Guide to cPanel Market Provider Modules

Introduction

Note:

We added this functionality in cPanel & WHM version 56.

Warnings:

We recommend that **only** advanced users use this feature.

This document explains how to create a third-party custom cPanel Market provider module with basic functionality. The cPanel Market system allows you to purchase and automatically install SSL certificates and other products through cPanel's *SSL/TLS Wizard* interface (*cPanel* >> *Home* >> *Security* >> *SSL/TLS Wizard*). cPanel & WHM servers ship with the cPanel Store provider module.

Third-party vendors can create their own provider modules in Perl in the `/var/cpanel/perl/Cpanel/Market/Provider/` directory and should use the `Cpanel::Market::Provider` namespace. For example, the Example certificate provider would create the `/var/cpanel/perl/Cpanel/Market/Provider/Example.pm` module in the `Cpanel::Market::Provider::Example` namespace.

Module function interfaces

Third-party developers **must** implement the following functions in order to create a functional cPanel Market provider module:

`convert_identity_verification_to_csr_subject`

This function retrieves the `identity_verification` information from the item description and returns the output that the `csr` subject will contain.

Parameters

A list of `key=value` pairs:

- Returns a list of `key=value` array references, which the system adds to the `csr` subject.

Returns

The information provided in the `identity_verification` hash from the `request_ssl_certificates` function.

Equivalent UAPI Function

None.

Constants

We recommend that third-party developers declare the following constants:

Constant	Description	Example
`REQUEST_URI_DCV_PATH`	The path to DCV check file, relative to the document root directory.	`'^/[A-F0-9]{32}\\.txt(?: Sectigo DCV)?$',`
`URI_DCV_ALLOWED_CHARACTERS`	The characters that the provider allows in the filename that it uses to check for DCV.	`[ 0 .. 9, 'A' .. 'Z' ],`

Market Provider Modules

- *The cPanel Market allows service providers to sell products, such as SSL certificates, in the cPanel Market.*
- *The product list appears in cPanel's* SSL/TLS Wizard *interface* (WHM >> Home >> Security >> SSL/TLS Wizard).

Compatible with:

- cPanel & WHM 56+

Related Documentation

`URI_DCV_RANDOM_CHARACTER_COUNT`	The number of characters that the provider allows in the DCV check filename.	`32,`
`EXTENSION`	The DCV check file's extension that the provider requires.	`'txt'`
`DCV_USER_AGENT`	The user agent string that the system uses for the imitated local DCV check.	`'SECTIGO DCV'`

cPanel Market purchase workflow

Third-party provider modules **must** use the following workflow:

1. The user prepares a shopping cart in the browser with information that the provider's custom Perl module returns through the `get_products_list` function.
2. When the user wants to check out, the cPanel application redirects the user to the login page that the UAPI Function `get_login_url` function provides.
3. After the user logs in, the login server redirects the user back to the cPanel application with a code in the URL query parameter. The cPanel application calls the `validate_login_token` function in order to send that code to the provider and obtain an access token.
4. The cPanel application prepares a shopping cart with the access token:
    a. The system validates the requests with the provider with the `create_shopping_cart` function. You can insert custom logic with the `validate_request_for_one_item` function.
    b. The `create_shopping_cart` function returns an `order_id` value that the cPanel application will use to maintain state after payment.
    c. The `create_shopping_cart` function also returns a unique `order_item_id` value for each item that the customer orders.
    d. If the user purchased a certificate, their server begins to poll the provider for that certificate's `order_item_id` value.
5. The cPanel application calls the `set_url_after_checkout` function to set a post-checkout redirection URL.
6. The user's server redirects the user to a checkout URL that the `get_checkout_url` function provides.
7. The provider processes payment, but if the order is for a certificate, they do **not** finalize the charge.
8. The provider redirects the user back to the cPanel application (for example, the URL that the `set_url_after_checkout` function previously set).

Certificate orders

For certificate orders, the module performs the following additional steps:

1. The cPanel application calls the `get_ssl_certificate_if_available` function at regular intervals to poll the provider for the certificate.
2. After the provider issues the certificate, the provider finalizes payment.

> **Note:**
>
> We recommend that you finalize payment **after** you issue the certificate in order to avoid unnecessary additional chargeback fees. If the user deletes the Domain Control Validation (DCV) file from their account before the DCV process happens, this effectively cancels the order.

3. If a poll is successful, the cPanel application downloads and installs the SSL certificate, and then it stops polling.

Example

The following custom provider module outline demonstrates a minimal set of functionality.

> **Warning:**
>
> This example does **not** reflect a fully-functional module and only demonstrates a basic workflow. Your implementation requires more internal logic. Also, this example does **not** demonstrate the necessary API functions that would allow your module to hook into your store.

```
package Cpanel::Market::Provider::Example;
```

