BLOBFOLIO
(We're actually good at what we do.)
(And we're friendly too!)

When a Stranger Calls: Sanitizing SVGs

SVGs aren't quite images, nor are they documents (except, you know, literally). They inhabit the soft places where matter is energy, time is space, and a simple picture of a girl in a bikini might be able to hijack your computer to **make a phone call**. If the format were a television character, it would be a supermodel, motorcycle rebel, and seedy lecher all at once.

To be clear, SVGs are damn useful. Their foot print is small and compressible, they're able to scale to any dimension or resolution without loss of fidelity, and can be manipulated like other DOM elements, opening the door to all sorts of really cool interactive web experiences. To modern developers, they are the answer to all sorts of different problems. We use them on almost every project.

But that said, they are also one of the most dangerous formats ever conceived. So prickly these pears are, no developers have managed to fully neutralize them. The problem is so apparently insurmountable that major web platforms like WordPress, more than 15 years since the spec was finalized, **still have yet to greenlight their inclusion**.

So what's the deal? Why so scary?

While to the average user an SVG seems like any other image, it isn't. Beneath the surface, SVGs _are code, _and code yearns to be run.

An SVG might load external resources from malicious servers. It might include Javascript, which can leak your personal information, steal logins, trigger malware popups, etc. SVGs can even embed entirely different formats, like PDFs, which can in turn launch their own code with even greater system access. Improperly handled SVGs can also infect web servers themselves.

In short, an SVG can be everything and anything.

To be able to open up SVG technology to the average web user (i.e. people who don't know or care how shit works, so long as it works), tools need to be put in place to analyze and scrub the contents. Gotta protect the people from themselves!

Surprisingly, no such tools exist. Not a one even begins to approach the nuanced ways in which an SVG can be evil.

So, like every other damn computer problem, we set about doing it ourselves.

Make It Work

And so it was I found myself once again playing the role of investigative journalist. What makes an SVG an SVG? How do hackers and nogoodniks exploit this format? What sort of approaches would provide the most comprehensive cleanup, while also being flexible enough not to choke on mildly corrupt but innocent images?

You know, background.

Well, it turns out that SVGs have been around a long time, but like the coelacanth, haven't really changed all that much since the age of the dinosaurs. For the most part, supporting applications and web browsers work best with the **second draft spec** published by W3 back in 2001. While a few later standards have been attempted, they've never really caught on.

To the second question, the answer evidently is: *every way imaginable*. There is almost no part of an SVG that can't be tempted to the dark side. So, we'll save this for later.

In terms of approach, I was able to find an existing sanitization library written in Javascript called **DomPurify**, as well as a **PHP port** of the same written by Enshrined. Their approach is taken from the *Book of Five Rings*: when faced with a large problem, [hit it with a sword to] make it smaller.

Anatomy 101

W3 standards are sort of like DNA blueprints for various web components. The items outlined in the spec are the gooey bits you'd expect to find in an autopsy. Anything left unmentioned is probably just cancer and can be tossed in a bucket.

To that end, DomPurify compiled a whiltelist of tags and attributes. Anything not in that whitelist is excised. That is definitely on the right track, but as implemented has room for improvement. For one thing, their whitelist is at once incomplete and not entirely specific to SVGs (DomPurify also tries to fix other kinds of common markup). This means that plenty of legitimate markup will be tossed, while other illegitimate markup will sneak through.

Rather than standing on the shoulders of DomPurify, I went back to the source and built a master list of **tags** and **attributes** from the official spec, including a few draft items that will probably be adopted someday, and excluding a few legal-but-dangerous items.

This simple but comprehensive prototype matching actually gets us about halfway. So well done!

Call Me Daddy

Turns out a spleen isn't always a spleen. Sometimes it is "namespaced". Namespaces are a feature of XML, and by extension SVG, whereby tags and attributes can be grouped under an arbitrary umbrella.

```
<!-- it starts with a declaration -->
<svg xmlns="http://www.w3.org/2000/svg" xmlns:foobar="http://www.w3.org/20
    <!-- a regular tag -->
    <style>.black { fill: black; }</style>
    <!-- and a namespaced version -->
    <foobar:style>.black { fill: black; }</foobar:style>
</svg>
```

A sanitizing script looking for *exact* tag and attribute matches is going to throw out `<foobar:style>` and `xmlns:foobar` as invalid, but they aren't. The first-pass solution doesn't require too big an intellectual leap. One can simply check to see if the whole thing matches the whitelist, and if not, check to see if the the non-namespace side of the colon matches.

Done and done. Except not done. In practice, this is a little more nuanced.

The largest issue when working this out in PHP is that the main tool used for basic XML parsing, **DOMDocument**, doesn't even list namespaced attributes when listing all other attributes. This was something overlooked by the PHP port of DomPurify as well as my own first stab at the problem. While `onclick` might be forbidden and stripped, `onclick:foobar` won't even turn up in a search.

What's more, once namespaced elements are located through other channels, any attempts to manipulate or remove them can send shockwaves across the whole document, for example, extra tags might pop up all over the place to match a "new" namespace. But other times, nothing special happens. Again, *nuance*.

One last trick: a namespaced element is only a namespaced element if its namespace corresponds to a namespace definition. Without it, it is likely to evade all but the most thorough of examinations.

In the end, proper namespace sanitizing required more lines of code than every other component combined. Haha.

