



# Pixel Perfect Timing Attacks with HTML5

---

**Paul Stone**

[whitepapers@contextis.co.uk](mailto:whitepapers@contextis.co.uk)

**July 2013**



## Contents

<b>Abstract</b>	<b>3</b>
<b>Thinking in Frames - using requestAnimationFrame to time browser operations</b>	<b>4</b>
requestAnimationFrame and Timing Attacks	5
Rendering Links and Redraw Events	6
Detecting Redraw Events	7
Calibration	10
Results and Practicality	11
Visibility	11
Speed	11
Reliability	12
<b>CSS, SVG and Filters</b>	<b>13</b>
Timing Attacks with SVG Filters	14
Timing the speed of SVG filters	16
History Sniffing with SVG Filters	17
Reading Pixels	19
Reading Text with Pixel Perfect OCR	21
<b>Conclusion</b>	<b>24</b>
<b>Security Bug Reports and Disclosure</b>	<b>25</b>
<b>About Context</b>	<b>26</b>
<b>Works Cited</b>	<b>27</b>



## Abstract

This paper describes a number of timing attack techniques that can be used by a malicious web page to steal sensitive data from a browser, breaking cross-origin restrictions. The new requestAnimationFrame API can be used to time browser rendering operations and infer sensitive data based on timing data. The first technique allows the browser history to be sniffed by detecting redraw events. The second part of the paper shows how SVG filters are vulnerable to a timing attack that can be used to read pixel values from a web page. This allows pixels from cross-origin iframes to be read using an OCR-style technique to obtain sensitive data from websites.



## Thinking in Frames - using requestAnimationFrame to time browser operations

The `requestAnimationFrame` JavaScript API is a recent addition to browsers and was designed to allow web pages to create smooth animations. `requestAnimationFrame` takes a single parameter, a function that will be called back just before the next frame is painted to screen [1]. The callback function will be passed a timestamp parameter that tells it when it was called.

```
var handle = window.requestAnimationFrame(callback);
```

The callback mechanism provides a way for developers to perform necessary tasks between animation frames. Within each frame of animation on a web page, a number of tasks may be carried out by the browser. These include executing JavaScript code, calculating the position of new and updated elements (known as layout or reflow) and drawing elements to screen. Each of these tasks may take a variable amount of time. For example, if the JavaScript code inserts a large number of new elements into a page, then the layout step may take a long time. If an element has complex styles applied to it, such as shadow or transparency then the painting step may take more time than simple operations such as moving an element across the screen.

The diagram on the right represents two consecutive frames of animation in which the page layout is adjusted. The timestamp passed to the `requestAnimationFrame` callback can be used to calculate  $t1$  and  $t2$ , the amount of time taken for both frames.

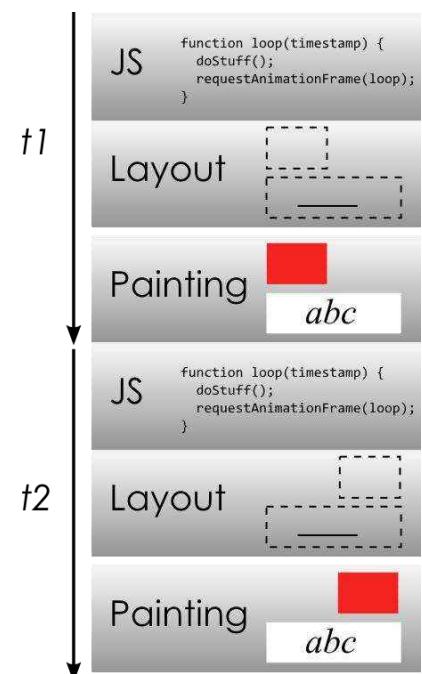
If called repeatedly, `requestAnimationFrame` will aim to paint up to 60 frames per second (i.e. every 16 milliseconds) and will schedule the callback function accordingly. If the total time taken by the code execution, layout and painting steps is longer than 16ms, then the next frame will be delayed until these tasks have completed. This delay will be measurable by `requestAnimationFrame`, allowing the frame rate to be measured.

The following code shows how `requestAnimationFrame` can be used to calculate the 'frame rate' of a web page, by calculating the time elapsed between each frame.

```
var lastTime = 0;
function loop(time) {
    var delay = time - lastTime;
    var fps = 1000/delay;
    console.log(delay + ' ms' fps);
    updateAnimation();
    requestAnimationFrame(loop);
    lastTime = time;
}
requestAnimationFrame(loop);
```

**Figure 1**

An example call to `requestAnimationFrame`

**Figure 2 - Browser rendering phases****Figure 3**

Measuring frame rate with `requestAnimationFrame`



## requestAnimationFrame and Timing Attacks

By synchronising JavaScript code execution with the browser's layout and rendering routines, `requestAnimationFrame` allows a web page to measure browser performance in a way that was previously difficult. In order for timing attacks to be viable certain criteria must be met by the browser function under consideration. Specifically, these criteria are:

- i) can be triggered from JavaScript
- ii) must take place during the layout or painting steps
- iii) should process sensitive data that is not directly readable by JavaScript
- iv) must take a variable amount of time depending on the data it is provided

Any browser operation that satisfies these criteria may be susceptible to a timing attack that could reveal sensitive information stored in a malicious web page.



## Browser History Sniffing

In the past it was possible to abuse a number of features in CSS and JavaScript to determine whether a given URL had been previously visited by a user. For example, the `getComputedStyle` function could be used to check the colour of a link, making it trivial for a webpage to distinguish between a visited and unvisited link. By checking a large number of URLs, a malicious website could 'sniff' a user's browsing history and build a list of previously visited websites.

These techniques were shown to be very effective, enabling a malicious site to check thousands of URLs per second to see if a user had visited them [2]. Far from being theoretical, research showed that the technique was being actively abused by number of websites to sniff the browsing history of their users [3].

In 2010 David Baron published a proposal for preventing such attacks [4], by restricting the styles that can be applied to visited links in CSS and ensuring that JavaScript API calls that query element styles always behave as if a link is unvisited. These fixes have since been implemented in all major browsers.

Since then, a few techniques have been shown that work around the new restrictions. These work by styling links in clever ways, such as CAPTCHAs or game pieces and rely on the user to visually identify visited links by clicking on them [5] [6]. While these can be effective, they require user interaction and can be slow.

In this section we describe a number of new techniques that can automatically sniff browser history and work without user interaction.

## Rendering Links and Redraw Events

When a web page contains one or more links, the browser must determine whether to render each link using the visited or unvisited visual style. Every browser has a database of previously visited URLs (i.e. the browser history) and will perform a lookup in this database to see if each URL has been visited before.

Both Internet Explorer and Firefox perform an asynchronous database lookup for each link. If the database query has not completed before the link is rendered, the browser will first use the 'unvisited' style to render the link. Once the database query has returned, the link will be redrawn if the URL is found to have been previously visited. If this 'redraw' event can be detected by a webpage, then it could tell if the link had been previously visited.

The Chrome web browser appears to perform synchronous database lookups when rendering links. Unlike Firefox and Internet Explorer, Chrome will wait until the database URL lookup has completed before rendering the link on-screen.

