WebTool Hacking HOWTO

WebTool Hacking HOWTO

Revision History

Revision $Revision: 1.1 $ $Date: 2006/01/03 17:19:58 $ Revised by: pax

Table of Contents

Chapter 1. Introduction

This document is intended as a guide to writing additional modules for the Guardian Digital WebTool, a component of EnGarde Secure Linux. It will detail the WebTool API, the components of WebTool modules, and how to create a new module for the WebTool.

1.1. About the Guardian Digital WebTool

The Guardian Digital WebTool is a web based application that is the main interface used to configure and administer an EnGarde Secure Linux system. It runs as a service on an EnGarde system and is accessed through a standard web browser using the HTTPS protocol on port 1023.

1.2. Requirements

This document assumes a familiarity with Perl CGI programming techniques, understanding of HTML coding, and basic Linux systems administration knowledge. It is intended for developers who wish to extend and customize the WebTool, and is not written to document the use of the existing modules of the WebTool. A familiarity with Object Oriented programming techniques will also be helpful.

Chapter 2. The WebTool API

The WebTool API contains a variety of functions useful to a module developer. This chapter will detail some of the more common functions, but is not intended to be an exhaustive reference. You should refer to the source code of the *WebTool.pm* and *UI.pm* modules located under */lib* in your WebTool directory.

2.1. Reading and Writing Files

A very common task for WebTool modules is the reading and writing of configuration files. The WebTool API provides a *read_file* function accepting a file path as a parameter, returning an array of the lines in the file, and a *write_file* function that accepts a file path and and array of lines to write to the file.

```
use WebTool;

# open file and read contents into @f
my @f = WebTool::read_file("/etc/sample.conf");

# manipulate file contents stored in @f here

# write file back based on new contents of @f
WebTool::write_file("/etc/sample.conf", @f);
```

2.2. Writing to the WebTool Audit Log

In order to aid in troubleshooting and for security auditing purposes, it is recommended to output log lines whenever the WebTool is used to commit a change. This logfile is located at */var/log/webtool-audit.log* and can be written to using the *audit* function. The logging module of the WebTool can be used to display and search this logfile.

```
use WebTool;

audit("Created a new user: $username");
```

2.3. Page Output Functions

Page output functions are the bulk of the WebTool API. These functions are used to create HTML tags for form entry, as well as controlling the template system. They are methods of the *page* object which is an instance of either the *WebTool::UI::Std* class or the *WebTool::UI::Popup* class depending on whether the page is a normal page or a popup window. The main differences between the two are header and footer layout and the presence of a menu, programmatically they behave very much the same.

```
use WebTool::UI::Std;
my $page = new WebTool::UI::Std;
```

2.3.1. HTML Widget Functions

The WebTool API contains a variety of functions designed to output HTML widgets such as textboxes, selects, checkboxes, and the like. Creation of links and submit buttons is also performed by calling WebTool UI functions, and different functions exist depending on if the link or submit button launches a popup window or not.

These functions can be found in the */lib/WebTool/UI.pm* library under your webtool directory.

2.3.2. Templating Functions

The WebTool uses the Perl *HTML::Template* library to separate code from presentation, and provides methods of the page object to perform template related functions. See Appendix A for links to more information about the HTML::Template library.

```
use WebTool::UI::Std;
my $page = new WebTool::UI::Std;
my $tmpl_file = "templates/index.tmpl";

$page->tmpl_load($tmpl_file);
$page->set_page_title("My Title");
$page->tmpl_set("sample_var" => "My Value.");

$page->draw();
```

2.3.3. Error Functions

The *err* method of the page object is used within action CGI scripts (see Section 3.2.3) to launch a javascript popup noting an error and return to the form page that called the action CGI.

```
my $page = new WebTool::UI::Popup;
my %in = WebTool::UI::process_form();

if (!$in{'required_field'}) { $page->err("The Required Field is required."); }
```

2.4. Performing Translations

It is recommended when writing WebTool modules to place all strings in a separate language file for easier localization. See Section 3.4 for details on the format of these files.

