TrustedSec
INFORMATION SECURITY MADE SIMPLE

SHIPS/2.0 Installation Manual

Shared Host Integrated Password System (SHIPS)

Prepared by: Geoff Walton - TrustedSec

Prepared for:
For Public Release

info@trustedsec.com ■ 14780 Pearl Rd. Suite 300 ■ Strongsville, OH 44136 ■ 877.550.4728

Table of Contents

1 SHIPS Summary

SHIPS is a solution to provide unique and rotated local super user or administrator passwords for environments where it is not possible or not appropriate to disable these local accounts. SHIPS also attempts to address secure sharing of these accounts when they must be controlled by multiple parties. Client systems may be configured to rotate passwords automatically. Stored passwords can be retrieved by desktop support personnel as required, or updated when a password has to be manually changed in the course of system maintenance. By having unique passwords on each machine and logging of password retrievals, security can be improved by making networks more resistant to lateral movement by attackers and enhancing the ability to attribute actions to individual persons.

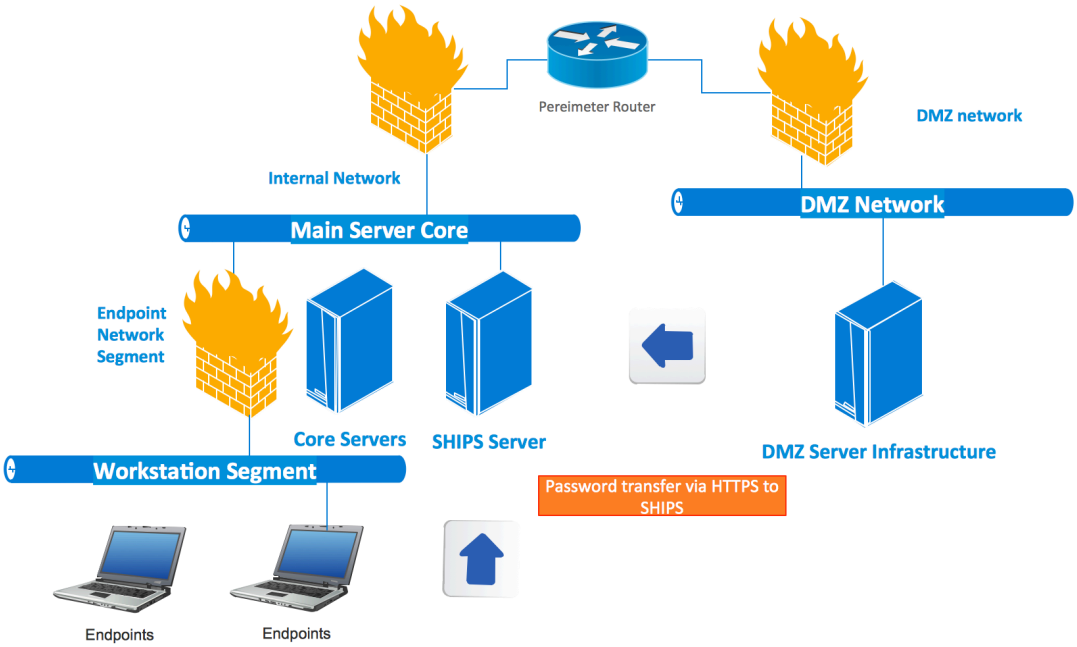
When performing penetration tests, our common attack vector is through compromising one host and pivoting to other systems with the information obtained. It is common to see large-scale breaches utilizing this method and that is where SHIPS comes into play.

SHIPS is designed to make post-exploitation more difficult and minimize what systems attackers gain access to. Once SHIPS is set up, there isn't much else that is needed and it's simple to integrate into existing business processes.

