Crouching Tiger – Hidden Payload: Security Risks of Scalable Vectors Graphics

Mario Heiderich
Chair for Network and Data Security
Ruhr-University Bochum, Germany
mario.heiderich@rub.de

Tilman Frosch
Chair for Network and Data Security
Ruhr-University Bochum, Germany
tilman.frosch@rub.de

Meiko Jensen
Chair for Network and Data Security
Ruhr-University Bochum, Germany
meiko.jensen@rub.de

Thorsten Holz
Chair for System Security
Ruhr-University Bochum, Germany
thorsten.holz@rub.de

ABSTRACT

Scalable Vector Graphics (SVG) images so far played a rather small role on the Internet, mainly due to the lack of proper browser support. Recently, things have changed: the W3C and WHATWG draft specifications for HTML5 require modern web browsers to support SVG images to be embedded in a multitude of ways. Now SVG images can be embedded through the classical method via specific tags such as `<embed>` or `<object>`, or in novel ways, such as with `<img>` tags, CSS or inline in *any* HTML5 document.

SVG files are generally considered to be plain images or animations, and security-wise, they are being treated as such (e.g., when an embedment of local or remote SVG images into websites or uploading these files into rich web applications takes place). Unfortunately, this procedure poses great risks for the web applications and the users utilizing them, as it has been proven that SVG files must be considered fully functional, one-file web applications potentially containing HTML, JavaScript, Flash, and other interactive code structures. We found that even more severe problems have resulted from the often improper handling of complex and maliciously prepared SVG files by the browsers.

In this paper, we introduce several novel attack techniques targeted at major websites, as well as modern browsers, email clients and other comparable tools. In particular, we illustrate that SVG images embedded via `<img>` tag and CSS can execute arbitrary JavaScript code. We examine and present how current filtering techniques are circumventable by using SVG files and subsequently propose an approach to mitigate these risks. The paper showcases our research into the usage of SVG images as attack tools, and determines its impact on state-of-the-art web browsers such as Firefox 4, Internet Explorer 9, and Opera 11.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Scalable Vector Graphics; Web Security; Browser Security; Cross Site Scripting; Active Image Injections

1. INTRODUCTION

One of the factors behind the huge success of the World Wide Web is its ability and capacity for viewing image files within a web browser. Compared to the text-only formats, an image can convey considerably more information. A typical browser supports many different image file formats, such as JPEG, PNG and GIF files, whilst the vast majority of websites on the Web contain at least one graphic in either one form or another. Since image files are complex and need to be parsed and rendered before they can be displayed by a browser, it comes as no surprise that the images have security implications. To give an example, there were several cases in the past where the validation routine of image libraries contained security flaws leading to vulnerabilities [1, 2, 4]. For this reason, we need to consider the risk of images as the attack vectors.

One image format that has up till now received very limited scrutiny and little attention from the web development community is *Scalable Vector Graphics* (SVG [5]). This family of file formats comprises several specifications and specification drafts for composition and rendering of the vector based images and graphics. SVG is based on XML and was first published by the W3C in 1999. SVG images have not gained much traction from the web developers, as the support provided by major browsers was not consistent and only

Figure 1: A classic GhostScript/SVG example

a small subset of SVG features had been known to work reliably on a sufficiently large base of the web browsers. Browsers, like Firefox 1.5, already supported a decent subset of SVG features in November 2005, showcasing SVGs such as the famous *GhostScript Tiger* shown in Figure 1. This particular image is often used to illustrate the abilities of the vector based graphics to display complex structures. Other browsers, namely Internet Explorer, did not support SVG, unless a user installed an external plug-in.

All this has significantly changed with the recent appearance of HTML5: the W3C and WHATWG draft specifications for HTML5 *require* modern web browsers to support SVG images' embedment in a multitude of ways [25]. SVG images can now for example be engrafted in a given document either in the classical way via specific tags such as `<embed>` or `<object>` tags, or in the novel ways such as with `<img>` tags or inline in any HTML5 document. Internet Explorer 9 currently supports a large subset of SVG features as tests with the tech previews and available beta versions show. Furthermore, both Firefox and Webkit-based browsers, such as Chrome and Safari, as well as Opera, provide thorough SVG support.

