (staff) password
  + Team password
  + Optional “Activation password” for online fees

DBUpdateScript will go through all the above records. The current encrypted password will be decrypted to get the clear text password value. Then a new salt will be generated and the decrypted password will be hashed and the salt and hashed password written to the DB.

* Updating customer passwords will be handled differently, for several reasons: Customer records aren’t maintained in cached data; there are far more customer records in an org DB than there are system users; and finally, customer passwords are currently encrypted using a slight variation on the encryption method used to encrypt other strings in ANet (i.e. Encryption.encryptString vs. Encryption.encryptPassword). Currently there isn’t any support for decrypting a customer password back to its clear text value.

So customer passwords will be updated using the ANet RaiseDBLevelThread mechanism. RaiseDBLevelThread runs as a background thread to update DB data as needed. This thread doesn’t block org initialization, it’s assumed that RaiseDBLevelThread updates aren’t required to be finished in order for the site to function.

For the customer password updates, the thread will work through all records with an unhashed password (i.e. absence of salt value). For each one:

* Try to decrypt it. A new method will be implemented with algorithm to iterate through possibilities given the differences between the Encryption.encryptString and Encryption.encryptPassword. Each possibility will be decrypted using Encryption.decryptString and the result encrypted again with Encryption.encryptPassword value to check if the two values are a match.
* If correct decrypted value is found then generate the salt and hash the value, storing salt and hashed password to DB.
* In the event that the customer password can’t be decrypted, or the customer logs on before their password has been hashed, then the login method will verify the customer existing encrypted password (current method), and then generate the new hashed password given the user provided clear text password and write it to the DB.
* When a customer password is converted to the new hashed password value, the update event will also be logged in the CUSTOMERLOG table (e.g. “Login password hash generated”). Note: The actual old and/or new password is \*not\* saved in the log table.

Note about RaiseDBLevelThread: There are a handful of production orgs which aren’t up to the current DB data level for some reason. Bug tickets ANE-43057 and ANE-43058 need to be fixed to resolve this issue before the customer password update can be executed.

select \* from systeminfo with (nolock) where keyword='background\_db\_level' and convert(int,(convert(varchar(max),keywordvalue)))<26

**Tasks**

1. Implement core functionality (2SP)
   * Generating salt
   * Generate hash password
   * DB Schema and stored proc modifications
2. System User passwords (2SP)

* DBUpdateScript code to convert System User passwords
* Password maintenance for AUI functions: New System User, Update System User, Change System User Password (logged on user)
* Password validation for logon
* Remove support for logon where password in DB is clear text (unless non-PROD system and new SystemInfo option is enabled).
* Implement logic on password change to check that 5 most recent passwords aren’t being reused. For old recent passwords, if not converted to new format, will need to decrypt the old password and then hash it with the same salt as the new password to check if it’s a match.

1. Customer password (3SP)

* Includes customer and pending customers.
* RaiseDBLevelThread code to convert customer passwords
* Password maintenance for AUI and CUI functions to create new customer accounts and/or update an existing customer password.
* Password validation for CUI customer logon
* Update logon to convert old password to hashed password if it hasn’t been updated yet

1. Team password (3SP)

* DBUpdateScript code to convert Team passwords
* Password maintenance for CUI and AUI user functions to create and manage team passwords.
* Update CUI to handle team “logon” with hashed password

1. Activation password for online fees (3SP)
   * DBUpdateScript code to convert values
   * Password maintenance for AUI functions for new fee record (activity fee, DC program fee, standard charges) or updates.
   * Update CUI validation of activation password entered by customer

**KIT**

CUSTOMERS  
Existing column: PASSWORD1 VARCHAR(100)  
Existing Column: PASSWORD2 VARCHAR(100)

PENDINGCUSTOMERS  
Existing Column: PASSWORD1 VARCHAR(100)  
Existing Column: PASSWORD2 VARCHAR(100)

TEAMS  
Existing Column: PASSWORD VARCHAR(100)

SYSTEM\_USERS  
Existing Column: PASSWORD VARCHAR(300)

SYSTEM\_USER\_PASSWORDS  
*This table contains all “recent” passwords for system users, used to validate that a new System User password isn’t a duplicate of the last 5 passwords for that user. When a system user password is updated, the current hashed password, salt, and key stretch iteration count will be inserted to this table.*  
Existing Column: PASSWORD VARCHAR(300)

ACTIVITY\_FEES  
Existing Column: ACTIVATION\_PASSWORD VARCHAR(100)

DCPROGRAMFEES  
Existing Column: ACTIVATION\_PASSWORD VARCHAR(100)

STANDARDCHARGES   
Existing Column: ACTIVATION\_PASSWORD VARCHAR(100)