



# **CarrotPay: Script Integration Guide**

Version 0.2

Copyright © RMP Protection Limited 2008

---

## Contents

Introduction.....	3
What you need.....	3
How it works.....	3
1. Registration.....	3
2. Integration.....	3
3. Purchase.....	4
The payment form.....	4
Preventing fraud.....	5
The secure URL algorithm.....	6
Example in PHP.....	6
Using CarrotPay in practice.....	7
Transactional systems.....	7
Ticket-based systems.....	8
Stateless Systems.....	8
Security of the seed value.....	9

---

## Introduction

This document gives a technical guide to integrating your website with the CarrotPay payment system using Web scripting languages such as PHP. If you are not a programmer or you can't install scripts on your Web server, then please see the "CarrotPay: HTML Integration Guide" for a simpler solution which only requires basic HTML.

You should also read the "CarrotPay: Overview for Merchants" document so you are familiar with the general concepts of CarrotPay payment before reading this guide.

## What you need

We're assuming you have, or are building, a Website where you want to sell goods or services, either for delivery online or physically. Unlike the HTML-only solution, using scripting we believe it is possible to secure your site against fraud sufficiently to allow delivery of real goods, but of course you will want to make your own decision on this.

We will assume you are familiar with general Web development concepts, including your chosen scripting language and SQL databases. This document gives examples in PHP, but the concepts can easily be translated into any other language such as ASP, Python or Java/JSP.

To collect your payments, you'll need to have a Carrot Purse installed, as will your customers who want to pay with Carrots. The CarrotMail Client from [www.carrot.org](http://www.carrot.org) acts as a full Carrot Purse as well as offering excellent anti-spam protection.

---

## How it works

As described more fully in the "CarrotPay: Overview for Merchants" document, the basic process for setting up for CarrotPay payments is as follows:

### 1. Registration

First you need to register for a CarrotPay merchant account, at the following URL: <http://www.carrot.org/carrotpay>

This is quick and anonymous, and gives you three things:

1. A CarrotPay merchant ID, which looks like this: "BJGC-QZPM-YQBC-GJQP"
2. A 'hash seed', which looks like this: "mjqbpzcejwxmbpzcb"
3. A 'secret key', which is never shown to you

All three of these are kept securely by your Carrot Purse (e.g. CarrotMail Client). You only need the first two for CarrotPay integration; the secret key is only used to fetch your received Carrots back from CarrotPay.

When you click on the "Register" button, CarrotPay will immediately create your account and send a special file to your Carrot Purse which enables the merchant functions and gives it the merchant ID, seed and secret key as described above. You are now registered with CarrotPay and your Carrot Purse is set up to fetch the Carrots as they come in from customers.

### 2. Integration

Then you need to add CarrotPay buttons to your website, either as individual "buy now" buttons or as the final stage in your shopping cart process. These are simply HTML FORMs which point to a CarrotPay service, and include details of the product and a 'return URL' where the customer will be redirect after authorising payment. The return URL can be modified by the service to provide security, as we will see below.

Here is an example of a simple payment form:

```
<form action="https://www.carrot.org/payment" method="POST">
  <input type="hidden" name="merchant" value="KBCS-ZMSW-PDRZ-DVRH">
  <input type="hidden" name="total" value="10">
  <input type="hidden" name="description"
    value="Image from Trehaverne Image Library">
  <input type="hidden" name="return_url"
    value="http://www.trehaverne.org/ret.php?tx=342&sig=[342]">
  <input type="hidden" name="fail_url"
    value="http://www.trehaverne.org/failed.html">
  <input type="submit" value="Pay with Carrots">
</form>
```

The CarrotPay HTML Integration guide gives some further hints on how to improve the look of the process, in particular how to avoid a blank page while waiting for the customer to authorise the payment on their Carrot Purse.