1.1 Project Goals:

- A complete solution packaged as a single application which can be deployed on a variety of platforms
- Deployments should be simple to move or relocate (this may be required in disaster recovery situations)
- Immediately useable with little or no training for support personnel
- Low resource consumption on server and clients
- Low impact on WANs
- Support a wide variety of clients
- Simple client protocol so various operating systems and devices can be integrated with the server through shell scripts and utilities such as cURL
- Simple to integrate with external directories or asset management tools
- Ability to easily script interaction with the server in order to facilitate system deployment processes, or integrate with other support tools

1.2 How it works:



A script is deployed to the endpoints, servers, and any other systems through group policy or similar deployment tools. The script is run on a determined timeframe from the organization. The script makes a password request to the server, which generates unique password string that it stores and transmits to the client. The client script then applies the new password on the client. TrustedSec recommends deploying SHIPS to servers, workstations, or any other windows-based systems. The passwords will now be unique per individual server and workstation.

For organizations where client and server support roles are segregated to different groups of employees, SHIPS now supports hierarchical folder structure with access control lists to manage such requirements. Additionally, two classes of storage are supported, plain test and RSA. When RSA storage is used the server uses public key encryption to store a copy of the record for each user with read permissions. This keeps the data at rest secure and ensures it can only be retrieved by a user in possession of their private key. This is appropriate for small numbers of assets and owners such as servers and third party services. Plain text storage is appropriate where assets and viewers are numerous and may need to retrieve records for locations where they may not have a private key present, such enterprise desktops and support personnel.

When users with permission to access account passwords wish to retrieve them, they

simply log into the SHIPS admin server and do a lookup of the machine name. The web application will display the current password associated with the device. SHIPS authorization can be tied to external systems such as LDAP or performed locally to SHIPS.

2 Architecture

Client Password Server is implemented as a standalone Ruby application. The application is web based, however, http handling is provided by WEBrick, an in-process webserver included in the standard library. This application has minimal external dependences, gems SQLite and optionally net-ldap. The net-ldap dependency is not required; if installation of net-ldap is a problem on your platform, LDAP integration will not be available if omitted. Keeping external dependencies minimized allows the application to be installed on a host that may have existing web services running and alongside applications that may use large frameworks such as RAILS without the need for a separate Ruby environment. It also helps ensure the application can be stood up quickly in a disaster recovery scenario.

Clients perform password updates using a single https request. The request and response are kept to a minimal size to facilitate large numbers of clients across WAN links while keeping impact and data utilization low. Remote clients can be supported as well as this request may be handled securely over the Internet and be passed by proxies if the client script implementation such as the VBS client provided for Windows clients supports it.

Support personnel access the information using a web interface. Functions such as creating, removing, and clearing passwords on computer objects may also be accessed via a web service call that does not require sessions or cookies for transactions with scripts.

2.1 Security considerations:

- The server can be configured to run as an un-privileged user.
- It is necessary to ensure the directory permissions of the server files are correct, for example, do not allow access to the database file by users who should not have access to the local account passwords. Guidance on file permissions is provided in the installation section.

- The server will validate the computer name exists in the database or external directory before issuing a password and storing the response. This provides basic protection from a DOS attack where the attacker would attempt to fill the password database.

- The server issues a nonce value to each machine along with the password, and requires the correct value be submitted with future password requests,

preventing attempts to corrupt the password database. This weakly authenticates the client. It may still be possible for an attacker to brute force the nonce making a large number of requests; this method was selected to keep client scripts simple while still providing some protection.

- When deploying the client script, the directory in which the script resides and the directory where the history file (stores suggested next update time, and nonce) is stored should be writeable only by groups who would otherwise have access to change the administrator password on the machine, Administrators. If file permissions allow a non-administrative user to alter the client script, this could create a local privilege escalation path; use care deploying client scripts.

- From a recovery standpoint, if the password database is ever lost or damaged it should automatically rebuild correctly as clients check in to update their passwords. The server will see the machine as a new client but valid in the directory and will issue a new password. The database may otherwise be backed up by simply copying the SQLite file.

