SAPI Passwords security

Background

SAPI – Secure and Private Internet – Passwords is a project to help you create and use high quality passwords over a range of services, online and local. The project is opensource and written in javascript for the online / mulit-platform version and in C# for a .net standalone application version. Both versions follow the description in this document.

Main security features

SAPI-PW main features include:

- SAPI-PW lets you add accounts, userids and create secure passwords for all your sites and webservices
- You can later retrieve the passwords when you need them to login to a service
- All sites will get strong unique passwords
- You can't store your current passwords, instead you are forced to create new passwords for all your sites, all these passwords will be unique and very strong.
- backup and print functionality to let you secure access to your passwords
- Includes both an online and offline version of the product, se below
- The passwords to your sites will never be stored, not even locally on the standalone version. They will be calculated every time you need one

Online mode

If you use the Online version then:

- you can store site-information on the server so you can access it anywhere, anytime.
- UserID and Password for the SAPI login will never be sent in clear text to the server. They always undergo a destructive nonreversible hashing operation before being sent.
- The online version stores site information and Userid's centrally so that you always can retrieve a current version of your accounts. Site-Passwords are never sent to the server.
- All information that are sent to the server are first encrypted locally with a key only the user controls. The server can never decrypt data that is stored there.

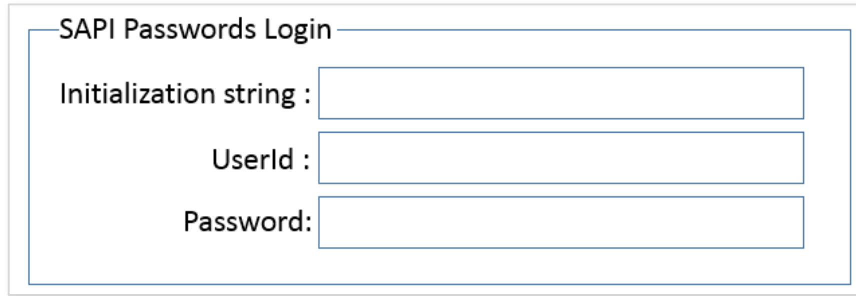
Offline version

If you use the offline version

- The offline version never sends data to a server. You can run it standalone. In the offline version you are more secure and independent, but you have to handle synchronization between computers yourself. The offline version does not work on iPhone, iPads.
- Then site-information will be stored locally, encrypted.
- You have to backup data and sync it between computers yourself

Usecase description

Initialization and Login



Before starting to use the service you have to initialize the product and login.

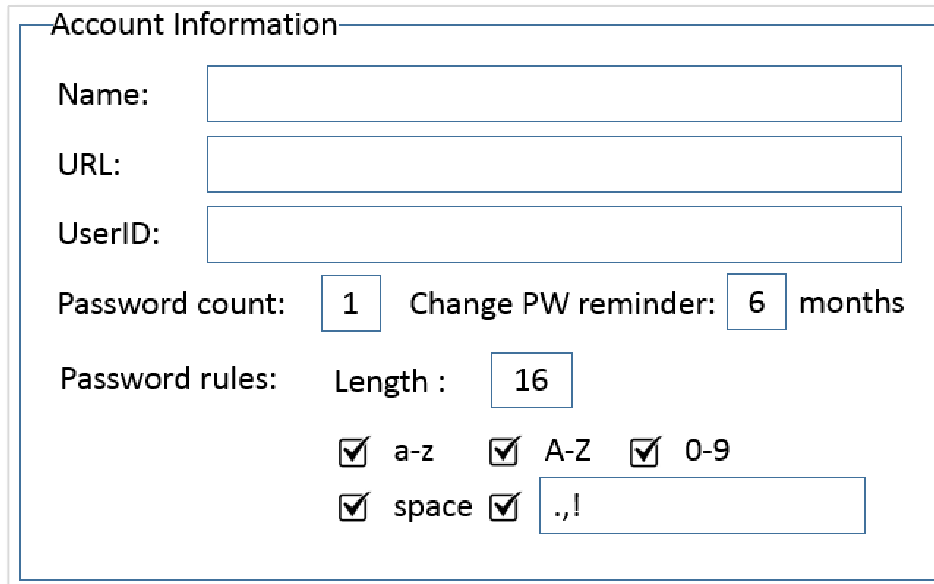
Initialization string: First time (or after a full reset of your system/browser) you have to enter an initialization string. Think of this as a long complicated password that you'll never remember. You only have to enter it once, and it is there to make it hard to guess /brute force your password. OBSERVER you cannot edit, delete characters, paste a value or go back to add characters to this field. You have to type it in one sequence from start to finish.