### 3. Purchase

When a customer visits your site and wants to buy something, the following happens:

1. They click on the CarrotPay button for the item they want to buy, or by choosing to pay with Carrots at the end of your shopping cart process.
2. The button/form points to the CarrotPay website, which generates a blank page containing some Javascript which communicates with the customer's Carrot Purse to request the payment.
3. The customer's Carrot Purse asks them to authorise the payment, or, if the amount which you are requesting is less than their automatic payment limit, will just notify them that a payment has been made.
4. The customer's Carrot Purse then sends the right value of Carrots to CarrotPay, where they are checked and are stored ready for you retrieve later.
5. CarrotPay modifies the return URL to provide security verification, and the Javascript pushes the user's browser back to your site using that URL.
6. You use the security information to verify that the purchase is genuine, and then continue with your delivery process.

Then at some time later your Carrot Purse will automatically fetch the received Carrots so you have them available locally, either to spend on other online services yourself, or to redeem for cash. Carrot Pay applies a small transaction charge (which is what pays to keep the CarrotPay service running) as you collect your Carrots.

---

## The payment form

The payment form or button is a standard HTML FORM which POSTs to **https://www.carrot.org/payment**

The FORM requires the following hidden INPUT fields:

Input name	Value	Example
merchant	Merchant ID (16 letters with dashes)	"KBCS-ZMSW-PDRZ-DVRH"
total	Price to charge, in Carrots	"10"
description	Short (one line) description of the product	"Image from Trehaverne Image Library"
return_url	Unmodified URL with square brackets	" http://www.trehaverne.org/ret.php?tx=342&sig=[342]"
fail_url	Failure URL	" <a href="http://www.trehaverne.org/failed.html">http://www.trehaverne.org/failed.html</a> "
Target (optional)	Window or frame target for return URL	"top"

The following is a template for such a form:

```
<form action="https://www.carrot.org/payment" method="POST">
  <input type="hidden" name="merchant" value="(your merchant ID)">
  <input type="hidden" name="total" value="(price)">
  <input type="hidden" name="description"
    value="(product description)">
  <input type="hidden" name="return_url"
    value="(unmodified URL with square bracket sections)">
  <input type="hidden" name="fail_url"
    value="(URL for failure page)">
  <input type="hidden" name="target" value="(return target)">
  <input type="submit" value="Pay with Carrots">
</form>
```

Additional INPUT fields **should not** be added, since they will be ignored, and future versions may provide more features and we will wish to allow backwards compatibility.

---

## Preventing fraud

The issue with any payment system which uses the customer's browser as the integration point (rather than back-end integration through SOAP, for example) is that the return URLs can be 'spoofed' by an attacker, short-circuiting the real payment mechanism and obtaining goods fraudulently. Other payment services have systems such as encrypted buttons, IPN, PDT and so on to avoid this. We believe the CarrotPay system is much simpler, but just as secure...

The CarrotPay security mechanism is simple yet powerful: When you register as a CarrotPay merchant you get a 'seed' value which is a shared secret between you and the CarrotPay service. The CarrotPay payment service then uses this seed to modify any elements of the URL that you enclose in square brackets ("[...]") to a hash value including the seed, the content of the brackets and the price of the transaction. The result is an 8-character text string, representing a 32-bit value. The full algorithm is described in the next section.

For example, the return URL

```
http://www.mysite.com/scripts/return.php?tx=45432871&hash=[45432871]
```

might be modified to become

```
http://www.mysite.com/scripts/return.php?tx=45432871&hash=hcjmxkzp
```

Your return URL handler will then extract one or more elements of the return URL and redo the same hash calculation internally to compare against them before allowing the transaction to proceed. The result of this is an attacker cannot just look at the HTML FORM to find out the return URL and 'spooF' it without going through the genuine payment mechanism. Because the price is included in the hash, they can't modify the price on a genuine payment form and get the goods for less, either.

The main issue you then have is to prevent the same return URLs being used more than once. This is an unavoidable issue with simple HTML integration, which is why we don't recommend it for delivery of physical goods. However, with scripting there are a number of ways to prevent this, as we will see below.

---

## The secure URL algorithm

After CarrotPay has verified that the customer has presented the right number of valid Carrots to meet the price quoted in the payment form, it then modifies the return URL before redirecting the customer's browser to it.