3 Installation

1) Install basic platform requirements
    a. Obtain Ruby 2.0.0-p481 or later packages from your operation system vendor. Many platforms will already have a compatible Ruby installation.
    `ruby –v`
    will report the current version and exit if Ruby is installed.
    b. Install ruby sqlite3
    On most Linux and Unix platforms this can be accomplished with:
    `gem install sqlite3`
    Some documentation for installing SQLite3 and the Ruby gem for Windows is available here:
    http://stackoverflow.com/questions/15480381/how-do-i-install-sqlite3-for-ruby-on-windows
    c. Install Ruby Net/LDAP
    `gem install net-ldap`
    d. Remove existing USA and SQLiteORM gems if present
    e. Install SQLiteORM
    Gem install SQLiteORM-0.9.?.gem
    (You should have downloaded a SQLiteORM gem with the project)
    f. Install Ruby USA
    `gem install usa-0.9.?.gem`
    (You should have download a USA gem with the project)

2) Unpack the Application Archive
    a. Change to a directory where you wish to store the application, such as /opt
    b. `tar –xvf /path/to/archive.tar`
3) Create service and groups accounts or select an existing account such as "httpd" for the application. The service accounts primary group should be the group account you selected for use. Both are assumed to be "ships" in this document. Consult your operating system documentation for account setup.
4) Set permissions
    a. On most Linux/Unix platforms appropriate permissions should be:
        i. ships.ships 770 for directories
        ii. ships.ships 640 for files
    b. The exceptions to this are:
        i. SHIPS.rb ships.ships 750
        ii. SQLiteIdentity.rb ships.ships 750
    c. Note: if you change the location of the database or log files you may need

to modify the permissions on the new location accordingly

5) Generate and obtain an SSL certificate
   a. The application expects these files in pem format with no password. Your clients must trust this certificate.
   b. To create a new certificate:
      openssl genrsa –aes256 -out ssl.key.orig 2048

      openssl req -new -key server.key.orig -out ssl.csr
      Complete the prompts for the certificate fields.

      openssl rsa -in ssl.key.org -out ssl.key

      Present the file ssl.csr to a Certificate authority for signing, save the response as ssl.pem (You may need to convert to pem format if the response is delivered differently)

      When finished, the files ssl.pem and ssl.key should be in the ./etc directory. You may remove the other files.
6) Minimal configuration
   This will provide a basic server using local database authentication and no external computer validation directory; this is good starting configuration for environments where no external directory integration will be used, and it will allow clients to register themselves. Leaving the server in this configuration is not advised, as it is vulnerable to a denial-of-service attack.
   a. Open the file ./etc/conf - The start of this file contains a skeletal config commented out, along with a short description of the configuration options. The configuration file uses YAML grammar the # indicates a comment. Lines starting with # may be removed if desired.
   b. In the web: section
      i. Locate the line Port: 443 and modify the port number if required; if you are already running an TLS/SSL enabled web server, most likely you may need to use a different port
      ii. Locate the line daemonUser: and set the value to the user you selected in step 2
      iii. Locate the line serverName: Set this to the DNS name of the server as clients understand it; this must match the CN name set on the server certificate or one of the SAN names if you used an existing certificate.
7) Arrange for your platform to start the application, for example, by adding it to /etc/rc.d/rc.local on many Linux systems. The application should be started as the system super user. This is necessary so that it may bind ports and perform a call to su.

For many deployments, authoring a shutdown script may not be required as it is common for many Linux platforms to send a TERM signal to all processes as part of the shutdown procedure.
If you do author a shutdown of the script, it should simply send a TERM signal to the process. The process will not terminate immediately but may take several seconds to ensure all changes are committed to the database file.

8) If you have used identSQLite, for authentication, you will need to use the SQLiteIdentity utility to create at least an account for the super user and possible and ACLAuthors group. Before you can preform those steps it will be necessary to complete the data: section of the configuration file.

4 Server Configuration

The configuration file is located at ./etc/conf in the application root. This file uses YAML grammar, it has multiple sections, and each is in the form a hash, starting in the left most column. This file is case sensitive. Values are required unless otherwise noted.