Aside from the initial rendering of a link, another time a browser may redraw a link is if its target, i.e. `href`, is changed to point to a different URL by JavaScript on the web page. Testing showed that this worked as expected in Firefox; changing a link's `href` would re-render the link if the 'visited' state had changed. This doesn't work in Internet Explorer – once a link has been created, changing its `href` will never cause the 'visited' state of the link to change.

<https://www.facebook.com>  
<http://www.google.com>  
<http://www.youtube.com>  
<https://www.twitter.com>  
<https://www.linkedin.com>  
<http://www.craigslist.org>  
<http://stackoverflow.com>  
<http://www.bing.com>  
<http://www.bbc.co.uk>  
<http://www.microsoft.com>  
<http://www.amazon.com>  
<http://www.mozilla.org>  
<http://www.contextis.co.uk/>  
<http://www.theregister.co.uk>  
<http://www.reddit.com>  
<http://news.ycombinator.com>

**Figure 4** – the default styling of visited and unvisited links. This information should be visible to a user, but not to the web page.



In Chrome, changing the href alone did not cause the link to be redrawn. However, by changing the href and then 'touching' but not altering the style of the link element, the browser would repaint the link if the new URL required caused the visited state to change.

```
<a href="http://www.google.com" id="link1">#####</a>

<script>
var el = document.getElementById('link1');
el.href = 'http://www.yahoo.com';

// below lines are required for Chrome to force style update
el.style.color = 'red';
el.style.color = '';
</script>
```

**Figure 5**  
Changing a link href in Chrome and Firefox

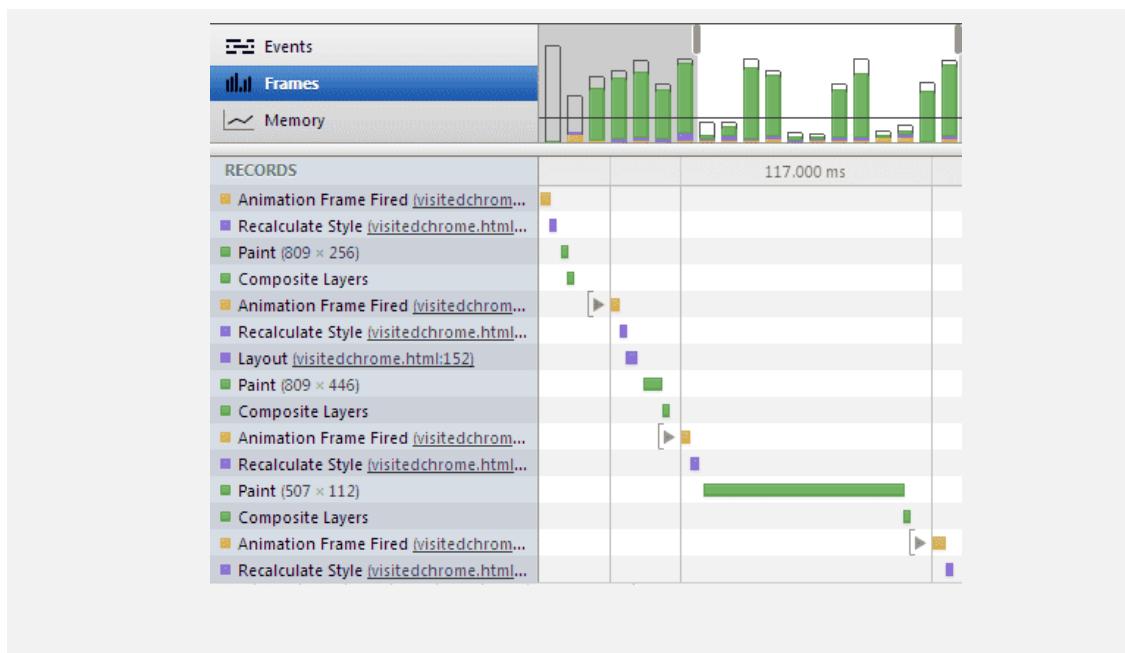
The table below summarises how links are repainted in three major browsers:

Browser	Asynchronous URL lookup	Changing href of existing link
Chrome 27		✓
Firefox 21	✓	✓
Internet Explorer 10	✓	

**Table 1**  
Events that cause link repainting

## Detecting Redraw Events

In the past, Firefox allowed a web page to be directly notified of redraw events, via the mozAfterPaint [7]. This event notified a page not only that a paint event had occurred, but which regions had been painted too. mozAfterPaint was however disabled by default due to the possibility of visited information leaking to web pages [8] [9].



**Figure 6**  
Animation and Paint events shown in Chrome's web inspector



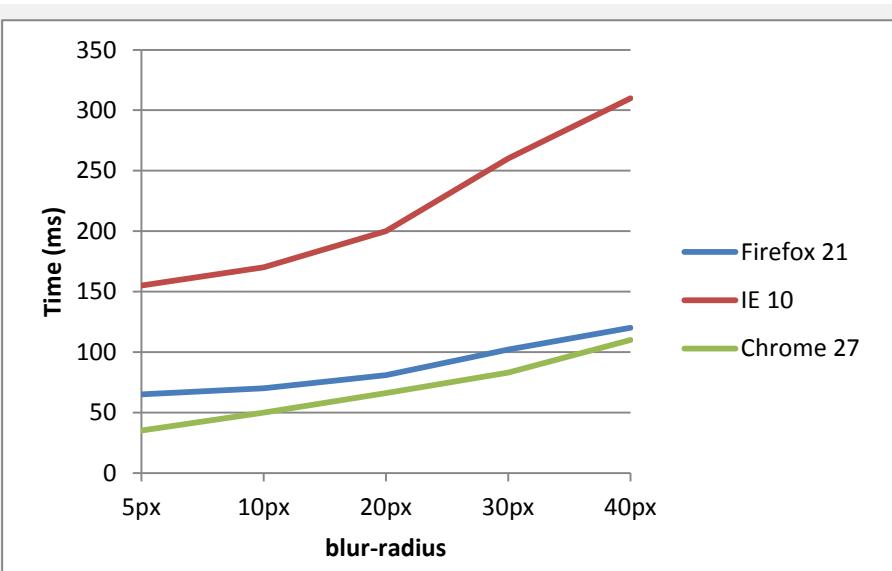
By using the `requestAnimationFrame` API, it is possible to detect when links are repainted. However, today's browsers are highly optimised, with many graphical operations offloaded to the GPU. If a repaint event takes less than 16ms to complete, then the period between callbacks will not change (i.e. the frame rate will not drop below 60fps). We therefore need to find a way to slow down link rendering to make redraws detectable.

The CSS `text-shadow` property [10] allows various effects to be applied to text on a web page, including drop shadows, glows and embossing. The amount of time to render an element with `text-shadow` applied is proportional to the value of the `blur-radius` property. Rendering can be further slowed down by applying multiple shadows to text and increasing the amount of text rendered. Experimentation has shown that this is an effective technique in Chrome, Firefox and Internet Explorer.