The *tmpl_translate* method of the *$page* object is used to perform translations. If you define a template variable in your template to begin with a *lang_* prefix, it will be looked up in the appropriate language file and replaced in the template automatically, and *tmpl_translate* is not required to be called. See Section 3.3 for more information on how templates work.

```
use WebTool::UI::Std;

my $page = new WebTool::UI::Std;

my $title = $page->tmpl_translate('indextitle');
```

2.5. Loading Other Module Libraries

A very important capability of the WebTool is the ability to load libraries from other modules. This allows module authors to leverage code already written for use in another module, promoting code re-use and simplifying the WebTool as a whole. The *load_module* function is provided for this purpose. It must be called from within a *BEGIN* block and before any other code is run as shown below, or it will throw errors at runtime.

```
use WebTool;

BEGIN { WebTool::load_module('services'); };
```

When writing a module, you should review the available modules and see if any of your required functionality is included in their library modules. Code re-use should be a priority to keep the WebTool as lean as possible and prevent introducing unnecessary bugs. Check the libraries of modules related to the action you are performing to see if a related function exists. For example, if your module needed to perform a DNS related task, the *named.pm* library in the named module would be a logical place to check for a function that performs your desired action.

2.5.1. Stopping and Starting Services

A common use for calling another module is in the loading of the *services* module to start, stop, and restart services on the system. As an example, the code below will restart the dhcpd server if it is currently running on the system.

```
use WebTool;

BEGIN { WebTool::load_module('services'); };

my $services = new services;
if ($services->current_state('dhcpd') { $services->restart('dhcpd'); }
```

2.5.2. User Account Manipulation

Another common task requires the use of the *users* module to read or change user or group related information. The following example would determine if a specified user exists on the system.

```
use WebTool;

BEGIN { WebTool::load_module('users'); };

my $users = new users;
my @userlist = $users->users_list();

foreach my $u (@userslist) {
  if ($u->{'name'} eq $testname) { return true; }
}
return false;
```

Chapter 3. Module Components

The WebTool is divided into *modules*, each of which has a narrowly defined area of responsibility, such as Apache configuration or displaying of logfiles. These modules are also divided into separate components, which will be detailed in this chapter.

Modules are created within their own directory under the modules directory of the WebTool tree.

3.1. Module Library

Every module contains a module library named the same as the module directory, with a *.pm* extension. This library should contain functions for every administration task the module performs. This library can be considered the module's own API, as it is the interface to the module from the rest of the WebTool.

Other WebTool modules can load and call this module. See Section 2.5 for eamples of how and why a module might need to load and use another module's library.

3.2. CGI Scripts

Every module contains at least one CGI script called *index.cgi*. This is the main page of the module and will be displayed when the module is chosen from a menu item. Additional scripts can be created if necessary to break the module's functionality onto separate pages and should be named based on their task.

3.2.1. Display CGI Scripts

Normal display CGIs should use the WebTool::UI::Std class. This will ensure they contain the proper headers, footers and menus.

```
use WebTool::UI::Std;
my $page = new WebTool::UI::Std;
```

3.2.2. Popup CGI Scripts

Popup CGIs are generally used for displaying information or allowing input of small forms. They should use the WebTool::UI::Popup class, which uses different headers and footers and does not contain a menu. Use the popup_link or popup_button methods of your page object on the launching page to create the popup windows, see Section 2.3.1 for more information.

```
use WebTool::UI::Popup;
my $page = new WebTool::UI::Popup;
```

3.2.3. Action CGI Scripts

These are scripts whose purpose is to perform an action rather than display a page. Your form pages should post to an action page to perform the required task, which will then redirect back to a normal page using either a *close_popup* or a *redirect* function. The form script generally would be named with an *edit_* or *create_* prefix, while the associated action script would be named with a *do_edit_* or *do_create_* prefix.