The **web** section – defines a number of WWW server level properties. It has the following members:

1) port: A numeric value expressing TCP port number, usually 443
2) daemonUser: a string value naming the user account the service should become after binding ports
3) Log: a string value giving the path to the log file the application should use. This may be omitted and defaults to ./var/log/passwordserver.log
4) SSLCertificate: a string expressing the path to the SSL certificate; this may be omitted, and defaults to ./etc/ssl.pem
5) SSLPrivateKey: a string expressing the path to the SSL private key in pem format; this may be omitted defaulting to ./etc/ssl.key
6) serverName: This is the host name for the webserver. This does not need to be the system host name but must be the name clients use when connecting to the host; the host should be able to resolve this name to itself. This value is a string.

The **data** section – references external data files, the path to the SQLite database and the style sheet the web site should use

1) dataPath – options, ./var/data/SHIPS.sqlite is used when omitted, otherwise the specified file path.
2) styleSheet ./etc/style.css is used when omitted, otherwise the specified file path.

The **app** section – defines most application and service settings. It has following the members.

1) foreground: a boolean value, when true, prevents the application from detaching the console; this is useful for debugging with Ruby debugging tools such as pry. This may be omitted and defaults to false. It is advisable to start the SHIPS the first time with this enabled and after any other configuration changes to check for error messages during the early start up process which are written to the console.
2) sessionTimeout: a numeric value specifying the number of seconds a client session may be idle before it is logged off. This may be omitted; the default

value is 300 seconds.

3) superUserIdent – The identity class used to authenticate the application's super user
4) superUserToken – The unique id as understood by the specified identify class for the super user, as a string. If you have not yet created a SHIPS database and intend to use local authentication for the super user, set this to "1" – this will be the id of the first user created.
5) ACLAuthorsIdent – This is the identity class that will authenticate users allowed to author or modify ACLs within the application.
6) ACLAuthorsToken – This is the unique id as understood by the specified identity class for the desired group. If you have not yet created a SHIPS database and intend to use local authentication for ACLAuthors, set this to "1" – this will be the id of the group created.
7) allowedLoginIdents: an array of strings, naming one or more identity (ident) classes that may be used to authenticate users to the application. In some cases, single Ident class is all that is required, however, there are cases where multiple Ident backends may be useful; for example, you wish to have users authenticated against active directory using identLDAP, but you also want the root user on the local system to be able to authenticate with their password against the local SQLite database using IdentSQLite.

   The following Ident classes are provided with SHIPS:

   a. IdentSQLite provides authentication against a table of users in the applications SQLite file; these passwords are stored using a salted SHA hash.
   b. IdentLDAP provides authentication against a remote LDAP server, allows specifying an LDAP group membership requirement for authentication to succeed.

   It is possible to provide your own ident class, passed as an additional module for Ruby to load when executing the server. It would be specified here as well.

8) defaultLoginIdent: a string naming the Ident class to use when a user accesses a page that requires a session and they do not have a session, or when any other resource requires a login method and one is not provided in the request. This must be one of the specified login_idents. It should be the identity class most uses will login with, this makes password re-entry after a session timeout simple.
9) syslog: a boolean, when true, messages are written to syslog when passwords are stored or retrieved. This may be simply omitted to disable syslog.

The **devices** section defines rules for devices.

1) age: a numeric value specifying the suggested password rotate time in days
2) length: a numeric value specifying the number of characters for generated passwords; do not set a value lower then 7.

The **identityOptions and ValidatorOptions** sections provide options that will be passed to all identity classes and validator classes. There is a special identity class always present in SHIPS identDevice. There are two important properties:

1) indentDevice_default_folder: This is the default folder a device performing an automatic password change should have a document created in. This folder **must** have permission for the "device" group to modify, for creating new documents. This folder will be used by legacy SHIPS clients which do not specify a folder id.
2) identDevice_validators: This specifies a list of validator classes, as an array, for identDevice to use to determine if a corresponding device should be able to create a new document. This applies to devices which are not currently in the SHIPS database. SHIPS provides two such validator classes out of box, DeviceValidatorLDAP and DeviceValidatorAny. It is possibly to create your own device validators.