The below table compares rendering times of elements with `text-shadow` applied in different browsers. For the purposes of the test, 50 link elements were drawn, each pointing to the same URL. The test was run on an Intel i5-2520M processor.

blur-radius	Firefox 21	Internet Explorer 10	Chrome 27
5px	65ms	155ms	35ms
10px	70ms	170ms	50ms
20px	81ms	200ms	66ms
30px	102ms	260ms	83ms
40px	120ms	310ms	110ms

**Table 2**  
Timings of redraw events for links with `text-shadow` applied



**Figure 7**  
Graph showing redraw times for various blur-radius values



By applying our knowledge of how link repaints occur in Firefox and Internet Explorer, the following steps can be used to detect visited links:

### Method 1 - Asynchronous URL Lookup

- Insert a number of links on page with text-shadow applied, all pointing to same URL
- Time the next N frames using requestAnimationFrame
- Analyse captured frames to determine if redraw occurred

The following screenshot from Internet Explorer shows a demo of the redraw timing technique. For each URL a number of links were inserted onto the page, with text-shadow applied. The numbers show the number of milliseconds that each of the 5 subsequent frames took to draw, according to requestAnimationFrame. The timing of the first frames shows the links being drawn in their 'unvisited' state. The highlighted frames show when the redraws occur for visited links:

<input checked="" type="checkbox"/> <a href="https://www.facebook.com">https://www.facebook.com</a>	168, 1, 159, 0, 6
<input checked="" type="checkbox"/> <a href="http://www.google.com">http://www.google.com</a>	171, 1, 11, 160, 0
<a href="http://www.youtube.com">http://www.youtube.com</a>	148, 1, 7, 14, 16
<a href="https://www.twitter.com">https://www.twitter.com</a>	154, 2, 10, 16, 16

**Figure 8**  
Timing repaints in Internet Explorer

The green ticks show that our demo code has detected a link repaint, indicating that the links have been visited.

The same technique works almost identically in Firefox. This time the first two frames are slow, with a third slow frame indicating a repaint of visited frames.

<input checked="" type="checkbox"/> <a href="https://www.facebook.com">https://www.facebook.com</a>	106, 155, 186, 3, 16
<input checked="" type="checkbox"/> <a href="http://www.google.com">http://www.google.com</a>	95, 152, 193, 11, 16
<a href="http://www.youtube.com">http://www.youtube.com</a>	93, 143, 15, 26, 7
<a href="https://www.twitter.com">https://www.twitter.com</a>	126, 150, 8, 16, 16

**Figure 9**  
Timing repaints in Firefox

In Chrome, the links are not repainted after they are initially drawn, as shown by the same demo running in that browser. The first frame after links are inserted takes the same amount of time regardless of whether a link is visited.

<a href="https://www.facebook.com">https://www.facebook.com</a>	165, 16, 16, 16, 16
<a href="http://www.google.com">http://www.google.com</a>	167, 16, 16, 16, 18
<a href="http://www.youtube.com">http://www.youtube.com</a>	165, 16, 16, 16, 16
<a href="https://www.twitter.com">https://www.twitter.com</a>	168, 16, 17, 15, 16

**Figure 10**  
Timing repaints in Chrome



The algorithm can be modified however to take advantage of the behaviour in Chrome when a link is target is changed:

### Method 2 - Changing Link Target

- Place a number of link elements on the web page, pointing to a known-unvisited URL
- Update the links to point to the URL to be tested.
- Time the next frame using requestAnimationFrame

Because history lookups are not asynchronous in Chrome only a single frame needs to be timed with requestAnimationFrame. The repaint will occur immediately after the link is changed if the new URL is visited.

## Calibration

The timing of the redraws will differ depending on the speed of the hardware on which the browser is running. To allow the same history sniffing code to run on different hardware with good results, a calibration step must first be done. In this step, the code attempts to find suitable values for the following variables:

- A blur-radius value for the text-shadow property (B)
- The number of links to draw (N)

The aim is to find values that cause the link painting to be slow enough to time with requestAnimationFrame, but quick enough to allow a large number of links to be checked.

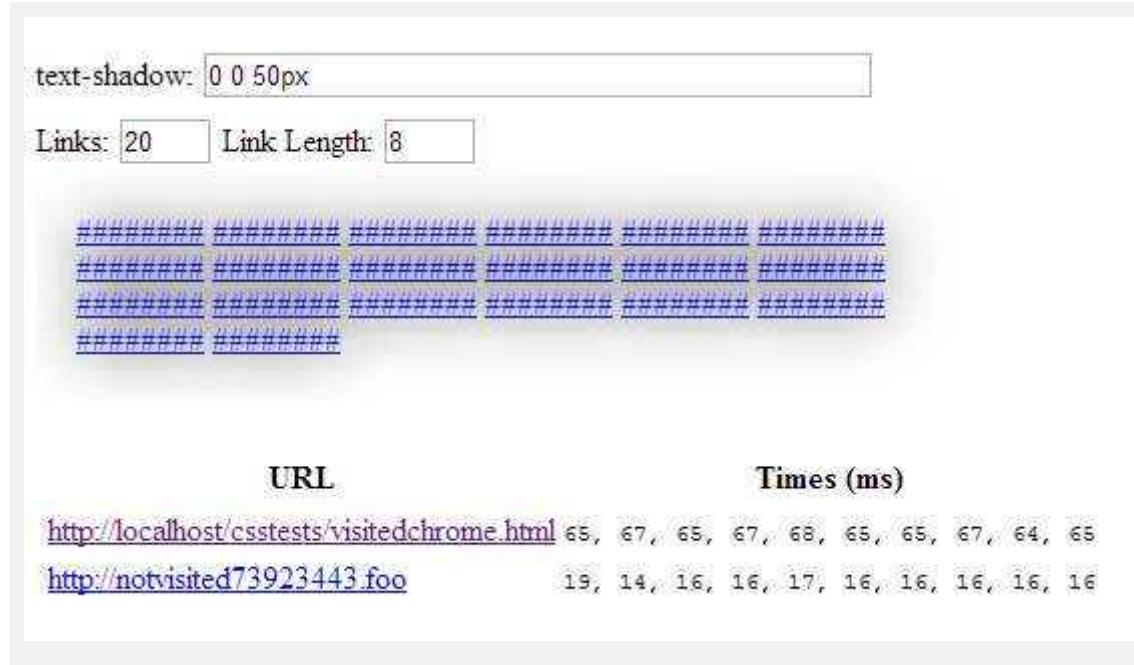
The following algorithm can be used in Chrome. Firefox and IE would require slightly more complex steps due to the asynchronous database lookups in those browsers.

### Method 3 – Calibrating text-shadow

- Start with small values for B and N (e.g. a 10px blur and 20 links)
- Time 10 frames with links that are visited
- Time 10 frames with unvisited links
- If the average timing difference between the visited and unvisited frames is under a certain threshold (e.g. 50ms) increase B and N, then repeat calibration
- Once the desired timing has been reached, calibration is complete.