The return URL is modified by substituting strings in square brackets “[...]” with a hash according to the following algorithm:

1. Begin a string with the text inside the square brackets (excluding the brackets themselves)
2. Append a single space (32)
3. Append the total value requested *in the exact textual format quoted in the payment form*
4. Append a single space (32)
5. Append the seed string (lowercase, no spaces or dashes)
6. Get the MD5 hash of the resulting string
7. Get the lowest 32 bits (final 4 bytes) of the MD5 hash as an unsigned 32-bit integer (big-endian, most significant byte first)
8. Convert to an 8-character string using a 'safe'<sup>1</sup> base 16 alphabet: "bcdghjklmpqrsvwz" (the equivalent for normal hexadecimal would be "0123456789abcdef"). Do not 'zero' (actually 'b') pad the output – i.e. value 1 is “c”, not “bbbbbbc”.
9. Substitute the resulting string for the entire bracketed item (including the brackets).

Alternatively, if (as in many text-oriented scripting languages), your MD5 operation gives you a hex string rather than a binary buffer, you can do the latter stages as a simple textual operation:

7. Take the last 8 characters (final 4 bytes) of the MD5 hash expressed in hex
8. Map the hex alphabet (“0123456789abcdef”) to the 'safe' alphabet (“bcdghjklmpqrsvwz”), trimming off any leading “zeros” ('b' characters).
9. Substitute the resulting string for the entire bracketed item (including the brackets).

## Example in PHP

The following PHP function does this hash for you for an individual 'word', using the textual version:

```
function carrotpay_hash_word($word, $price, $seed)
{
    // Construct hash text from seed, price and word
    // Note space delimited
    $text = "$word $price $seed";

    // Get md5 (hex string)
    $md5 = md5($text);

    // Get last 8 hex chars (32 bits, unsigned)
    $hex = substr($md5, -8);

    // Replace with 'safe' alphabet
    $safe = strtr($hex, "0123456789abcdef", "bcdghjklmpqrsvwz");

    // Trim off leading 'b' (zero)
    return ltrim($safe, "b");
}
```

---

<sup>1</sup> The selection of letters is made to avoid the possibility of any offensive words being created accidentally, and any confusion between letters and numbers.

---

## Using CarrotPay in practice

So how does CarrotPay integration work in practice? The general URL-modification system described above gives you plenty of flexibility, but there are some standard ways of using it that will probably make sense in most situations which we'll describe here.

The requirements for a secure payment integration are:

1. A valid return URL (which ends up delivering a product) cannot be obtained other than by genuine payment or unfeasible brute-force attack.
2. A valid URL which has been used once cannot be reused by:
  - a) The same person; or
  - b) A different person

Normally you would aim for both 2(a) and 2(b), but there are some cases – for example, online content delivery - where you would accept reuse by the same person within a short window of time, and this may save you effort in simple cases as we'll see in “Stateless Systems” below.

### Transactional systems

In fully-fledged e-commerce Websites there is usually a concept of a 'transaction' or 'sales record' which is stored in a database once the user has reached a certain point in the process, and which can be referred to by a simple transaction ID (e.g. the database's row ID, or a separate serial number).

In this case, the normal return URL (if you weren't bothered about security) would have the transaction encoded in it, which the site can pick up to continue the transaction through to fulfillment. For example:

```
http://www.mysite.com/scripts/return.php?tx=45432871
```

If this URL were put into the form unmodified, an attacker could simply copy it to their browser and obtain the goods without going through payment. The solution is to use the URL modification system to hash the transaction ID, which you can then verify by repeating the hash calculation (using a function like that above) to check that payment has been made.

Of course, if you just put the transaction ID itself in square brackets:

```
http://www.mysite.com/scripts/return.php?tx=[45432871]
```

you would only get back the hashed version:

```
http://www.mysite.com/scripts/return.php?tx=bcjmxkzp
```

and you wouldn't be able to look up the transaction (hashes are not reversible, and are not guaranteed to be unique).