5 First Time Setup

If you plan to use IdentSQLite as your identity backend you will need to create some initial users and groups. It may be a good idea to use IdentSQLite for at least the superuser so that local authentication is possible in emergencies. If you don't plan to use identSQLite skip section 2.

1) Create a basic conf file by updating the sample ./etc/conf file. Make changes to suit your environment as described in chapter 4.
2) Create SQLiteUsers
    a. Verify you have completed the database section in ./etc/conf
    b. Add a super user (root)
        i. ./SQLiteIdentity.rb --user root --operation AddUser
        ii. You should see the SQLite database file created, the program will also output the initial password. Be sure to record this password.
    c. Add a group (root)
        i. ./SQLiteIdentity.rb --group root --operation AddGroup
    d. Add the user root to the group root
        i. ./SQLiteIdentity.rb --group root --operation AddUserToGroup --user root
    e. Create any additional users and groups desired. Additional groups and users can be added anytime in the future as well.
    f. Use ./SQLiteIdentity.rb –listUsers and ./SQLiteIdentity.rb –listGroups to display the unique tokens assigned.
3) Use your superuser and ACL authors configuration settings as needed.
4) Once satisfied with your configuration, run the server for the first time, it is highly recommended "foreground: True" is configured in the app section so you can easily see any startup problems.
    a. SHIPS is designed to not depend on specific identity and validator classes, when starting SHIPS, ruby must be told to pre-load these objects with the –r flag.
        i. ruby –r ./lib/identsqlite –r ./lib/identldap –r ./lib/devicevalidatorldap SHIPS.rb
        ii. The above start ships assuming that you wish to use IdentSQLite, IdentLDAP and DeviceValidatorLDAP. A simple configuration might only need to load a single identity class and no additional validators.
        iii. With the exceptions of the identity IdentDevice, and the validator DeviceValidatorAny, SHIPS does not automatically load identity and DeviceValidator classes. SHIPS provides some useful out of box classes
        iv. Idents
            1. ./lib/identsqlite.rb – IdentSQLite – local database authentication
            2. ./lib/identldap.rb – IdentLDAP – authentication against LDAP or AD
            3. ./lib/devicevalidatorldap.rb – DeviceValidatorLDAP – device lookup in an LDAP directory
            4. Additionally, the USA gem provides IdentETC which can authenticate against local password and shadow files on Linux systems, you may

need to recompile the shadowtest binary for your system.

5) Visit the site with a browser, you must use the HTTPS protocol, and login as the super user. Make sure to select the login method for the identity type you have configured for the super user.
    a. Create a private key for the super user, instructions should be provided in the application.
        i. ***Important*** SHIPS considers the super user to match all ACLS, that is the super user will be a reader on all documents. In the case of secure documents this means a copy will be generated for the super user if a keypair has been created. You can prevent this by skipping step 5a, however this may result in the potential to create documents within the application that cannot be read by anyone, or to create documents that cannot be recovered if users misplace private keys.
        ii. A good practice may be to store the super user's private key in a safe offline location.
    b. If you have used IdentSQLite it would be a good idea to change the password now; a link will appear at the left.
    c. Create an access control list (ACL) for devices, and click Manage ACLs at the left. Add a new ACL by clicking New. Name the ACL as desired. Add an access control entry (ACE). Choose the identity type "IdentDevice", make the ACE a group ACE, and choose the "devices" group, finally check the "Write Permission" box. It is not recommended to assign the "read" permission for devices. Add an additional ACEs for other users and groups; most likely you will wish to assign at least read permissions. The ACL can also be updated at a later time.
    d. Add a default folder for devices. Click "Browse Folders", the root folder has a special name ".", add a sub folder by clicking "Edit Folder" then click "New Subfolder". Assign the folder a desired name and assign the device ACL to the folder. Note the folder ID, you may wish to update your identDevice_default_folder value in your configuration.