Hitherto, SVG is mainly used in the context of screen and print design, as well as in the cartography and medical imagery, which all can be attributed to the lack of proper browser support [35]. This is however expected to change, owing to the SVG support being implemented in all modern web browsers, consequently lifting SVG files from being a niche format for W3C compliant and plug-in-equipped browsers only, to a widely used toolkit for enhancing images, diagrams and rich-text documents across the board. Depending on the rendering client's capabilities, an SVG file can contain interactive and animated elements. Processing events and raster images, embedding videos, and rich-text are also feasible. Contrary to popular belief, SVG files should thus not be considered to be plain images or animations, and it is necessary to treat them as fully functional, one-file web applications capable of potentially containing HTML, JavaScript, Flash and other interactive code structures.

In this paper, we elaborate on the security risks of improper SVG handling. We introduce several novel attack techniques of using SVG images to target modern, real life web applications (such as MediaWiki installations like Wikipedia, DeviantArt, and other high profile websites), as well as their unsuspecting users. Specifically, in Section 3 we present an *Active Image Injection* (AII) attack, in which arbitrary JavaScript code can be delivered via SVG files. Several other attacks such as SVG-based cross site scripting attacks or SVG chameleons (i.e., files that are interpreted differently depending on how they are opened) are also delivered. AII attacks are particularly important, since they are caused by faulty SVG implementations in modern browsers, thus affect all websites allowing users to embed external images – thereby our research significantly extends and partly falsifies the information available in the Browser Security Handbook, so far only covering risks connected to browser-deployed SVG in plugin containers. Furthermore, we discuss how current state-of-the-art filtering techniques are deceivable via using SVG files.

The basic idea behind all of our attacks is the fact that SVG files can accommodate active content, whereby browsers actually interpret this content due to its being standard-compliant. This idea is related to similar attacks that take advantage of code embedded in document formats [8,13,32] and we show that SVG images can be turned into an attack vector. In addition, we demonstrate the damaging potential of SVG files embedding arbitrary data formats and show how this property can be used to carry out attacks using Adobe Reader, Java Runtime Engine and Flash Player vulnerabilities. We discuss the impact of our attack on modern browsers such as Firefox 4, Internet Explorer 9, and Opera 11, showing that especially inline SVG grants new possibilities for bypassing website- and browser based XSS filters.

To mitigate the risks introduced by SVG-based attacks, we debate and evaluate several defense strategies. We showcase a filtering solution that is capable of removing potential malicious content from a given SVG file. Our approach does not break the functionality of the core features of fully interactive and descriptive SVG images. Instead, we extend an existing and widespread filtering software to support filtering of illegitimate and malicious content from SVG files, without damaging the benign file structure and contents. We have formerly implemented a prototype of the system and tested it with 105,509 SVG images obtained from Wikipedia. We found that we can filter 98.5% of the files without causing any difference in the visual appearance of the image, and for the remaining 1.5% we determined the visual deviation to be negligible in more than half of the cases.

In summary, we make the following contributions:

- We are the first ones to demonstrate the security risks tied to SVG files in the context of the World Wide Web and comparable client-server environments. Furthermore, we argue that SVG files must not be perceived as images, but rather full stack applications, provided the cases of them being rendered by a web browser or similar client software. This holds for SVG images rendered via `<embed>` and similar tags, as well as displayed as standalone file in modern web browsers.

- We show several innovative attack techniques illustrating the potential of maliciously crafted SVG files, which we call *Active Image Injection* (AII). We exhibit how SVG files can cause damage to major websites, and discuss the damage potential of SVG files embedding arbitrary data formats. The hereby discussed AII attacks affect a whole browser family, rendering *any* website leaving user submitted images vulnerable if the users visit it with the affected web browsers. Moreover, we show how inline SVG can be used to facilitate XSS filter bypasses on current web browsers.

- We introduce a defense solution to prevent SVG-based attacks by filtering potentially malicious content from a given SVG file. The large-scale evaluation results suggest that this approach can successfully defeat AII attacks on a practical level.

2. SVG BASICS AND SPECIFICS

This section provides a brief overview of the SVG file format and discusses the attack surface enabled by this image format, i.e., we illustrate the different ways available for an attacker to send arbitrary SVG files to a victim.

2.1 Overview and Benefits

The *Scalable Vector Graphics* (SVG) file format was introduced in 1999 when it was published by the W3C in an attempt to combine the best of both the specification drafts for *Precision Graphics Markup Language* (PGML) developed and published by Adobe, and the *Vector Markup Language* (VML) developed and published by Microsoft, Autodesk, Hewlett Packard, and others, all in 1998. SVG is a vector graphics format, i.e., it uses geometrical primitives such as points, lines and curves to describe an image, while it supports both static and dynamic content. The above mentioned static content and dynamic behavior are described in an XML-based format, which implies that SVG files are in fact text files.

