Processing HTML, Part 2

By Cal Henderson

If you've read my previous article, "Processing HTML", then you'll know that filtering user input can be a real pain. But i've got bad news - it's worse than you think. There are a number of exploits that will get around our system. And these exploits can present a real problem when they're used to run javascript on the client browser. We'll look at each of these issues seperatly, finding code solutions for each. Afterwards we'll try and combine all we've learnt into a fully fledged filter library.

You really need to read the first article first - we're going to be building on that code from the start.

Tag Balancing

The easiest loophole to "exploit" is tab balancing. This doesn't cause security issues, as with the exploits below, but can be very annoying, and effectively disableyour site. Imagine a user posting the following message:

```
<b>This is bold
```

This is relatively benign - a mistake rather than an explict attempt at naughtiness. But it could be much worse:

```
Foo</div></div></div></div></div></div>
```

...which would break out of your site's structure and break everything after it (replace the closing div tags with however your site is structured.

Stopping it is reasonably easy. First, we need a list of tags that we don't care about here - that is, a list of tags that open but don't close (like img, br, input, etc.). What we'll want to do with these is always self close opening tags, and strip closing tags:

```
<img src="a">              becomes    <img src="a" />
<img src="a">foo</img>     becomes    <img src="a" />foo
</img>                     becomes    -nothing-
```

The current code for outputting closing tags looks like this:

```
if (in_array($name, array_keys($allowed))){
 return '</'.$name.'>';
}else{
 return '';
}
```

But if the tag is an self closer, then we never need to output a closing tag. We'll create an array of self closer called $no_close and make sure it's passed through to the tag processor. The new end tag generator looks like this:

```
if (in_array($name, array_keys($allowed))){
 if (!in_array($name, $no_close)){
  return '</'.$name.'>';
 }
}else{
 return '';
}
```

So now we just need to make sure that self closing tags always self close. We can do this with a simple alteration just before we output the opening tag. The current code:

```
return '<'.$name.$params.$ending.'>';
```

$ending might contain a closing slash - since normal tags can self close if they fancy. If the tag is always a self closer, then we can just rig the value in $ending at this point:

```
if (in_array($name, $no_close)){
 $ending = ' /';
}
return '<'.$name.$params.$ending.'>';
```

Great, now we've taken care of the simple tags, we can look at tags that open and close with stuff in between them. We have two bugs we need to plug here - people closing tags they didn't open, and people opening tags and not closing them. It turns out we can actually fix both problems with the same fix - keeping a running count of what tags are open. We start the process assuming there are no tags of any kind open. When we encounter an opening tag, we increment the count (assuming it's not self closing). When we encounter a closing tag, we decrement the count.

```
before outputting a closing tag:
$tag_counts[$name]--;

before outputting an opening tag:
if (!$ending){
 $tag_counts[$name]++;
}
```

The $tag_counts array is a global which is initialised (emptied) before the process starts.

So now we have a running count of tags, we can tackle the two issues. Firstly, people closing tags that they didn't open. Well, if we are about to output an opening tag, the count for it must be above zero. If it's zero, then the tag was never opened (or opened and already closed), so we needn't output it:

```
if ($tag_counts[$name]){
 $tag_counts[$name]--;
 return '</'.$name.'>';
}
```

Secondly, tags might be open after we've finished parsing. If they've been opened and not closed, they'll still have a positive count. So all we need to do is loop through the tag counts, outputting closing tags after the calls to process_tag(). Easy enough:

```
foreach(array_keys($tag_counts) as $tag){
 for($i=0; $i<$tag_counts[$tag]; $i++){
  $data .= "</$tag>";
 }
}
```

So now we can guarantee that are tags are balanced. Except we don't always know where the tags

really are..