6 Application Quick Start

Most functions are reached from /client, or by simply starting from /. This page allows users to perform their key management functions, as well as browse folders and documents. SHIPS provides basic on screen instructions for most operations. It's important to have users of secure documents create key pairs as soon as possible; they will not be able to retrieve documents, they have read permission to that are created before their key pair is established. If new users must be added later an existing user with read / write access can simply re-save the document.

After looking up a computer, users are presented with the current and previous three passwords, which may be useful in cases where the machine failed to adjust its password after issuing the server request. The current password may be modified, by overtyping it and then clicking the save button. If the machine is out of sync with the server, the nonce value may be returned to the "open" state by pressing "save". This will permit the computer to request a password update with any nonce value on the next request. The delete button can be used to remove a computer that is no longer in the environment from the database.

If a computer needs to set a password but cannot be validated, or if you are not using an external validator, it may be manually entered as a new document with the device flag applied.

/devicews
Provides a non-session method by which commands may be sent to the server. This allows for clients that need to transact using single requests and do not wish to store cookies. This interface is mainly designed to be used by scripts, but will function from a web browser. Use cases are, for example, a Microsoft Deployment Manager script adding a newly deployed computer to the database, or an asset management tool removing a retired computer from the database.

When issued a GET request, this resource will respond with a simple web form that provides information on how to create various POST based transactions. Only the POST method transacts; GET can be sent with a single query string parameter "method" specifying a desired Ident class. If the method is not specified, forms with fields corresponding to the authentication parameters that belong to the default Ident method are generated.

/password
This has been designed for maximum compatibility with proxies, for clients that may need to set a password via the Internet. **If you have reverse proxy or next-gen firewall in your environment, consider using it to restrict access to only GET requests to this resource from external clients.** Clients do not need to make any other requests to update their passwords.
This is the resource clients use to request password updates. The client sends a simple request via the GET method with the following parameters:

nonce – followed by any 32bit unsigned number as a decimal string if the computer has not previously set a password. Otherwise this is the value from the server's response in a previous request.

name – The unique name of the computer or device.

The server will respond with a minimalistic HTML page, for example:

```
<!DOCTYPE
html><html><body>true,KnBzSmYmWHg2P3ZWO0VnVyZxNDw=,1988738762,2014-11-27
17:59:32</body></html>
```

The response body consists of comma-separated fields:

a. Success – true if the request was accepted. The client should **not** modify the local password if this value is false, and should try the request again later.

b. Message – the base64-encoded password if the request was accepted, otherwise a failure message as a plain string.

c. nonce – the nonce to use with the next request

d. Suggested change time – the server's proposed time for the next update request to be sent. It is up to the client to respect this value, and to retry as required.

7 Deploying the Windows Client Script

Most environments will want to use some type of deployment management solution to propagate the client script to PCs. The following steps will allow manual deployment of the script. The VBS script provided should work on most Windows Clients win2k and above.

0) Create a directory such as C:\password on the Windows client. Ensure permissions on this directory allow only Administrators to have write or modify access.
1) Copy the file setAdminPass.vbs from the ./clients directory on the server to the directory on the client PC.
2) Open the SetAdminPass.vbs file and update two constants at the top of the script.
   QUERYSTRING="https://myserver.example.com/password?"
   HISTORYFILE="c:\password\password_history.txt"
3) Open a command prompt. This must be done with the right-click "run as administrator" method on Windows 7 or Windows Server 2008 or later versions.
4) Run the command:
   schtasks.exe /rl HIGHEST /ru "SYSTEM" /Create /SC HOURLY /TN "Update Administrator Password" /TR "c:\windows\system32\cscript c:\password\SetAdminpass.vbs"

This will call the client script every hour. The first thing the script does on startup is read the history file. If the change time has not been reached, the script exists silently. The file is not updated if the script fails to reach the server. This will enable the client to retry every hour until successful. There will be WSH entries in the Application event log that will indicate if the password was updated or where in the process the update failed.