A typical action script uses the *process_form* function to gather the HTTP post variables from the previous page, validates the input, calling the *err* function if any input is deemed invalid, performs the required task, then redirects or closes the popup.

```
use WebTool::UI::Popup;
my $page = new WebTool::UI::Popup;
my %in = WebTool::UI::process_form();

if (!$in{'required_field'}) { $page->err("The Required Field is required."); }

# Do action here

$page->close_popup();
# or if the calling form page was not a popup :
# WebTool::UI::redirect('index.cgi');
```

3.3. Templates

Templates are stored in the *templates* subdirectory of your module directory. Template files should be named the same as the CGI script that refers to them, with the extension changed to *.tmpl*. A template should exist for each CGI script in your module, except the action CGIs. They contain all the HTML code for the display of your module as well as special *HTML::Template* tags which are placeholders for variables you can set using the WebTool's templating functions (see Section 2.3.2).

There are several different *HTML::Template* tags that can be used. The <*TMPL_VAR*> tag replaces the tag with whatever value was set for it using the *tmpl_set* function, but there are also special tags to provide if/else branching and looping. See Appendix A for links to more information about the *HTML::Template* library.

<*TMPL_VAR*> tags are assigned a name in the template file. If this name begins with a *lang_* prefix, it is automatically replaced by the matching entry in the modules appropriate language file. No English strings should ever be present in a template file, they should be entered in the language file and called with a <*TMPL_VAR*> tag.

Templates should use <*DIV*> tags and follow the layouts and CSS styles established by the existing WebTool modules.

```
<DIV CLASS="pageEntry">
  <DIV CLASS="pageEntryTitle">
    <TMPL_VAR NAME="lang_indextitle">
  </DIV>
  <DIV CLASS="pageEntryContent">
    <TMPL_VAR NAME="lang_indexintro">
  </DIV>
</DIV>
```

3.4. Language Files

Language files are used for WebTool localization. They consist of a list of named tags, each followed by an equals sign and the string to replace the tag with. They are stored in the *lang* subdirectory of the module directory, and named with the appropriate two letter language code, i.e. *en* for English or *fr* for French. Currently the WebTool only contains English

translations, but French is supported in the code, and other languages can be added. Please contact the WebTool developers using the EnGarde Users mailing list (see Appendix A) if you would like to contribute localization help in any other language.

```
indextitle      = DHCP Server Management
indexintro      = Welcome to the DHCP Server Management module.  This module allows you
                  to set up this system for use as a DHCP server, which will assign IP
                  addresses to other systems.
```

3.5. Access Control Files

Access Control Files are used by the WebTool to constrain access to certain modules from restricted WebTool users. The Access Control file is stored in the your module directory and is always named *.acl*. It consists of a listing of the CGI scripts belonging to the module, each followed by two four digit module codes (assigned in the webtool module, see its source code for further details) and a *1* for pages that only require read access or a *2* for pages which require read and write access. Generally, the action CGI scripts are assigned a *2* since they are the scripts that make actual changes, other CGI scripts are assigned a *1*.

The four digit module codes are assigned by Guardian Digital. Please contact Guardian Digital via the EnGarde Users mailing list (see Appendix A) if you need to be assigned a code for your module.

```
# --- BEGIN ---
# 0020  0290    POP/IMAP Server Management
# --- END ---
#
index.cgi                       0020    0290    1
edit_cert.cgi                   0020    0290    1
do_edit_cert.cgi                0020    0290    2
do_edit_interface.cgi           0020    0290    2
```

Chapter 4. Notes For Module Authors

This chapter contains notes and other miscellaneous information useful to module authors.

4.1. Writing New Modules

When writing a new module, you should try to adhere to the way existing modules are laid out and constructed. There is no better documentation than the source code itself, so study and dissect the existing modules carefully before attempting to write your own from scratch. Experiment with changing existing modules to see how the changes affect the module behavior, and be sure to familiarize yourself with the API functions and use them whenever possible to avoid duplication of effort.

4.2. Style Sheets

The WebTool uses CSS for all styling and layout. Some WebTool pages use a single column layout, while others are divided into left and right columns. This layout is specified in the WebTool CSS file, located at */templates/webtool.css*. You should refer to the existing template files for examples of how to declare your <*DIV*> tags for each type of layout.