Confirm initialization string: type it again to make sure you got it right.

UserID: The UserID has to be unique and is used to differentiate you from other users on your computer or on our server.

Password: Every time you log in you have to give your password before you get access to your passwords. You can set different timeout values on how often you have to give the password.

Adding an account:



When you add an account for a site or service you need to supply the following information:

Account information : all the information (but the password) about an account. This information consists of:

Name: You can name the account whatever you wish. This name is listed in the overview. You can call one "My Appstore" i.e. It should be descriptive. The name has to be unique and can never be changed once you created an account.

URL: This is the URL to a webservice or a local URI to a local service. The link should lead to the loginpage of the service. You can change this later of the loginURL to the service has changed.

UserID: The UserID you have for the service. This should be the exact name you use to login to the service. It can never be changed after you entered it to an account.

Password Count: This starts at 1, and increases by 1 every time you create a new password for a service.

Change PW reminder: You can set how often you should be reminded to change the password for this site. Set this to zero if you never want to be reminded.

Password rules: The password rules defines what kind of password the site accepts. You should always aim to reach the maximum complexity of passwords for each site. The product does know about some sites and tries to offer the best rule for these sites. For unknown sites we aim to find a compromise that should work for most sites. You can define:

Password Length: The length of the password you create for the site. Should be at least 20 characters if the site allows it. Longer is better.

Allowed characters: Next follows a list of allowed characters. You check those that are allowed by the site. At the end you have a box where you can enter special characters allowed by the site.

Security

The following chapter describes the security of the SAPI-PW application.

About Hashes and Salts

Before anything is hashed in Sapi-PW then the original data is always salted. Hashes with different purpose will all get a different and unique salt. The bitlength of the salt matches the size of the hash (usually SHA512). The names of the individual salts are listed below. If you want to compile your own version of this application then you should change all these salts.
The reason for salting everything is to:

- Add entropy to weak in-data such as passwords or initialization-strings
- Resist attacks with rainbow tables
- Provide the ability to create your own version that uses your own unique salts to create a high entropy unique version for you.
- Every hash we use does not require salting, but if you do it everywhere you won't miss the important hashes to salt (and it does not hurt)

Initialization and Login

Local Initialization

1. Entering Initialization string:
   As you type each character of the Initialization string, they will be caught, salted and hashed with SHA512. The first character will be salted with ***SapiInitStringSalt***. The following characters will be salted with the previous characters hash before they are hashed. The repeated hashing of additional characters and their salt is stored in the ***SapiInitStringHash***
   The character that is written to the Initialization string field on the webpage is a random character just to force display of a '*' character to give you feedback that you have entered a character.

This means that the Initialization string is never stored in memory, or in a text field. A keylogger can catch the string by recording the characters as they are entered, but a Trojan that tried to read memory variables or html-fields won't catch the string. Also, the initialization string is only entered once and that also lowers the risk of it being caught in a keylogger.