The following screenshot shows the results of the calibration steps in the demo code in Chrome:



**Figure 11**  
Results of automatic calibration in Chrome

## Results and Practicality

### Visibility

A real-world malicious page that wished to perform this history sniffing attack would not want to display the links as shown in the above screenshot. However, actually showing the links is not necessary. CSS Transforms [11] allow web page elements to be scaled and rotated. As transforms are applied after an element has been drawn off-screen, they do not affect the speed that effects such as text-shadow take to apply. The following CSS can be applied to the links used to perform the timing attack, making them effectively invisible.

```
#link-container { transform-origin: 0 0; transform: scale(0.01) }
```

**Figure 12**  
Hiding elements with CSS Transforms

### Speed

The speed of this technique is limited by the maximum frame rate of the browser as we must wait for a frame to be drawn before it can be timed with requestAnimationFrame. In Firefox and Internet Explorer, due to the asynchronous history lookups, we must wait between 3 and 5 frames to see if links are redrawn. This gives a rough limit of between 13 and 20 URL checks per second.

In Chrome, a URL can be tested by every other frame, putting the theoretical limit at 30 URL checks per second. However, each frame in which a visited URL is found will be slower to render, reducing the overall frame rate.

There are a number of optimisations and improvements that could be made to these basic algorithms. One such optimisation is to check a large number of different URLs at once; if a



redraw is detected then we know that at least one of these URLs is visited. This leads us to the following algorithm:

#### Method 4 – Binary chop history sniffing

- Place a large number of different URLs (e.g. 1000) on the page at once
- If a redraw is detected, divide the URLs into two sets
- To each set add dummy unvisited URLs to keep the total number at 1000. This will ensure the timing remains constant.
- Separately test each new set of URLs for redraws. If no redraw is detected, discard the URL set.
- Continue testing, dividing and discarding until unique URLs are found.

This method will work best with a large set of URLs in which only a small portion are actually visited. This technique was implemented in Internet Explorer. A simple test of 1000 URLs with 10 visited links completed in 17 seconds, a speed of 58 URLs tested per second.

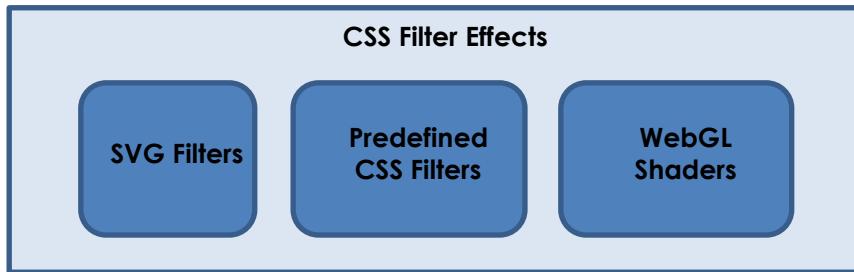
#### Reliability

As with any timing attack, the reliability of this exploit is highly dependent on the quality of the gathered timing data. Any background tasks that are running such as other processes or tabs can slow down the rendering of the page performing the timing attack, leading to false positives. A real world attack could use `requestAnimationFrame` to measure the frame rate of the current web page, to detect when the user's computer has become idle. For example, when a web page is initially loading, the frame rate will drop as network resources are loaded and the page is rendered. When the frame rate stabilises at 60fps, the malicious page could begin the timing attack.



## CSS, SVG and Filters

Modern browsers are continually gaining features that allow web pages to create rich interactive graphics. As seen in previously with text-shadow, complex graphics features can be a good place to look for timing vulnerabilities. Filter Effects [12] is a relatively new specification that allows visual effects to be applied to web pages using CSS properties. It is actually a combination of three quite separate features:



SVG Filters originate from an earlier specification [13] that allowed visual effects to be applied to SVG drawings. The Filter Effects specification incorporates SVG Filters and allows them to be applied to any HTML element using CSS. The SVG filter specification defines 16 basic operations (known as filter primitives) including convolution, colour transforms and image composition. Each of these filter primitives has a number of input parameters that can be set, and primitives can be combined to produce more complex effects such as drop shadows and bump mapping [14] [15].

```
<filter id="greyscale">
  <feColorMatrix type="matrix" values="0.3333 0.3333 0.3333 0 0
                                         0.3333 0.3333 9.3333 0 0
                                         0.3333 0.3333 0.3333 0 0
                                         0       0       0       1 0"/>
</filter>

<style>.f1 { filter: url(#greyscale) }</style>

<div class="f1"> Hello World</div>
```

**Figure 13**  
Applying a simple SVG filter to an HTML element

A number of simple pre-defined filters are also defined by the specification. These allow effects such as greyscale, brightness, contrast and drop shadows to be applied without needing to write out complex SVG filters.

```
<style>.f2 { filter: grayscale(100%) }</style>

<div class="f2"> Hello World</div>
```

**Figure 14**  
Pre-defined CSS Filters

The Filter Effects specification also incorporates the GLSL ES-based shader language from WebGL. This allows complex pixel and vertex shaders to be applied to HTML elements via CSS. Timing attacks using WebGL shaders have been demonstrated by Context previously [16], allowing cross-origin images to be read by a malicious web page. Recent research has showed how a similar technique could be applied to CSS shaders to read cross-origin HTML content [17].



Currently no browser ships with CSS shader support enabled by default. SVG Filters are the most widely supported part of the Filter Effects specification, being supported by the latest versions of Firefox, Chrome and Internet Explorer. Therefore SVG filters are a good place to look for timing vulnerabilities.

	SVG Filter Support	WebGL Shader Filter Support	Pre-defined CSS Filter Support	Can apply filters to link elements?	Can apply filters to any HTML element?
<b>Firefox</b>	Yes	No	No	Yes	Yes
<b>IE10</b>	Yes	No	No	Yes	No ‡
<b>Chrome</b>	Yes	No*	Yes	Yes	Yes

**Table 3**  
Browser support for the Filter Effects specification

\* Experimental support can be enabled via chrome://flags

‡ IE only allows filters can only be applied to SVG content

## Timing Attacks with SVG Filters

Filters are interesting from a security perspective because they can be applied to arbitrary HTML content, including cross-origin iframes and links. Although the latest revision of the Filter Effects specification advises against allowing filters to be applied to cross-origin content, both Firefox and Chrome currently allow it. If an SVG filter (or combination of filters) can be found that take a variable amount of time to apply depending on their input, then a timing attack may be possible.

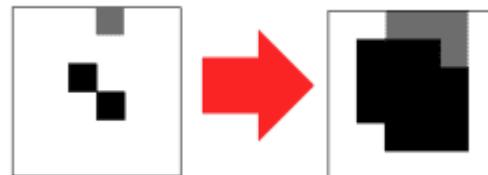
SVG filters in Internet Explorer are hardware accelerated, and run on the GPU. In Firefox and Chrome filters are implemented C++ code; they run relatively slowly as they don't take advantage of the GPU<sup>1</sup>. To speed up filter rendering a number of software optimisations are used in these browsers. Most software optimisations attempt to speed up common cases but will often perform slowly in less-common edge cases. By studying the Firefox source code, a number of filters were identified that would perform quicker on certain types of image.

One filter of particular interest in Firefox was the morphology filter. This filter supports two operations – erosion and dilation. These are typically used in image processing to make edges and lines thinner or thicker. This is implemented by passing a 'kernel' of a certain size (e.g. 3x3 or 5x5) over each pixel of the input image. In a dilation operation, each pixel will be set to the darkest pixel found in its surrounding kernel. The size of the kernel in the SVG morphology filter can be controlled through the radius parameter (r). In a naïve implementation, an input image of  $w * h$  pixels in size will require  $x * h * r * r$  pixels to be scanned in total.