The impact of SVG on the Internet can be described with the following characteristics:

- **Scalability**: SVG files are, as the name indicates, *scalable* based on their nature as vector graphics. This means that graphical output devices of any size can render SVG images without significant information loss or facing deficiencies in display quality. In times of websites' inhomogeneous output devices such as browsers, feed- and screen readers, smartphones and *Wireless Markup Language* (WML) compatible cellphones, this enables web developers to publish rich online documents without having to worry about the screen dimension of the device requesting the document.

- **Openness**: Unlike classical, raster-based image formats, SVG files are neither stored in a binary format nor is there a compression scheme rendering the actual content of the file unreadable for the human eye. SVG images can be enriched with meta-data and comments, so that a (handicapped) human as well as a program (e.g., a search engine or comparable parser) can effecively extract relevant information from the file in question. In this case accessibility is ensured by storing more descriptive information in a given file, compared to the rather limited possibilities of image comments provided by GIF and PNG files, or the embedded *Exchangeable Image File Format* (EXIF) data in RAW and JPEG images. Note that gzipped SVG files (also know as SVGZ) create an exception, for they do not use compression as means to reduce the file size.

- **Accessibility**: Related to the aforementioned openness, an SVG image can be enriched with sufficient meta-data and information to be *Web Accessibility Initiative* (WAI) compliant, i.e., visually impaired users can extract relevant information by having their tools parse and read the meta-data embedded in the SVG. Furthermore, screen-readers can (theoretically) parse SVG data and describe the shapes and visuals used by the image, allowing a broader range of users to benefit from its contents. In contrast, raster-based images are void of this kind of support.

The SVG family consists of several members and we use the following three file types as examples in later sections:

- **SVG Full 1.1**: SVG Full describes the full SVG feature set including 81 different SVG elements and tags. The specification is designed without a special focus on the devices parsing the SVG data.

- **SVG Basic 1.1**: SVG Basic is supposed to deliver a subset of the SVG Full specification to ease the implementation for developers of browsers for PDAs and handheld devices. SVG Basic only provides 70 of the 81 SVG elements specified in SVG Full 1.1. Contrary to SVG Tiny 1.2, the SVG Basic 1.1 features also include support for SVG fonts.

- **SVG Tiny 1.2**: SVG Tiny is specifically designed for smartphones and similar mobile devices with limited computing, rendering, and display capabilities. The subset of allowed SVG elements and tags has been reduced to 47 elements. SVG Tiny also ships several exclusive possibilities for event binding and external resource loading which we discuss in Section 3.

Additionally, the SVG specification provides interface descriptions for an *SVG Document Object Model* (DOM), which implies that SVG files also offer some dynamic capabilities. Users can create SVG files capable of providing event handling, effects, time-based changes and animations, as well as zoom effects and other helpful display enhancements. A large set of filters can be applied to the elements of SVG files to even more greatly increase the possibilities for image transformation and animation.

The ability to combine SVG with the *XML Linking Language* (XLink) features allows SVG files to link elements to other elements in the same image file, other image files or arbitrary objects referenced via *Uniform Request Identifiers* (URIs). Furthermore, these image files support the implementation of *International Color Consortium* (ICC) and *Standard Red-Green-Blue* (sRGB) color profiles, allowing the embedment of arbitrary content such as Flash, PDFs, Java and HTML via the `<foreignObject>` element.

2.2 HTML, SVG, and XML

Being historically an XML-based language, processing of SVG documents has been quite different from the way browsers process classic HTML websites. For instance, a slight violation of the XML syntax, such as missing closing tags or attribute value quotations, typically cause SVG processors to exit with an error. However, with the integration of SVG capabilities into modern browsers, this strict parsing approach got amalgamated with their more tolerant way of processing HTML, CSS, JavaScript and the like.

This mixture is causing browsers to process SVG through using two different processing modes: an HTML processor engine for CSS and JavaScript contents, and an additional SVG parser supporting XML-specific features like XML transformations (e.g. XSLT), XML Entity resolution, and tracking of XML Namespace bindings. Depending on the particular website's style of using SVG, the browser switches between the two processing modes on the go. For instance, encountering an inline `<svg>` tag within an HTML5 document causes the browser to switch from HTML mode to XML/SVG mode. Vice versa, if the browser encounters an HTML-specific tag (e.g. `<p>`) within an SVG mode context, it automatically closes all open SVG elements, switches to HTML mode, and renders the given tag.

As we show in the following sections, this approach is error-prone and may cause a lot of SVG-related vulnerabilities in most state-of-the-art browser engines.