The solution is to include the transaction ID **twice**, once as usual without square brackets, and once as a separate parameter (e.g. 'hash') inside square brackets:

```
http://www.mysite.com/scripts/return.php?tx=45432871&hash=[45432871]
```

The first one will be left alone, and the second one turned into the hash value, so you get back:

```
http://www.mysite.com/scripts/return.php?tx=45432871&hash=bcjmxkzp
```

You can then lookup the transaction with the real transaction ID, and repeat the hash (including the transaction ID, price and our secret seed value) to verify that it has gone through payment. Alternatively, you could calculate the hash beforehand and store it in the database ready for comparison. Either way, an attacker cannot create the hash without the seed, so you meet requirement (1).

**Note that it is critical that the price used in the hash calculation is textually identical to the price quoted in the payment form.** If you store prices in the transaction database, beware of it changing format when you retrieve it (this is one reason for pre-calculating the hash at the time you create the payment form).

In this case, because you have persistent state in the database, meeting both parts of requirement (2) – avoiding reuse of URLs – is easy: You simply have a status flag in the database which indicates whether a transaction has already been delivered, and refuse to allow a repeat delivery on the same transaction ID. Even if an attacker could guess your transaction IDs (because they are serialised), they still can't generate a valid hash without the seed value.

## **Ticket-based systems**

If your Website doesn't have a full database back-end, there are still ways you can protect your content from 'spoofed' or reused URLs. This involves creating a 'ticket' for each purchase, which you store either in a minimal database table or just in a flat text file.

The ticket is just a large random number, which you invent when the user starts to buy a product and then use as a 'transaction ID' exactly as above, encoded in the return URL both inside and outside square brackets. When a return URL is fetched, you check that your internal calculation of the hash (using the ticket number and a fixed price) matches the hash quoted, and that the ticket is in your 'database'. If so, you allow the user to continue, and delete the ticket, which stops it being used again.

### **Differing prices**

If you're selling products at different prices you will need to have the individual price available to construct the hash. Could you just include it in the URL? You could, but an attacker could then change both the value in the URL **and** the one in the payment form and generate a valid hash for a different price, and you'd be none the wiser. In this case you either need to store the price with the ticket, or have another way to get the price for a given product to verify the hash.

### **Confirmation pages**

Note that you can't quite replicate the functionality of the simple HTML buttons discussed in the HTML Integration Guide this way, unless you invent a ticket for every button. It's better to take the user through a confirmation page first, where you can invent the ticket with a reasonable expectation that they will complete the transaction.

### **Pruning dead tickets**

Over time, you'll also need to prune out old tickets which never completed. For this you'll need a timestamp on each ticket and some process which deletes them when they are more than a certain age (a day is probably ample for someone to complete a transaction). A good time to do this if you're using flat files is when you delete used tickets, because you have to scan and resave the whole file then in any case. If you're using a database, could do it at any time.

## **Stateless Systems**

But what if you don't want to have any stored state? There is a set of interesting ways to use simple scripting to improve on the basic renaming-of-files trick we describe in the HTML Integration Guide, but which doesn't involve storing any state in databases, files or otherwise. In this case you can meet our requirement 2(b) to stop links being shared between customers, but you can't stop the same customer fetching it more than once in quick succession. Hence this only makes sense for online delivery where you don't mind (or even might want to allow) a customer downloading something multiple times.

### **Simple reusable URLs**

The very simplest technique doesn't try to provide protection against reuse at all, so is just the same as the HTML-only version. However, it is very easy to implement and allows you to provide basic payment without the hassle of renaming files. The trick is to use a return URL with the filename included twice, once as normal and once inside square brackets – e.g.:

```
http://www.mysite.com/scripts/return.php?file=cat.jpg&hash=[cat.jpg]
```

The 'return.php' script then regenerates the hash of the filename, a fixed price and the secret seed, and compares it to the 'hash' parameter. If it matches, and given suitable checks on the filename, it can then just output the file to the user (e.g. using readfile in PHP).