Tag Formation

There's another class of error/exploit that our code doesn't catch, and it's pretty easy to find:

```
<img src="foo"
```

...which will hide everything up to the next >. Annoying, but not fatal. But this can be:

```
<<script>script<script>>
```

Your trusty filtering code strips out the script tags, and, oops, makes a new script tag. Not a good thing. We can combat this using a similar technique to tag balancing. I'll call it bracket balancing :)

First of all, we can correct the inner errors. That is, errors that occur without the html (rather than at the boundries, which are a special case). We do this using a pair of regular expressions:

```
$data = preg_replace("/<([^>]*?)(<)/", "<$1><", $data);
$data = preg_replace("/>([^<]*?)(>)/", ">$1<>", $data);
```

The first matches an opening bracket, a sequence of characters that aren't opening or closing brackets, and then another opening bracket. This would match, for example, "<a<". It then adds in the missing closing bracket after the content of the tag (but before the next opening bracket). The second regular expression does the reverse (adds missing starting brackets just before extra closing ones).

It turns out this isn't exactly what we want. Consider the following:

```
<<<foo>
```

Here the expression matches the first two opening brackets, and adds a closing bracket between them. Next it moves forwards and can find no more matches. We want it to be able to move forward and start before the second opening bracket. But how do we do that? With a zero width positive look-ahead assertion - it matches part of the string but doesn't move the matching cursor forwards:

```
$data = preg_replace("/<([^>]*?)(?=<)/", "<$1>", $data);
$data = preg_replace("/>([^<]*?)(?=>)/", ">$1<", $data);
```

Now back to those edge cases i mentioned. These involve open tags that start or end the edges of the input. For example:

```
b>foo...
...foo<b
```

Our regular expressions don't match these, but a couple of changes and they will. The "boundaries" of the expressions are currently start start of the next tag (expression 1) or the end of the previous tag (expression 2). If we change these boundary conditions to also include the start and end of the data, then the expressions will work in all cases. So that's what we'll do:

```
$data = preg_replace("/<([^>]*?)(?=<|$)/", "<$1>", $data);
```

```
$data = preg_replace("/(^|>)([^<]*?)(?=>)/", "$1<$2", $data);
```

Now our data all always have perfectly balanced brackets, even if they don't contain anything ("<<>"). This allows our tag processor to catch all tags, and the only brackets that can be output are by the start and end tag sections in the tag processor - we now have complete control of the characters needed to output tags.

But using forbidden tags isn't the only problem...

Protocol Filtering

In our original code, we had the following line:

```
$data = str_replace('javascript:','#',$data);
```

This helped strip out scripts that ran inline. Unfortunately, this is far from a full fix. Any of the following also work, depending on browser:

```
<a href="javascript:foo">bar</a>
<a href="java script:foo">bar</a>
<a href="jscript:foo">bar</a>
<a href="vbscript:foo">bar</a>
```

And also the less horrific, but also annoying:

```
<a href="about:foo">bar</a>
<a href="view-source:foo">bar</a>
<a href="aim:foo">bar</a>
```

...to name but a few. The real worry are the script tags, and it gets more complicated when we take IE5 into account - because it allows any sequence of whitespace or control characters between "java" and "script".

What we're going to have to do here, is take the same approach as with tags: there are lots of "bad" protocols, and more are being created all the time. What we *do* know, is the ones that *aren't* bad. So we'll build a mechanism that only allows a certain list of protocols, otherwise uses the standard hash method to disable it.

We'll also need a list of paramamters that contain protocols - we don't want to get filtering stuff out of an image's alt text. So we have two lists:

```
$protocol_attributes = array(
 'src',
 'href',
);

$allowed_protocols = array(
 'http',
 'ftp',
 'mailto',
);
```

We might have other elements in the attributes list, such as "dynsrc" for images, but remember we only need to filter attributes which we aren't already filtering out via the $allowed tags hash.

All we need to do then, is catch the relevant attributes and process their values. We can do this just as the attributes are serialised:

```
$params .= " $pname=\"$match[2]\"";
```

By adding a call to an external routine we'll write next:

```
$value = $match[2];
if (in_array($pname, $protocol_attributes)){
 $value = process_param_protocol($value, $allowed_protocols);
}
$params .= " $pname=\"$value\"";
```

The function needs to check the protocol at the start of the parameter (if any) and replace it with a '#' if it's not on the allowed list. Very straight forward:

```
function process_param_protocol($data, $allowed_protocols){

 if (preg_match("/^([^:]+)\:/i", $data, $matches)){
  if (!in_array($matches[1], $allowed_protocols)){
   $data = '#'.substr($data, strlen($matches[1])+1);
  }
 }

 return $data;
}
```