Blocks of HTML should be declared as displayed below. First a *pageEntry* class *DIV* tag, followed by a *pageEntryTitle DIV* tag containing the title of the block, then a *pageEntryContent DIV* tag containing your content, with tables enclosed by a *popupEntryWhite DIV* tag. Tables should always be assigned a *listing* class, and any table header rows should be assigned a *listing-title* class. Following these guidelines will ensure that your module adheres to the look and feel of the existing modules, which helps usability and maintains a consistent experience within the WebTool.

```
<DIV CLASS="pageEntry">
  <DIV CLASS="pageEntryTitle">
    <TMPL_VAR NAME="lang_indextitle">
  </DIV>
  <DIV CLASS="pageEntryContent">
    <TMPL_VAR NAME="lang_indexintro">
    <DIV CLASS="popupEntryWhite">
      <TABLE WIDTH="100%" CLASS="listing">
        <TR CLASS="listing-title">
          <TD>Header 1</TD>
          <TD>Header 2</TD>
        </TR>
        <TR>
          <TD>Data 1</TD>
          <TD>Data 2</TD>
        </TR>
      </TABLE>
    </DIV>
  </DIV>
</DIV>
```

For pages with a multi-column layout, use a *pageEntryColumn* class rather than *pageEntry*, with either a *pageEntryLeft* or *pageEntryRight* DIV within it, as seen below.

```
<DIV CLASS="pageEntryColumn">

  <DIV CLASS="pageEntryLeft">
```

```
    <DIV CLASS="pageEntryColumnTitle">Left Column Title Here</DIV>
    <DIV CLASS="popupEntryWhite">
       <DIV CLASS="pageEntryContent">
         Left Column Content
       </DIV>
    </DIV>
  </DIV>

  <DIV CLASS="pageEntryRight">
    <DIV CLASS="pageEntryColumnTitle">Right Column Title Here</DIV>
    <DIV CLASS="popupEntryWhite">
       <DIV CLASS="pageEntryContent">
         Right Column Content
       </DIV>
    </DIV>
  </DIV>

</DIV>
```

Please follow these guidelines as closely as possible when designing the layout of a WebTool module, maintaining a consistent user experience with your module relative to the rest of the WebTool is as critical as proper functionality.

4.3. Menus

Alterations to the WebTool menus need to be made in the */lib/WebTool/UI/Std.pm* library. Currently the menu is hardcoded in this library, as part of the *make_page_menu* function.

4.4. Module Configuration Files

If your module needs to save information about itself, such as preferences, a file can be created in the */webtoold* subdirectory of the main webtool directory. An example of this can be found in the source code to the backup module, which keeps preferences stored in a *backup.conf* file in that directory..

4.5. Sample Module

I've written an extremely basic sample module in the *Hello World* vein for aspiring module authors to examine and dissect. Download the tarball from the link below, and copy it to your WebTool *modules* directory. Extract it by typing *tar xvfz webtool-module-sample-<version>.tar.gz* and then access it by entering *https://yourhost:1023/modules/sample/* in your browser's address bar after logging in to the WebTool.

This basic sample module can be useful in understanding the necessary pieces of a WebTool module and how they inter-relate in a simplified way. It can be used as a convenient starting point when writing your own modules as well.

ftp://ftp.engardelinux.org/pub/engarde/people/pax/webtool-module-sample/

Appendix A. Resources

This is a list of valuable online resources for WebTool developers.

- **EnGarde Users Mailing List** - The EnGarde users mailing list is an excellent place to ask questions about EnGarde Secure Linux and WebTool development. The developers of the WebTool and its API are active contributors to the list.

  http://infocenter.guardiandigital.com/community/

- **EnGarde Secure Linux wiki** - This wiki is a community developed document repository for EnGarde Secure Linux documentation, and is where the most recent versions of this document, as well as many other useful EnGarde Secure Linux documents, can be found.

  http://wiki.engardelinux.org/

- **EnGarde Secure Linux IRC channel** - EnGarde friends are welcome to come chat on the EnGarde Secure Linux IRC channel. To chat, join the *#engarde* channel on *irc.freenode.net*.

- **HTML::Template** - This is the home site for the HTML::Template library used by the WebTool. A good understanding of how this library works is critical to creating the layout of WebTool modules.

  http://html-template.sourceforge.net/