2. When the entire Initialization string has been entered and treated according to 1, we use HMAC-SHA512, PBKDF2 a unique salt and 20.000 repetitions:
   PBKDF2 (HMAC_SHA512, ***SapiInitStringHash***, ***SapiInitPbkdf2Salt,*** 20000, 512 bit)
   This gives us a 512 bit key : ***SapiInitStringKey***
3. The UserID (***SapiUserID***) is stored in memory in plain text (and in the html-field)
4. The Password is handled exactly the same as the initialization string in step 1, but using a unique salt: ***SapiPasswordSalt*** storing the result in ***SapiPasswordHash***.
5. To get a key for local login and to base coming encryption keys on, we use
   PBKDF2 (HMAC_SHA512, ***SapiPasswordHash*** , ***SapiLocalPasswordPbkdf2Salt,*** 20000, 512 bit)
   giving us the ***SapiLocalPasswordKey***. This key is now stored in memory as long as the session lasts (configurable by the user, max one day or until the browser restarts)
6. Now we encrypt the ***SapiInitStringKey*** for local storage. We use 256 bit AES, deriving the key from
   PBKDF2 (HMAC_SHA512, ***SapiLocalPasswordKey*** || ***SapiUserID***, ***SapiInitAESSalt,*** 2000, 256 bit)
7. The encrypted ***SapiInitStringKey*** is stored locally in the browser. Now it is erased from memory.

Local Login

When the user has initialized his app he only has to login in the future. The UserID will be stored so the user don't have to enter this. He just have to give his password so we can decrypt the locally stored data. The steps from initialization is repeated except for the Initialization string steps that are omitted for later logins.

Online Registration

Skriv något här.

Online version Login

You always have to perform a local login before you can do an online login. These steps are only performed for an online login.

1. When the ***SapiUserID*** is entered it is salted (with UserIDSalt) and hashed with SHA512, resulting in ***SapiUserIDHash***
2. To get a key for online login we use PBKDF2 function again as :
   PBKDF2 (HMAC_SHA512, ***SapiPasswordHash*** || ***SapiUserIDHash***, ***SapiOnlinePasswordPbkdf2Salt,*** 20000, 512 bit)
   using a unique salt to get the 512bit ***SapiOnlinePasswordKey***
   By adding the server unique UserIDHash to the online key derivation function we ensure that all onlinepasswordkeys on the server will be unique, even if several users picked the same actual password. We want to keep information leakage to the server to a minimum.
3. Connection to the server is using
   a. Server SSL cert from : .... Issued to :
   b. 2048 bit SSL cert
   c. TLS 1.1 or TLS 1.2 protocol
   d. TLS_RSA_WITH_AES_256_CBC_SHA (0x35) encryption
   e. Secure renegotiation
   f. Strict Transport Security with a 5 year max age
   g. Headers that protect against XSF, XSRF and clickjacking

4. Serverlogin is performed using ***SapiUserIDHash*** as the UserID and the ***SapiOnlinePasswordKey*** as the password. The server will therefore never see the real userID or password, but only securely hashed versions of them.
5. The ***SapiUserIDHash*** is used to figure out which user to login and to make sure that all users are unique. We felt that it was unnecessary to expose the users real ID to the server, when a hashed version works just fine.
6. When received by the server, the ***SapiOnlinePasswordKey*** is salted and hashed with SHA512 and matched with the stored hashed OnlinePasswordKey for the user.
7. If login is successful, the server will return ***SapiAccountInfoAESBlob*** to the application. See below how this was created. If the AccountInfo is different from the locally stored version, the application will create a backup of the local version and then replace it with the new fetched from the server.

Saving account information

The user is logged in and now he enters account information into the app. As these are entered they will be stored locally, and if the user so chooses, on the server.