Hidden Dragon

There is pretty much no place within an SVG that Javascript cannot hide. The tag and attribute whitelisting (along with proper namespace handling) is sufficient to knock out the places Javascript is *meant* to be, namely `<script>` tags and `onwhatever` attributes. But Javascript can also masquerade as a URL, like `href="javascript: alert('Gotcha!');"`.

DomPurify decided on a scatter approach for scripty protocols; it simply removes any string leading up to "script:". That works (except when it doesn't), but will also break an SVG containing something innocent like `<title>My First Script: How Lassie Got Her Groove Back</title>`.

What I needed was a way to determine if a particular attribute should have a URL-ish value or not. If so, I could throw a million protocol and host and formatting tests at it. If not, a gentler touch would do.

I found what I was looking for in the W3 **IRI** spec. And so, a third master list was added.

Unfortunately, XML makes asking simple questions like, "Do you start with *javascript:*?" very difficult to ask. The simplest trick is whitespace. Browsers ignore it, programming languages don't. If a value is " javascript:...", that leading space can throw a basic check for a loop.

That problem is solved easily in PHP by using a function called `trim()`, which trims whitespace. Except when it doesn't (certain types of whitespace, like the non-breaking space, are not stripped). And so more sophisticated filters are needed.

XML attributes can also contain encoded entities which the browser will graciously decode and play with. A value of "&#106;avascript:..." will look just like that in PHP, and so again, easy to miss. But to a web browser, "&#106;" is just another way to write the letter "j".

Okay, so again in PHP, decode everything, then check. Only, what about "&amp;#106;avascript..."? Decoding that will result in "&#106;avascript...". So what, decode it again? HOW DEEP DOES THIS SHIT GO?! (As deep as the universe, dear, as deep as the universe.)

Okay, so we need to run decoding in an indefinite loop until the string it returns matches the string it returned on the previous run. Are we there yet?

No.

PHP's entity encoding and decoding functions are, for reasons nobody still living can explain, incomplete. There are more things in HTML than are dreamt of in your PHP interpreter. And so we need custom functions to standardize, fix, and decode entities PHP might miss.

And?

And Unicode. And control characters. And, while I haven't been able to confirm, probably Demotic Egyptian.

Javascript isn't even the only protocoltergeist to watch out for. There are various other scripty things like vbscript, as well as an encoding format for data, like, any data. Data URIs can literally represent any kind of file or content type, be it a script, an image, or the entire Indiana Jones trilogy (of course there's only three, don't be silly).

But enough. This section has gone on long enough. One by one, cut the pieces down, and eventually, checks can be run.

Server Dangers

We're a long way from done. Haha. One thing I found interesting about DOMDocument's XML parsing behavior is that it doesn't filter out code from dynamic scripting languages like PHP or ASP. Insofar as it is concerned, they're just text. Which, if they make it all the way to a web browser, is true.

But if a person is loading an SVG's contents in PHP while building an HTML page, and if that person just runs `include()` or `require()` like they would for any old template file, PHP will include the SVG source *as code*. If there is PHP code within that document, PHP will run that code. The same applies to ASP or any other dynamic scripting environment.

(For reference, always load the contents of an SVG to a variable using a function like `file_get_contents()`.)

And so, more things to strip out.

On the bright side, we're nearly done!

Off-Site Trickery

The last, and hardest piece to account for, are links to external resources. There are plenty of legitimate reasons to link to an externally-hosted document. For example, maybe your SVG needs to embed a web font hosted by Google Fonts. Add the CSS and you're good to go.

But from the where PHP stands, there's no telling what any of those resources actually are. They could point to malicious Javascript or even another SVG. And aside from the IRI attributes I mentioned earlier, more general links can be shoved just about anywhere, though usually in CSS via `import` or `url` functions.

In the end, I decided external links are mostly pointless. I built a small whitelist of the most common (and often necessary) third-party hosts, places like **W3.org** and **CreativeCommons.org**, and decided to leave it at that. The functions I built implementing all of these sanitizing passes are extensible, so individual projects can tweak as needed.

The Fruit

The results of all of these investigations and labors have been incorporated into the general/file security WordPress plugin **Lord of the Files**. A native PHP (5.6+) port has also been worked into our general-purpose collection of code helpers **blob-common**, which can be used on any PHP project.

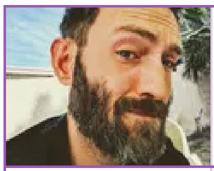
At present, this work has taken SVG sanitizing farther than it's ever been, but I'm not convinced it is completely sewn up. If I had to guess, I'd say we're at about the 95% mark. The main features include:

- Removal of comments (and / / varieties);
- Removal of XML, PHP, ASP;
- Removal of Javascript tags, attributes, and values;
- Sanitizing of CSS url(...) rules;
- Namespace and IRI sanitizing via protocol and host (both extensible);
- Extensible tag whitelist;
- Extensible attribute whitelist;
- Miscellaneous formatting repairs;

- Whitespace collapsing;

Enough progress has been made to put official SVG support on WordPress' media priorities list. For those interested, keep an eye on ticket #**24251** for news and updates on that front. I'm hoping we'll get there sometime this year!

And of course, if you happen to have any evil SVGs and would like to advance the cause, please zip them up and send them my way. :)



Josh Stoik
9 March 2017

Previous	Climbing MIME Improbable
Next	Fix Linux DNS Issues Caused by Systemd-Resolved