<sup>1</sup> Experimental hardware-accelerated SVG filters are implemented in Chrome. But are turned off by default



**Figure 16** - Applying a dilation morphology operation thickens lines

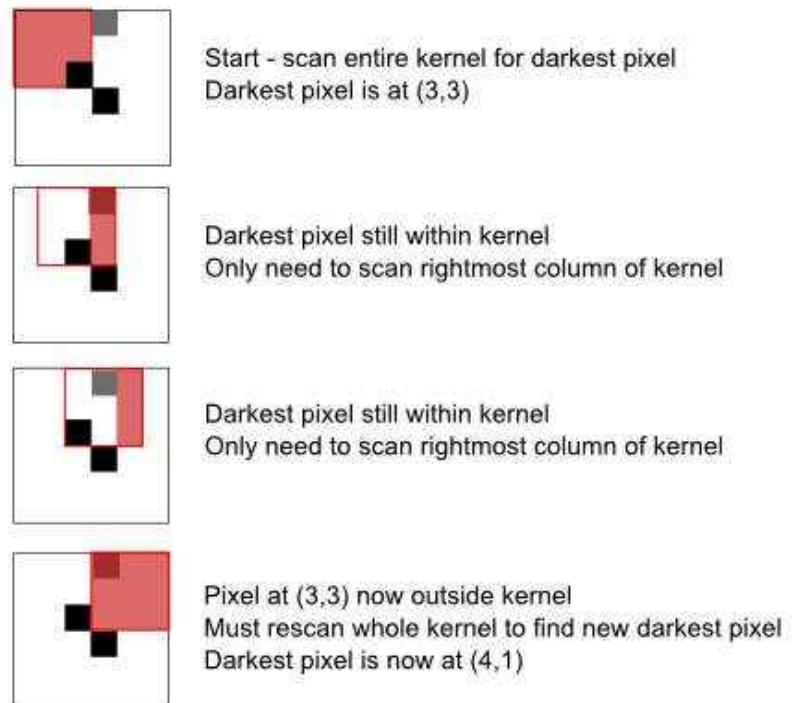


**Figure 15** - Applying a dilation with a 3x3 kernel (radius 3)

However, the Firefox code has an optimisation. As the kernel is moved over the input image from left to right, the algorithm remembers the location of the darkest pixel ( $D$ ). As long as  $D$  remains within the kernel, only the rightmost column of pixels in the kernel need to be scanned for darker pixels - a total of  $r$  pixels. If a pixel is found that is equal to or darker than  $D$  then  $D$  is updated to the location of that pixel. Once the kernel moves beyond  $D$ , the entire  $r * r$  kernel must be scanned again.

This algorithm will perform quickest on an image of uniform colour, since  $D$  will always be updated to be on the right hand side of the kernel, and the whole kernel will never need to be scanned in full. This means that in the best possible case, only  $w * h * r$  pixels will need to be scanned in total. For example, if an erosion filter with a radius of 15 is applied to an image 320x240 in size, then the best case is 1,152,000 comparisons. The worst possible case would be an image where the darkest pixel was always on the leftmost column of the kernel (e.g. a gradient). In this case, 17,280,000 comparisons would be needed in total - 15 times slower.

The diagram to the right shows how the optimised algorithm works.



**Figure 17** - The optimised morphology algorithm in Firefox



## Timing the speed of SVG filters

To test the performance of SVG filters in a web browser, `requestAnimationFrame` can again be used. The following code shows how a simple test harness can be set up to time how long a filter takes to apply to an image:

```
<style>.f { filter: url(#morphology) } </style>

<svg>
  <filter id="morphology">
    <feMorphology operator="erode" radius="30">
  </filter>
</svg>
<script>
var element= document.getElementById('e');
var count = 10;
var times = [];
var lastTime = 0;
loop(t) {
  var diff = lastTime - t;
  lastTime = t;
  if (element.classList.contains('f'))
    times.push(diff);
  element.classList.toggle('f');
  if (count--)
    requestAnimationFrame(loop);
}
requestAnimationFrame(loop);
</script>
```

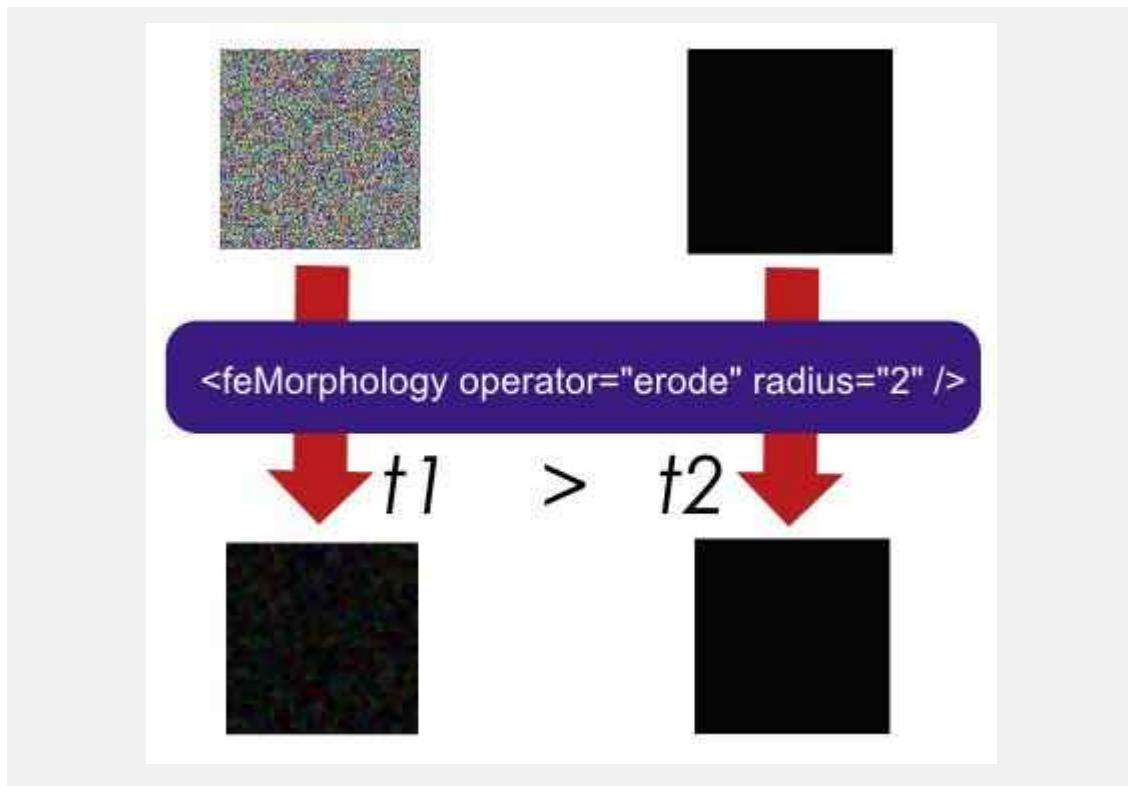
**Figure 18**

Timing the performance of an SVG filter

The above code works by repeatedly toggling the SVG filter on and off in each frame of animation. Each time the filter is turned on, the duration of that frame is recorded. After 10 frames, 5 timings of the filter code will have been recorded. An average can then be calculated.