Encryption of Account Information:

1. The account information for all the accounts is stored in a long json string, containing all site data for all the accounts. This string does NOT contain the passwords for the sites. The string includes the following information for the accounts: This string is called the ***SapiAccountInfoJsonString***
   a. Account name (***AccountName***)
   b. Account URL
   c. Account UserID (***AccountUserID***)
   d. Account Password Count (***PasswordCount***)
   e. Account Change Password Reminder
   f. Password length
   g. Password rules (a-z, A-Z, 0-9, space, special characters)
2. This string is encrypted with 256bit AES using the key: ***SapiLocalAccountInfoAESKey***. This key is derived using :
   PBKDF2 (HMAC_SHA512, ***SapiInitStringKey*** || ***SapiLocalPasswordKey*** || ***SapiUserID***, ***SapiAccountInfoAESSalt,*** 2000, 256 bit)
   The encrypted account info is called ***SapiAccountInfoAESBlob.*** It is stored locally in the browser.
3. If you are running the application in online mode, this string is also sent to the server and stored there for the user's convenience.

Generating passwords:

The SapiPW application only supports generating passwords. It won't accept old user choosen passwords for encryption and storage. The reason for this is twofold:

- Most users choose crappy passwords and it's probably best to change them anyway
- If we accept user chosen passwords, we are forced to encrypt and store the passwords, locally and on the server. Even if we use strong encryption it's harder to trust such a solution. In SapiPW the passwords are never stored anywhere. Instead they are generated on the fly when needed.

1. User chooses an account he want to log in to from the account-list on the main page.
2. The user may click on the URL-link. This will open a browser in a new window with the URL to the servers as stored in the Account Info.
3. The user may copy/paste the UserID to use for this service.
4. When it's time to enter the password, the user clicks on "copy to clipboard". This triggers the password generation process:

a. First we create a password random number generator seed (***PrngSeed***)by using :
   PBKDF2 (HMAC_SHA512, ***SapiInitStringKey*** || ***SapiLocalPasswordKey*** || ***SapiUserID*** || ***AccountName*** || ***AccountUserID*** || ***AccountPasswordCount***, ***SapiAccountInfoAESSalt,*** 2000, 256 bit)
b. Next we generate an array of valid password characters ***ValidChars*** for this account based on the password rules.
c. We initialize a Prng function with the ***PrngSeed*** seed. We are using a home built prng (***Sha3Prng***) built on sha3, se below. The ***Sha3Prng*** gives us the same sequence of random numbers every time it is fed an identical seed.
d. We now generate the first password character by
   i. Use ***Sha3Prng*** to create a random number between 0 - (length of ***ValidChars*** -1)
   ii. Use the random number to pull a character from ***ValidChars***
e. Repeat from d.i until we have enough characters.
f. Place the result in the clipboard and tell the user he can paste it into the password login field.
g. Wait 12 seconds before clearing the clipboard.

Sha3Prng how does it work

1. The ***Sha3Prng*** is initialized with a seed. This seed is salted with a fixed 512 bit salt (***ShaPrngSalt***) and then hashed using Sha3 resulting in ***RandomArray.*** Sha3 will produce high quality random numbers based on the seed (see literature)
2. When we need a new random number in the range : ***range*** we will
   a. Divide the full span of a 32 bit unsigned into ***range*** number of equally sized buckets, placed after one another. This gives us buckets of size : Floor(MaxUint32 / ***range***) (***BucketSize***)
   b. The buckets will seldom reach to the max value of a 32 bit word (unless the reminder of a division between a MaxUint32 and range is zero).
   c. Our random values from will be equally distributed over the 32 bits, and thus they will also be equally distributed over the buckets.
   d. Take the first 32 bit word from the ***RandomArray.*** This word is called ***RandomWord***
   e. Now we check that ***RandomWord*** lands in one of the buckets. If it does, we calculate which bucket it lands in, and the bucket nr is our random number.
      RandomNumber = Floor(***RandomWord*** / ***BucketSize***).
   f. Next we advance to the next word in ***RandomArray*** and we are ready to serve a new random number

   g. In some cases the ***RandomWord*** is too large and misses all the buckets (is larger than ***BucketSize * range***). If that happens we have to throw that randomWord away and pick the next word from ***RandomArray.*** We continue from e with this new randomWord.
   h. When we run out of numbers in ***BucketSize*** we have to get a new random value, We do this by hashing (***RandomArray*** || ***ShaPrngSalt***) and continuing from d.