Why a fixed price? Again, you can't trust the URL not to be modified in sync. with the payment form, and you now have nowhere to store it locally. One way around this would be to make the filename somehow indicate the price, and extract the price for verification from that – e.g. 10-cat.jpg, 20-beach.jpg. Alternatively you may have a static list of products (e.g. in a PHP Array, or a flat file which you read in) in which you could look up the filename to get the price.

### **Limiting by time**

As we warned above, these URLs can just be reused by anyone. What could you do to limit their re-use without having to store anything on the site? Firstly, you could limit how long the URLs are valid for. If you kept this to a few minutes, or even an hour, this would radically reduce the URL's usefulness.

The solution here is to include a timestamp in the URL, and also in the hashed text in square brackets, to stop someone just changing the timestamp. When you get the URL back in your return script, you can then check the timestamp is no more than a certain time in the past, and use it to construct the hash. For example:

```
http://www.mysite.com/scripts/return.php?
```

```
file=cat.jpg&t=1201586402&sig=[1201586402-cat.jpg]
```

In this case the timestamp 't' is the Unix timestamp (as returned by time()), but you could use any format. Notice how the timestamp in the square brackets is separated from the filename by a dash – separating elements of a hash like this is always a good idea to avoid any attack by moving characters from one element to another.

### Limiting by IP address

Finally, we can tie things down even more by including the user's IP address. It's fairly unlikely (although just about possible, with dynamic dialup addresses and/or browser caches) that the user will change IP address between fetching your payment button and you getting the return URL. Hence you could include their IP address (PHP: \$REMOTE\_ADDR) in the URL and hashed text as well, and check it is the same when they return:

```
http://www.mysite.com/scripts/return.php?  
file=cat.jpg&t=1201586402&ip=84.1.2.3  
&sig=[1201586402-84.1.2.3-cat.jpg]
```

Now an attacker would have to use the same IP address and within a certain time to make use of the link – pretty difficult to organise! It still allows the same user to reuse the URL within a certain time, but for delivery of online content, that is usually a bonus, since downloads may fail – but bear in mind that the reason downloads often fail is people lose their Internet connection, in which case they may come back with a different IP address.

### Security of the seed value

In all of the above scenarios, possession of the secret seed value enables anyone who has read this document to create a valid hash value in the return URL, and hence bypass the CarrotPay checks and obtain goods or services fraudulently. It is therefore critical that this is kept secret!

The problem arises because the script code which generates your hash for checking the return URL has to have access to the seed as well. What's worse, this code is sitting on a public Web server, so you have to be **very** careful where you keep the seed value.

There are two sources of attack you need to protect yourself from:

1. People using standard Web access to the site, but perhaps changing URLs or submitting malicious values into forms
2. Other users who are logged into the same Web server's command line (if it's shared).

### Protecting from Web attackers

In the first case, it's critical that the seed value isn't kept in a file which can be fetched as source from the Web. In most systems, if you include it as a constant in the script itself you should be OK, but beware of putting in into include files unless they are named properly. For example, in PHP, you may have a convention to put shared constants and functions in “xxx.inc” files, but this is dangerous, because the Web server probably isn't configured to interpret “.inc” files as PHP, and will deliver the raw source. Similarly, make sure the seed isn't stored in a configuration file or properties file that could be read from outside.

Also, and for a host of other reasons, you need to ensure that your site is protected from 'injection' attacks where attackers create malicious URLs or form data to get the site to execute whatever code they like. One very common use of this type of attack is to read out the content of any file in the system!

### Protecting from other users

Protecting from other users on the same shared Web server is harder. In general, you can't protect from the administrators of such servers, and if this is a concern to you, you will just have to use a dedicated server where you are fully in control. However, the bigger problem comes from other users like yourself, particularly if the Web server provides a command line (SSH) or open FTP access where users can read each other's directories.

Usually, the Web server process runs as a special user, called “www-data” or “httpd”. You can protect yourself to some extent by ensuring that the file containing the seed value is *only* readable by this user, and not anyone else. How you do this will depend on how you manage your files on the server.