But, as usual, that's not quite the whole story. Some browsers allow carriage returns in the middle of tags, and particularly of attributes. The tag matcher doesn't match tags that span more than one line, so even though the brackets are balanced, the matcher ignores tag halves across two or more lines. The usually results in broken html which the browser ignores. But not when it's a carriage return inbetween the words 'java' and 'script'. Oh. So this will get through, unfiltered:

```
<a href="java
script:foo">bar</a>
```

The solution is very easy - we just start to treat a carriage return as we would with any other character, by adding the /s flag to the tag parsing regular expressions.

With these changes in place, the library filters most problems out of HTML. I say most, because people are always finding ingenious new ways of breaking things. Looking at what's going on is essential - don't assume your users wont find away around your filters.

The full code

The full code for lib_filter can be downloaded here. The version discussed in this article is shown below.

```
 $tag_counts = array();

function filter_html($data){
 global $tag_counts;

 $allowed = array(
  'a' => array('href', 'target'),
  'b' => array(),
  'img' => array('src', 'width', 'height', 'alt'),
 );

 $no_close = array(
  'img',
 );

 $protocol_attributes = array(
  'src',
  'href',
 );

 $allowed_protocols = array(
  'http',
  'ftp',
  'mailto',
 );

 $tag_counts = array();

 $data = balance_html($data);
 $data = check_tags($data, $allowed, $no_close,
  $protocol_attributes, $allowed_protocols);

 return $data;
}


function balance_html($data){

 $data = preg_replace("/<([^>]*?)(?=<|$)/", "<$1>", $data);
 $data = preg_replace("/(^|>)([^<]*?)(?=>)/", "$1<$2", $data);

 return $data;
}


function check_tags($data, $allowed, $no_close,
 $protocol_attributes, $allowed_protocols){

 global $tag_counts;

 $data = preg_replace("/<(.*?)>/se",
```

```
   "process_tag(StripSlashes('\\1'), \$allowed, \$no_close,
    \$protocol_attributes, \$allowed_protocols)",
   $data);
  $data = str_replace('javascript:','#',$data);

  foreach(array_keys($tag_counts) as $tag){
   for($i=0; $i<$tag_counts[$tag]; $i++){
    $data .= "</$tag>";
   }
  }

  return $data;
}


function process_tag($data, $allowed, $no_close,
 $protocol_attributes, $allowed_protocols){

 global $tag_counts;

 # ending tags
 if (preg_match("/^\/([a-z0-9]+)/si", $data, $matches)){
  $name = StrToLower($matches[1]);
  if (in_array($name, array_keys($allowed))){
   if (!in_array($name, $no_close)){
    if ($tag_counts[$name]){
     $tag_counts[$name]--;
     return '</'.$name.'>';
    }
   }
  }else{
   return '';
  }
 }

 # starting tags
 if (preg_match("/^([a-z0-9]+)(.*?)(\/?)$/si", $data, $matches)){
  $name = StrToLower($matches[1]);
  $body = $matches[2];
  $ending = $matches[3];
  if (in_array($name, array_keys($allowed))){
   $params = "";
   preg_match_all("/([a-z0-9]+)=\"(.*?)\"/si", $body,
    $matches_2, PREG_SET_ORDER);
   preg_match_all("/([a-z0-9]+)=([^\"\s]+)/si", $body,
    $matches_1, PREG_SET_ORDER);
   $matches = array_merge($matches_1, $matches_2);
   foreach($matches as $match){
    $pname = StrToLower($match[1]);
    if (in_array($pname, $allowed[$name])){
     $value = $match[2];
```

```
     if (in_array($pname, $protocol_attributes)){
      $value = process_param_protocol($value,
       $allowed_protocols);
     }
     $params .= " $pname=\"$value\"";
    }
   }
   if (in_array($name, $no_close)){
    $ending = ' /';
   }
   if (!$ending){
    $tag_counts[$name]++;
   }
   if ($ending){
    $ending = ' /';
   }
   return '<'.$name.$params.$ending.'>';
  }else{
   return '';
  }
 }

 # garbage, ignore it
 return '';
}

function process_param_protocol($data, $allowed_protocols){

 if (preg_match("/^([^:]+)\:/si", $data, $matches)){
  if (!in_array($matches[1], $allowed_protocols)){
   $data = '#'.substr($data, strlen($matches[1])+1);
  }
 }

 return $data;
}
```