- Market Provider Manager
- Market Provider Manager
- WHM API 1 Functions - get_login_url — This function retrieves the login URL for the cPanel Store or a cPanel Market provider.
- WHM API 1 Functions - validate_login_token — This function validates a login token with the cPanel Store or a cPanel Market provider, and then returns access tokens.
- WHM API 1 Sections - Market — Market functions allow you to manage cPanel Market providers and products available to your users.

```
use strict;
use warnings;
use autodie;

use constant {
    #The last bit here doesn't actually happen; we just want
to
    #tag the regexp so that it's very clear where it came
from since
    #otherwise this pattern is somewhat generic.
    REQUEST_URI_DCV_PATH           => '^/[A-F0-9]{32}\\.txt
(?: Sectigo DCV)?$',
    URI_DCV_ALLOWED_CHARACTERS     => [ 0 .. 9, 'A' .. 'Z' ],
    URI_DCV_RANDOM_CHARACTER_COUNT => 32,
    EXTENSION                      => 'txt',
};

use HTTP::Tiny ();

sub _DISPLAY_NAME { return 'Example's stuff' }

#-------------------------------------------------
### STEP 1 - The products list.
### example1 is a certificate, which will exist in the
ssl_certificate product group.
### example2 is something other than a certificate, which
will exist in the example_things product group.
sub get_products_list {
    return(
        {
            product_id => 'example1',
            description => 'A certificate',
            display_name => 'Example Cert',
            price_unit => 'USD',
            product_group => 'ssl_certificate',
            x_price_per_domain => 100,
            x_ssl_per_domain_pricing => 1
        },
        {
            product_id => 'example2',
            description => 'A thing',
            display_name => 'Thing #2',
            price => 5,
            price_minimum => 4,
            price_unit => 'USD',
            product_group => 'example_things',
            x_price_per_wildcard_domain => 200,
            x_price_per_wildcard_maximum => 1000,
            x_price_per_wildcard_minimum => 100
        },
    );
}

#-------------------------------------------------
### STEP 2 - Get the login URL.

sub get_login_url {
    my ($after) = @_;

    return 'http://example.com/cpmarket_login.html?' . HTTP::
Tiny->www_form_urlencode(
        {
           ### This is where we input the redirection URI
           redirect_uri => $after,
        },
    );
}

#-------------------------------------------------
### STEP 3 - Let's validate the code and send the code to the
```

```
provider get the token

sub validate_login_token {
    my ($token) = @_;

    return { access_token => "access_$token" };
}

#-------------------------------------------------
### STEP 4a - If you want to validate the request for an item.
### Validation of the item depends on your product list and
store API structure.
### This subroutine intentionally left blank.

sub validate_request_for_one_item {}

### STEP 4a - Create the shopping cart.

sub create_shopping_cart {
    my (%opts) = @_;

        #Usually this will come from a remote service that
manages the shopping cart.
        #For the purposes of this demonstration module,
however, we generate a random order_id.
        #The order ID can be any provider-unique string or
numeric value;
        #i.e., no two orders may ever have the same order ID.
        my $order_id = int rand 100000;

        return(
        ### STEP 4b - Your provider generates the true
order_id. We use "int rand 100000" to skip that step.
        int( rand 100000 ),
        [
            {
                ### Each item in the cart must receive a
provider-unique order_item_id.
                                ### A given provider may only
ever use a given order_item_id once.
                                order_item_id => sprintf('%
04x', int rand 65536),
            },
        ],
    );
}

#-------------------------------------------------
### STEP 5 - Tell the checkout where we should go after
checking out.

sub set_url_after_checkout {
    my (%opts) = @_;

    #This needs custom code that will use the "access_token"
    #to set the "url_after_checkout for the given "order_id".
}

#-------------------------------------------------
### STEP 6 - Redirect to the checkout URL.

sub get_checkout_url {
    my ($after) = @_;

    return 'http://example.com/cpmarket_checkout.html?' .
HTTP::Tiny->www_form_urlencode(
        {
            order_id => $after,
        },
    );
}
```

```
#-------------------------------------------------

sub get_support_uri_for_order_item {
    my (%opts) = @_;
    my $query = HTTP::Tiny->www_form_urlencode(
        { map { ( $_ => $opts{$_} ) } qw( order_id
order_item_id ) },
    );
    return "http://help.me.rhombus?$query";
}

#-------------------------------------------------
### LOGIC SPECIFIC TO SELLING SSL CERTIFICATES
#-------------------------------------------------

sub get_ssl_certificate_if_available {
    my ($order_item_id) = @_;

    #Check for the certificate; if it's available, return it
(PEM format).
        #Otherwise...

        return undef;
}

### Domain control validation depends on how your Certificate
Authority
### confirms ownership of the domain. For example, Sectigo’s
validation
### looks for specific content at a specific, publicly-
accessible URL;
### both the content and the URL are functions of the CSR.

### If you are not doing automatic DCV, (e.g., you're
requiring the end
### user to respond to an email) then you can safely leave
these functions blank.

sub prepare_system_for_domain_control_validation {
        my (%opts) = @_;        #product_id, csr
}

sub undo_domain_control_validation_preparation {
        my (%opts) = @_;        #product_id, csr
}

### The following functions are available to retrieve
information from the
### item description. The retrieved information is then added
to either the
### CSR subject or the item parameters.

sub convert_ssl_identity_verification_to_csr_subject {
    my ($product_id, %id_ver) = @_;
    my @keys_in_csr = grep { _is_in_csr($_) } keys %id_ver;
    #Any kind of manipulation of the information could be
done here.
    return map { [ $_ => $id_ver{$_} ] } @keys_in_csr;
}

sub
convert_ssl_identity_verification_to_order_item_parameters
    my ($product_id, %id_ver) = @_;
    my @keys_not_in_csr = grep { !_is_in_csr($_) } keys %
id_ver;
    #Any kind of manipulation of the information could be
done here.
    return map { $_ => $id_ver{$_} } @keys_not_in_csr;
}
```