2.3 Deployment Techniques

The capabilities, in terms of scripting and content inclusion of SVG files, strongly depend on how they are embedded in a website or loaded by the browser attempting to display them. In this section, we focus on five diverse manners of SVG files being deployed by a webserver or web application. In addition, we outline the attack surface we have discovered in connection to the five deployment techniques. The specific attack vectors we use are discussed in detail in Section 3, followed by Section 4 focusing on how to mitigate and defend against those attacks.

1. **SVG deployed via uploaded files**: A large number of tested web applications (e.g., MediaWiki and Wikipedia, OpenStreetMaps, DeviantArt, OpenClipArt, and several other free image hosting services) consider SVG files to be equivalent to raster images such as PNG, JPEG, and GIF files in terms of security implications. MediaWiki and Wikipedia claim to block the upload of SVG files containing script code, but we did manage to easily bypass this restriction. As we show in the next section, SVG files should be displayed and executed with a heavily limited set of features to prevent *universal XSS attacks*, since these files might contain scripts, embed arbitrary content and process events. In addition, we discovered alternative course of action for outmaneuvering the capability limitations of current browsers, which are used to protect sensitive DOM properties such as a website's cookies. Those are discussed in detail in Section 3.2

2. **SVG deployed via CSS backgrounds and *img* tags**: This way of deploying malicious SVG files can be considered as the most dangerous and effective attack, granted that the majority of the web applications judge `<img>` tags as part of user generated HTML to be harmless: `<img src="evil.svg">`
   Filter software, such as the HTMLPurifier [43], OWASP AntiSamy [16], and similar tools whitelist image tags and a large number of web applications allowing user generated HTML are prone to be sensitive to a novel class of attacks we term *active image injections*. Again, we underline that SVG files should be displayed and executed with a heavily limited set of features to prevent universal XSS attacks. In Section 3.2 and 3.5, we particularize on the attack vectors we discovered through using this presumably harmless way of deployment, and outline an innovative method of attacking browsers and high traffic web applications.

3. **SVG deployed via inline SVG**: The HTML5 specification draft suggests the web browsers to support websites providing *inline SVG*. This means a developer and an attacker are equally able to inject arbitrary SVG content right into the markup tree of a HTML document. The browser will then switch its parsing mode, use an intermediary layer to parse the (possibly non-well-formed) SVG content, clean it up, pass it on to the internal XML parser and layout engine, and then commence parsing and rendering the remaining optional HTML content. The last step likely includes even more inline SVG elements [17] that are capable of interacting with the already parsed content. In Section 3.4, we illustrate how this facilitates XSS filter bypasses of existing websites, filter libraries, and most importantly browsers and comparable user agents.

4. **SVG deployed as font file (*SVG Fonts*)**: The SVG standard specifies several possibilities to create font files completely consisting of SVG data [3]. Modern browsers allow their inclusion via CSS and the `@font` directive. In case when the browser supports SVG fonts, for every character with an SVG font assigned, the parser checks whether the character has a representation as an SVG path/glyph data and applies this to the view port if possible. SVG fonts provide a prominent range of features for detailed and complex font formatting, Unicode support, alternative glyphs, default behavior for missing glyphs, and more. We detected attack vectors allowing the deployment of arbitrary plug-in content via SVG fonts working on a variety of desktop and mobile user agents.

5. **SVG deployed via *iframe*, *embed*, or *object* tags**: The attack surface is comparably large to the one for the classic XSS and does not differ much from the regular `<iframe>` and `<script>` injections. It will thus not be discussed in more depth in this paper.

3. ATTACK VECTORS USING SVG FILES

Based on the prerequisites discussed in the previous section, we now introduce several different attacks based on SVG files and discuss their security impact.

3.1 Responsible Disclosure and Ethical Aspects

We describe several novel attacks related to SVG files and their security impact, ranging from universal XSS attacks to triggering vulnerabilities based on SVG images. Presenting such attacks is obviously an ethically sensitive area and one question that arises is if it is acceptable and justifiable to publish the attack details. In the following, we describe most attack vectors from a high-level point of view and do not present all implementation details. If a (legitimate) researcher is interested in test cases, we are happy to share them. Furthermore, we have contacted all major browser vendors and informed them about these problems. We are in contact with them and several reported problems have already been fixed. As a result, an attacker cannot easily take advantage of these identified attack vectors.

In addition, we introduce in Section 4 an approach to mitigate the attacks presented in this paper based on removing suspicious content from SVG files at the server side. We are in contact with the affected website's security teams, and discuss with them the possibility to deploy our countermeasure for these attacks. The WikiMedia team is interested in testing and deploying the tool we created and discuss in Section 4. Our mitigation approach can also be implemented as a local filter proxy to protect a web browser against malicious SVG images and we evaluate at the moment the effort required to implement this local mitigation approach.