One last thing

There's something i've glossed over in this article and the one before it. How do we know that making each of these fixes isn't reintroducing a hole we patched early? Easy - regression tests. When an exploit is found, it's added to the test suite, and the code is changed to try and fix the bug without allowing any of the old bugs. The test quite simply calls the filter routine with a set of purposefully broken input and checks for the expected output. It's included here to pursuade you that setting up regression tests is easy - and ultimately saves you lots of time and trouble:

```
 # basics
filter_harness("","");
filter_harness("hello","hello");
# balancing tags
filter_harness("<b>hello","<b>hello</b>");
```

```
filter_harness("hello<b>","hello<b></b>");
filter_harness("hello</b>","hello");
filter_harness("hello<b/>","hello<b />");
filter_harness("<b><b><b>hello","<b><b><b>hello</b></b></b>");
filter_harness("</b><b>","<b></b>");
# end slashes
filter_harness('<img>','<img />');
filter_harness('<img/>','<img />');
filter_harness('<b/></b>','<b />');
# balancing angle brakets
filter_harness('<img src="foo"','<img src="foo" />');
filter_harness('b>','<b></b>');
filter_harness('<img src="foo"/','<img src="foo" />');
filter_harness('>','');
filter_harness('foo<b','foo<b></b>');
filter_harness('b>foo','<b>foo</b>');
filter_harness('><b','<b></b>');
filter_harness('b><','<b></b>');
filter_harness('><b>','<b></b>');
# attributes
filter_harness('<img src=foo>','<img src="foo" />');
filter_harness('<img asrc=foo>','<img />');
filter_harness('<img src=test test>','<img src="test" />');
# non-allowed tags
filter_harness('<script>','');
filter_harness('<script','');
filter_harness('<script/>','');
filter_harness('</script>','');
filter_harness('<script woo=yay>','');
filter_harness('<script woo="yay">','');
filter_harness('<script woo="yay>','');
filter_harness('<script woo="yay<b>','<b></b>');
filter_harness('<script<script>>','');
filter_harness('<<script>script<script>>','script');
filter_harness('<<script><script>>','');
filter_harness('<<script>script>>','');
filter_harness('<<script<script>>','');
# bad protocols
filter_harness('<a href="http://foo">bar</a>', '<a href="http://foo">bar</a>');
filter_harness('<a href="ftp://foo">bar</a>', '<a href="ftp://foo">bar</a>');
filter_harness('<a href="mailto:foo">bar</a>', '<a href="mailto:foo">bar</a>');
filter_harness('<a href="javascript:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="java script:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="java'."\t".'script:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="java'."\n".'script:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="java'.chr(1).'script:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="jscript:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="vbscript:foo">bar</a>', '<a href="#foo">bar</a>');
filter_harness('<a href="view-source:foo">bar</a>', '<a href="#foo">bar</a>');
# auto closers
filter_harness('<img src="a">', '<img src="a" />');
```

```
filter_harness('<img src="a">foo</img>', '<img src="a" />foo');
filter_harness('</img>', '');


function filter_harness($in, $out){
 global $tests;

 $tests[filter]++;
 $got = filter_html($in);
 basic_harness($in, $out, $got, "Filter test $tests[filter]");
}

function basic_harness($in, $out, $got, $name){
 global $verbose;

 echo "$name : ";
 if ($out == $got){
  echo "<span style=\"color: green;\">pass</span>";
 }else{
  echo "<span style=\"color: red; font-weight: bold;\">fail</span>";
 }
 if ($verbose || ($out != $got)){
  echo " (<b>in:</b> ".htmlentities($in)." <b>expected:</b> ";
  echo htmlentities($out)." <b>got:</b> ".htmlentities($got).")";
 }
 echo "<br>\n";
}
```

The full code for lib_filter, including a demo and test suite can be downloaded here.

(end of article)