To compare the performance of the morphology filter, two different input images were provided and then timed with the above code – an image of random noise and an image of flat colour.

By experimenting with the filter parameters and different sized input images a timing difference of over 89ms between the two images was achieved – an average of 35ms for the black image and 124ms for the noisy image. `feMorphology` filter in Chrome was also found to have timing differences, though to a lesser extent. Now that this variable timing has been established, the next step is to use this in some useful way in a timing attack.

**Figure 19**

Two different inputs with the erode filter applied

## History Sniffing with SVG Filters

The example above shows how a noisy image takes longer to filter than a flat image. If we could make a link element appear as a black square if it is unvisited, and a noisy square if it is visited, then we could use the morphology filter to tell the difference. Unfortunately, due to the restrictions on what styles can be applied to visited links, only a few CSS properties can be changed using the `:visited` selector, including color and background-color. However, we can use the following CSS to make a link element appear as either a black square or a white square:

```

<style>
a { width: 100px; height: 100px; background-color: black }
a:visited { background-color: white }
</style>

<a href="http://www.google.com"> </a>
<a href="http://notvisited.asd"> </a>

```

**Figure 20**

Making links appear as black or white squares

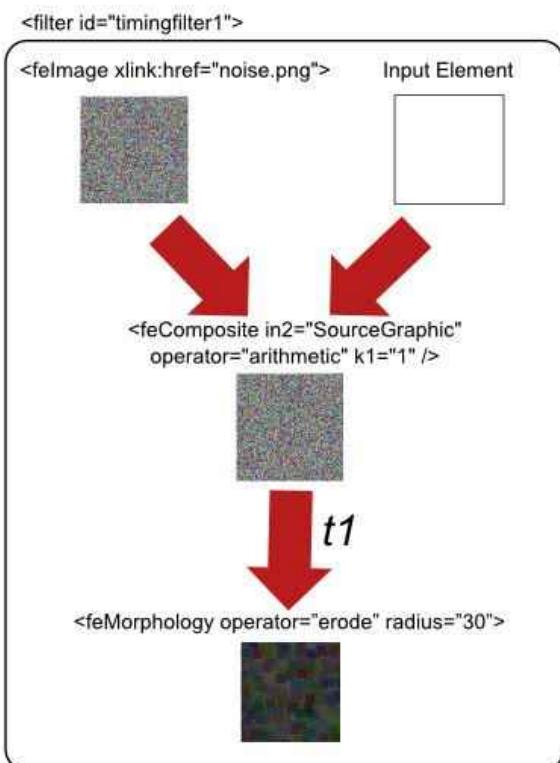
Since visited and unvisited links now differ only in colour, the morphology filter will take the same amount of time on both inputs. However, SVG filters can help us to overcome this. The following SVG filter multiplies the source image with an image of random noise, using the `feImage` and `feComposite` filter primitives. An erosion filter is then applied to the resulting image. The `feComposite` filter takes each pixel in the input image and multiplies its value with the corresponding pixel of the noise image. If the input is black (i.e. 0) the output image will also be black. If the input image is white, the result will be the noisy image.



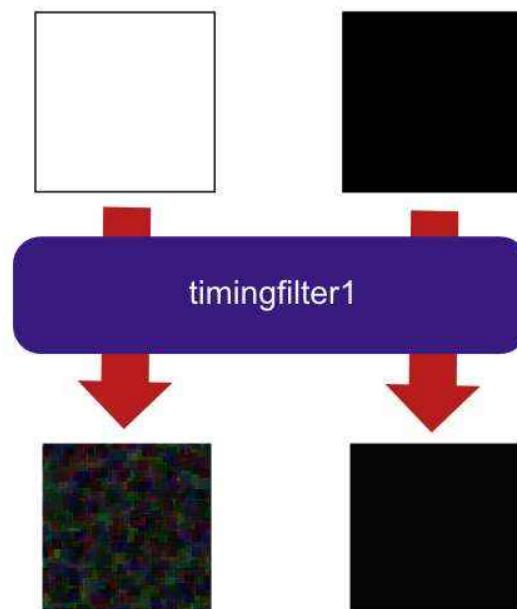
```
<filter id="timingfilter1" filterRes="172">
  <feImage xlink:href="noise.png">
  <feComposite in2="SourceGraphic" operator="arithmetic" k1="1">
    <feMorphology operator="erode" radius="30">
  </filter>
```

**Figure 21**  
A timing attack filter using feMorphology

There are three parameters to this filter that can be adjusted to affect its timing – the erode radius, the value of the filterRes parameter and the size of the input the filter is applied to. The filterRes property can be applied to any SVG filter and sets the resolution at which the filter is applied to the input – the larger the value, the longer the filter will take to compute.



**Figure 23** – The internal steps of the timing filter



**Figure 22** - black and white inputs will take different amounts of time

The timing filter can now be used to tell if a link is visited. To do this, the following method can be used to perform a history sniffing attack.

#### Method 5 – SVG filter history sniffing timing attack

- Apply the filter to a known visited link  $N_{\text{calib}}$  times, and calculate the average time  $T_{\text{pos}}$
- Apply the filter to a known unvisited link  $N_{\text{calib}}$  times, and calculate average time  $T_{\text{neg}}$
- Apply the filter to an unknown link  $L_1$ ,  $N_{\text{test}}$  times, and calculate the average time  $T_1$
- If  $T_1$  is closer to  $T_{\text{pos}}$  than  $T_{\text{neg}}$  then the link is visited
- Repeat for all links to be tested –  $T_2, T_3 \dots$



The first two steps are calibration steps.  $N_{calib}$  can be set to something like 10. Once calibration has been completed, then  $N_{test}$  can be set to a lower number (e.g. 2 or 3). A lower number for  $N_{test}$  will speed up the sniffing attack at the cost of accuracy – the lower the number tests, the higher the number of false positives or negatives will be.

When developing a history sniffing proof of concept using SVG filters, it was found that the timing was being affected by the link repainting behaviour described in the first part of this paper. Each time the filter was applied and removed Firefox would perform a lookup of the link URL and restyle the link if it was visited. This caused the filter to be repainted more often than expected, meaning that the timings were not accurate.

To overcome this, a custom Firefox CSS property called `-moz-element` was used [18]. `-moz-element` allows a 'snapshot' of an HTML element on a page to be used as the background image for another element on the web page. Essentially, one element can copy the appearance of another, but not its behaviour. `-moz-element` can be used to 'reflect' the appearance of the original link. The SVG filter can then be applied to this reflection, preventing Firefox from redrawing the original link element and also preventing any URL lookups that may affect the timing of the filter<sup>2</sup>.

```
<style>
a { background-color: black; width: 100px; height: 100px }
a:visited { background-color: white }
#reflect { background: -moz-element(#link); width: 100px; height: 100px; }
</style>

<a id="link" href="http://www.google.com"> </a>

<div id="reflect"></div>
```

**Figure 24**

Using `-moz-element` to reflect another element

## Reading Pixels

Now that SVG filters can be used to distinguish a black square from a white square, could this technique be used to read arbitrary pixel values – for example pixels from a cross-origin iframe? In theory, a malicious website could load a site that the user is logged into in an iframe. It could then use the feMorphology timing attack to read one pixel at a time from the iframe, building up an entire image of it. There are, however, a number of obstacles to this:

- Most websites don't just have black and white pixels.
- Filtering a single pixel probably won't be slow enough to time – we want a square of around 100x100 to apply the filter to.