8 Implementing Validators or Ident Classes

If you wish to implement your own Validator or Identity integration, it's a good idea to look at Identlap.rb and identSQLite as examples. Note that with custom idents you will also need to create a corresponding directory class. This should be sourced from the file implementing your identClass. Similarly, for validators look at devicevalidatorany.rb.

Start SHIPS using your custom classes by having ruby preload the class files. Although you may need to reference them in the conf file, SHIPS will otherwise discover objects implementing the abstract Identity, Directory, and DeviceValidator classes.

ruby –r MyCustomIdent.rb –r MyCustomValidator.rb SHIPS.rb

Sample Ident Class:

```
require 'usa'
require_relative 'DirectorySample' #A file with DirectorySample

Class IdentSample < WEBrick::USA::User::Identity
      self.directory = DirectorySample

      def initialize(optional={})
            super optional
            #other constructor code here
            #after calling super @optional[:symbol_name]
            #will allow access to identityOptions values from the config
      end

      def login(form_data)
      #form data will be a Hash will the input values from
self.form_inner_htmls
      @usertoken = 'SampleUser' #unique user id
      @username = 'SampleUser' #Display name, can be equal to @usertoken
      return false unless form_data['password' == 'foo']
      #return false on failed login
      self #return self on login successful
      end

#A class function that presents a simple HTML login form, add whatever fields
#you need.
      def self.form_inner_html
    <<WWWFORM
       <table><tbody>
       <tr>
       <td>User Name:</td>
       <td><input class="PlainTextIn" maxlength=50 name="username"
```

```
type="text"></td>
       </tr>
       <tr>
       <td>Password:</td>
       <td><input class="PasswordIn" maxlength=255 name="password"
type="password" autocomplete="off"></td>
       </tr>
       </tbody></table>
WWWFORM
  end
end
```

A Sample Directory Class
All instances of an identity share a single instance of a directory.

```
require 'usa'

class DirectorySQLite < WEBrick::USA::User::Directory

  def get_groups
    groups = Array.new
    groups << 'Everyone'
    groups << 'Power Users'
    groups # An array of all groups
  end

  def get_idents_groups(identity)
    groups = Array.new
    groups << 'Power Users' if identity.usertoken == 'Bob'
    groups #The groups this identity belongs to as an array
  end

  def get_idents
    idents = Array.new
    idents << sefl.identity.new.impersonate('Bob', 'Bob')
    idents #An array of all users in the directory as identities
    #Note impersonate means this identity instance is not the result of a
#   login
  end

  def get_groups_by_name(name)
    groups = Array.new
    #Code to convert a name to an array of groups that have that display name
    groups
  end

  def get_group_by_token(token)
    group = lookup_function_to_find_a_group_by_id_or_token
    return WEBrick::USA::User::Group.new(group.groupname, group.id) if group
    nil #return nil if not matched
  end
```

```
  def get_user_name(token)
    return 'Bob' #the display name for database token.
  end

  def get_idents_by_name(name)
    idents = Array.new
    #Code to populate this array with identities for users with the
#    display name name.  Identities should impersonate
#    that is be instanced as
#    self.identity.new.impersonate('username','usertoeken')
    idents
  end

  def get_ident_by_token(token)
    user = some_lookup_code_to_find_user_details(token)
    return self.identity.new.impersonate(user.id, user.username) if user
    nil
  end

  def members_of_group(group)
#example from directorySQLite
    idents = Array.new
    GroupUserRelation.each_with(:GroupAccount, group.grouptoken, :equal) do
|gur|
      user = gur.UserAccount
      idents << self.identity.new.impersonate(user.id, user.username)
    end
    idents
  end

end
```

A Sample Validator Class

```
require_relative 'devicevalidator'

class DeviceValidatorSAMPLE < DeviceValidator

  def initialize(optional={})
    super optional
    #Anything else you need to do here
    #@optional[:symbol_name]
    #Provides access to the value from conf file
  end

  def lookup(devicename)
    return false if devicename == 'something I don't like'
    true #device is valid
  end
end
```