The first obstacle can be solved by applying an SVG filter to the iframe to perform a 'threshold' operation, converting the iframe to monochrome. The following code shows an SVG filter that will perform a threshold operation, making all pixels either white or black.

<sup>2</sup> Chrome does not support `-moz-element`, however it does support the SVG `<pattern>` element which does essentially the same thing for SVG content.



```

<filter id="threshold" color-interpolation-filters="sRGB">
  <feColorMatrix type="matrix"
    values="0.333 0.333 0.333 0 -.16
           0.333 0.333 0.333 0 -.16
           0.333 0.333 0.333 0 -.16
           0     0     0     0 1" />
  <feComponentTransfer>
    <feFuncR type="discrete" tableValues="1 0" />
    <feFuncG type="discrete" tableValues="1 0" />
    <feFuncB type="discrete" tableValues="1 0" />
  </feComponentTransfer>
</filter>

```

**Figure 25**  
Threshold SVG filter

To enlarge pixels from the iframe, CSS transforms can be used. Normally when transforms are used to scale up an image, the resulting image is smoothed to prevent pixellation. However, pixels are exactly what is needed – the input to the timing filter must be either a black or white square. Fortunately, another CSS property helps here - image-rendering [19] allows a web page to select the scaling algorithm used for on elements. Specifying image-rendering: crisp-edges instructs the browser to turn off any smooth scaling algorithms.

```

<style>
#f { border: none }
#zoom {
  background: -moz-element(#f);
  transform: scale(8);
  transform-origin: 0 0;
  image-rendering: -moz-crisp-edges;
}
</style>

<iframe src="http://example.org" id="f"></iframe>
<div id="zoom"></div>

```

**Figure 26**  
Enlarging pixels in an iframe

These techniques, combined with the feMorphology filter timing attack allow a webpage in Firefox to read pixels from an iframe. The following method can be used:

#### Method 6 – Reading pixels from an iframe

- Apply the filter to a known white pixel  $N_{calib}$  times, and calculate average time  $T_{white}$
- Apply the filter to a known black pixel  $N_{calib}$  times, and calculate average time  $T_{black}$
- Load target site into an iframe
- Apply the threshold SVG filter
- Crop the iframe to the first pixel to be read and enlarge 100x
- Apply the filter to the pixel,  $N_{test}$  times, and calculate the average time  $T_1$
- Calculate the colour of the pixel by comparing  $T_1$  to  $T_{black}$  and  $T_{white}$
- Repeat for each pixel in the desired area of the iframe to build up the stolen image

**Figure 27**

Top left: Original iframe  
 Top right: Iframe enlarged 8x  
 Bottom left: Threshold applied  
 Bottom right: A single enlarged pixel from the iframe

The `-moz-element` reflection method is again required for the timing to be accurate. Applying the filter directly to the iframe results in the contents of the iframe being redrawn before the filter is applied, which can give inaccurate results. Using `-moz-element` to make a visual copy of the iframe, means that the filter is only applied to a static bitmap, preventing any additional rendering from taking place.

Shown below is the resulting image that was built up by reading pixels from a cross-domain iframe using this technique. The image was built up in around 60 seconds. A few pixels have been read incorrectly but most are correct.

**Figure 28**

Resulting image that was read using the timing attack

## Reading Text with Pixel Perfect OCR

Building up a picture of an iframe makes for an interesting demo, but it is relatively slow. The example above required hundreds of pixels to be read, and only created an image of a few characters of text. Furthermore, an attacker isn't necessarily interested in stealing a picture. Most sensitive information that would be of interest to an attacker is textual – for example usernames or security tokens. Could an information-stealing technique be created that would recognise text and require fewer pixels to be read? In theory text could be read by only looking at a few pixels of each character, provided enough is known about the font used to display the text.

It is possible to construct a text-recognising algorithm that can efficiently steal data from cross-origin iframes, if a few constraints are applied:

- The font face and size of the text to be read must be known



- The location of the text on the page must be known (e.g. the location of username or email address on a profile page)
- The set characters to be recognised must be limited and known (e.g. is the data to be read numeric, hexadecimal or alphabetic)

A good example of text that is a known font and size is the view-source view in Firefox. A quirk in Firefox allows the source code of a web page to be loaded in an iframe instead of the page itself. It is also possible to automatically scroll to a given line number in the source code, by specifying it in the URL hash:

```
<iframe src="view-source:http://example.com#line28"></iframe>
```

**Figure 29**

## Loading a page's source in an iframe in Firefox

The source code of authenticated web pages often contain information that is sensitive. In addition to a user's personal information, such as their name or email address, the source may also contain security tokens such as CSRF or authentication tokens.

**Figure 30**

The source code of a Google+ comments widget, displayed in an iframe. The source contains the currently authenticated user's Google ID and other profile information.

On Windows, the default font for view source in Firefox is 13px Courier. By analysing this font, an algorithm could determine which pixels must be read in order to recognise a character.

We created a JavaScript-based algorithm to analyse a character set and determine which pixels differentiate each character. The algorithm starts with a given character set in the target font, with the threshold filter applied. In the example below, the character set is lower case hex:

0123456789abcdef

**Figure 31**

The target character set, in the default font used by Firefox in view-source on Windows 7. The same threshold filter has been applied to match how the characters are read in the timing attack.



Next, the algorithm builds up a heat map by stacking the characters on top of each other, counting how many characters use each pixel. In the image below, the red pixels are those used by exactly half of the characters. The highlighted red pixel in the first heat map is used only by characters 4, 7, a, b, c, d, e and f. The second heat map shows the accumulation of those characters and again highlights a pixel that can be read to halve the search space. The algorithm builds up a binary search tree that allows a set of 16 characters to be recognised by reading only 4 unique pixels for each character.

**Figure 32**

Character heat maps for 13px Courier.

The first heat map shows all 16 characters combined. The red pixels are those that are set by exactly half of the characters.

The last heat map shows a pixel that can be used to distinguish between characters 7 and f.

In general a text from a character set of size  $N$  can be read by testing only  $\log_2 N$  pixels for each character. For example a base 64 encoded token could be read by testing 6 pixels for each character.

A proof of concept using this OCR data stealing technique was constructed that could read around 1 hex character per second. Unlike the image reading technique where a few incorrectly read pixels can still give a usable result, the OCR technique requires each pixel read to be correct. If a single pixel is read incorrectly then the wrong path through the binary tree will be taken, resulting in an incorrect character being read. This can be prevented by taking several readings for each pixel, improving the accuracy of the read text.



## Conclusion

This paper has demonstrated how a malicious website can use the timing of browser graphics operations to steal sensitive user data. Fortunately for users, timing attacks that are easily demonstrated in a controlled environment can prove tricky to implement reliably in the wild. However, this does not mean that browser vendors should not fix these holes. The basic techniques described in this paper will inevitably be improved upon to increase their speed, reliability and real-world usefulness.

Browser vendors have a difficult task in finding and preventing timing vulnerabilities. Such issues are not easy to identify from a code review perspective and may be found only through testing and experimentation. Fixing these timing holes can also have an effect on performance. Browser vendors are in a constant race to improve the speed of their browsers. The asynchronous URL lookups and filter optimisations that make these timing attacks possible were intended to increase browser performance. Fixing them could involve a trade-off between privacy and performance.

Website owners can protect themselves from the pixel reading attacks described in this paper by disallowing framing of their sites. This can be done by setting the following HTTP header:

X-Frame-Options: Deny

**Figure 33**  
Setting the X-Frame-Options header to prevent framing

This header is primarily intended to prevent clickjacking attacks, but it is effective at mitigating any attack technique that involves a malicious site loading a victim site in an iframe. Any website that allows users to log in, or handles sensitive data should have this header set.

Firefox users who wish to protect themselves against the history sniffing attack can disable visited link styling. This can be done by following these steps:

- Type 'about:config' in the location bar and press enter
- Find the 'layout.css.visited\_links\_enabled' setting using the search bar
- Double click the setting to toggle it to false

Users of all browsers can also reduce the risk of history sniffing attacks by regularly clearing their history. This can be done by using the Ctrl-Shift-Del keyboard shortcut in Firefox, Chrome and Internet Explorer.



## Security Bug Reports and Disclosure

Context has reported all of the security related findings detailed in this paper to the relevant browser vendors. Our original SVG filter timing attack research was done against Firefox and was reported to Mozilla in December 2011.

All the other findings in this paper were researched in 2013 and were reported to vendors in June 2013.

Listed below are links to the bug reports for Chrome and Firefox. At the time of writing, these links are not public, though it is expected that they will be opened shortly:

Firefox 22 updated the feMorphology filter to remove the timing differences described in this paper. However, other filters may still be vulnerable.

### SVG Filter Timing Attack:

- Firefox bug report: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=711043](https://bugzilla.mozilla.org/show_bug.cgi?id=711043)
- Mozilla advisory: <http://www.mozilla.org/security/announce/2013/mfsa2013-55.html>
- Chrome bug report: <https://code.google.com/p/chromium/issues/detail?id=251711>

### Link Repainting Attack:

- Firefox bug report: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=884270](https://bugzilla.mozilla.org/show_bug.cgi?id=884270)
- Chrome bug report: <https://code.google.com/p/chromium/issues/detail?id=252165>



## About Context

Context Information Security is an independent security consultancy specialising in both technical security and information assurance services.

The company was founded in 1998. Its client base has grown steadily over the years, thanks in large part to personal recommendations from existing clients who value us as business partners. We believe our success is based on the value our clients place on our product-agnostic, holistic approach; the way we work closely with them to develop a tailored service; and to the independence, integrity and technical skills of our consultants.

The company's client base now includes some of the most prestigious blue chip companies in the world, as well as government organisations.

The best security experts need to bring a broad portfolio of skills to the job, so Context has always sought to recruit staff with extensive business experience as well as technical expertise. Our aim is to provide effective and practical solutions, advice and support: when we report back to clients we always communicate our findings and recommendations in plain terms at a business level as well as in the form of an in-depth technical report.

---





## Works Cited

[1] W3C, "Timing control for script-based animations," 2012. [Online]. Available: <http://www.w3.org/TR/animation-timing/>.

[2] S. Emrys, "CSS Fingerprint: preliminary data," 2010. [Online]. Available: <http://saizai.livejournal.com/960791.html>.

[3] D. Jan, R. Jhala, S. Lerner and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *17th ACM Conference on Computer and Communications Security*, 2010.

[4] D. Baron, "Preventing attacks on a user's history through CSS :visited selectors," 2010. [Online]. Available: <http://dbaron.org.mozilla/visited-privacy>.

[5] Z. Weinberg, E. Y. Chen, P. R. Jayaraman and C. Jackson, "I Still Know What You Visited Last Summer," in *2011 IEEE Symposium on Security and Privacy*, 2011.

[6] M. Zalewski, "Some harmless, old-fashioned fun with CSS," 2013. [Online]. Available: <http://lcamtuf.blogspot.co.uk/2013/05/some-harmless-old-fashioned-fun-with-css.html>.

[7] Mozilla, "MozAfterPaint," 2013. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/Reference/Events/MozAfterPaint>.

[8] "Bug 600025 - CSS timing attack on global history still possible with MozAfterPaint," 2010. [Online]. Available: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=600025](https://bugzilla.mozilla.org/show_bug.cgi?id=600025).

[9] Mozilla, "Bug 608030 - Disable MozAfterPaint for content by default," 2010. [Online]. Available: [https://bugzilla.mozilla.org/show\\_bug.cgi?id=608030](https://bugzilla.mozilla.org/show_bug.cgi?id=608030).

[10] W3C, "Text Shadows," [Online]. Available: <http://www.w3.org/Style/Examples/007/text-shadow.en.html>.

[11] W3C, "CSS Transforms," 2012. [Online]. Available: <http://www.w3.org/TR/css3-transforms/>.

[12] W3C, "Filter Effects 1.0," 2013. [Online]. Available:



[http://www.w3.org/TR/2013/WD-filter-effects-20130523.](http://www.w3.org/TR/2013/WD-filter-effects-20130523)

- [13] W3C, "SVG Filter Effects Specification," 2011. [Online]. Available: <http://www.w3.org/TR/SVG/filters.html>.
- [14] "SVG Wow!," 2011. [Online]. Available: <http://svg-wow.org/>.
- [15] Microsoft, "Hands On: SVG Filter Effects," 2012. [Online]. Available: [http://ie.microsoft.com/testdrive/graphics/hands-on-css3/hands-on\\_svg-filter-effects.htm](http://ie.microsoft.com/testdrive/graphics/hands-on-css3/hands-on_svg-filter-effects.htm).
- [16] Context Information Security, "WebGL - A New Dimension for Browser Exploitation," 2011. [Online]. Available: <http://www.contextis.co.uk/research/blog/webgl-new-dimension-browser-exploitation/>.
- [17] R. Kotche, Y. Pei and P. Jumde, "Stealing cross-origin pixels: Timing attacks on CSS filters and shaders," 2013. [Online]. Available: <http://www.robertkotcher.com/pdf/TimingAttacks.pdf>.
- [18] P. Rouget, "Firefox 4: Drawing arbitrary elements as backgrounds with -moz-element," 2010. [Online]. Available: <http://hacks.mozilla.org/2010/08/mozelement/>.
- [19] Mozilla, "Mozilla Developer Network - image-rendering," [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/CSS/image-rendering>.



## Context Information Security

### London (HQ)

4th Floor  
30 Marsh Wall  
London E14 9TP  
United Kingdom

### Cheltenham

Corinth House  
117 Bath Road  
Cheltenham GL53 7LS  
United Kingdom

### Düsseldorf

1.OG  
Adersstr. 28  
40215 Düsseldorf  
Germany

### Melbourne

4th Floor  
155 Queen Street  
Melbourne VIC 3000  
Australia