3.2 Local JavaScript Execution and SVG Chameleons

One of the least sophisticated attack techniques (that is still rather likely to work in real-world scenarios) is tricking the victim into saving an SVG image from a website and opening it later on for repeated viewing pleasure. There are only a few ways for technically less affine users to tell a classic raster-based image (PNG/JPEG/GIF) apart from an SVG image. Once saved locally and double-clicked, the browser will open the file – since most users do not have a dedicated software installed that changes the application to handle the SVG MIME type. The SVG file is consequently opened from a *file* URI and in case it contains JavaScript, this code will be executed in the same context. Depending on the web browser the victim is using, the JavaScript can then attempt to read other files from the hard-disk or neighboring directories, and cause a data leakage incident. A thrifty adversary can cause the locally running JavaScript to load an applet from an arbitrary domain, thus even bypassing many of the security restrictions modern browsers apply for local script execution.

Similar attacks could be performed with *SVG Chameleons*, i.e., files containing both SVG and HTML content. Using in-line XML transformation (XSLT), we managed to craft an SVG file that acts like an image if embedded via `<img>`, CSS or similar ways, but unfolds to a full stack HTML file containing no SVG elements anymore as soon as opened directly [14]. This attack works with Gecko-based browsers, since it appears to be the only layout engine supporting in-line XSLT in SVG files. The attack would involve uploading an SVG Chameleon to a website such as Wikipedia, and trick the victim into right-clicking the image shown embedded and choosing to view the original. As soon as that happens, the XSLT will transform the SVG into an HTML file and execute embedded script code or worse [23].

Interestingly, some browsers such as Firefox do not allow cookie access in case the SVG file is being opened directly. This is especially important in cases where an attacker can upload SVG files to the same domain a targeted user is being logged into. The reason for that limitation is a different handling of the *SVGDOM* compared to the regular website's DOM. The SVGDOM does not know properties such as `document.cookie` or even `document.body`. We discovered ways to get around this limitation, though, by having the SVG create a `<foreignObject>` tag containing an Iframe loading the affected website. After the `onload` event of the Iframe, we injected JavaScript into its scope capable of extracting sensitive data such as the cookies and bypassing the more or less unconscious security restrictions.

3.3 Facilitating Cross Site Scripting Exploits

SVG images provide many possibilities for executing JavaScript in uncommon ways. Many of these are not known to typical web developers and thus are not covered by filter software protecting websites against XSS attacks. SVG Tiny, for example, allows to execute JavaScript by using a `handler` element with an `event` attribute, as shown in Listing 1. In case the event assigned to the handler element is specified as `load`, the text content of the handler element will be executed as JavaScript without any user interaction. Blacklist-based XSS filter systems are usually not aware of such ways of executing code, thus they are not capable of detecting this kind of attacks.

```
<svg xmlns="http://www.w3.org/2000/svg">
 <handler
  xmlns:ev="http://www.w3.org/2001/xml-
      events"
  ev:event="load">
    alert(1)
 </handler>
</svg>
```

Listing 1: Example for uncommon SVG-based JavaScript execution via <handler> tag

Another uncommon way of embedding malicious JavaScript in SVG files is shown in Listing 2. Using SVG's `<set>` tag, we dynamically equip an `<feImage>` tag with an `xlink:href` pointing to a `data:` URI. This kind of image element is meant to be used to apply overlay effects for SVG elements utilizing external resources. Shielded by the Base64 encoding, this URI contains another SVG image that itself contains malicious JavaScript – which is run immediately on loading the `<feImage>` tag.

```
<svg
 xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink
     ">
 <feImage>
  <set
   attributeName="xlink:href"
   to="data:image/svg+xml;charset=utf-8;
   base64,PHN2ZyB4bWxucz0iaHR0cDovL3d3dy53
   My5vcmcvMjAwMC9zdmciPjxzY3JpcHQ%2BYWxl
   cnQoMSk8L3NjcmlwdD48L3N2Zz4NCg%3D%3D"/>
 </feImage>
</svg>
```

Listing 2: Example for uncommon SVG-based JavaScript execution via <set> tag

These and other ways of executing JavaScript from within an SVG file were used to bypass the filter used by the MediaWiki software, which is the most commonly used open source wiki software and, among many others, the platform used by Wikipedia. We established contact with the MediaWiki team and work together with them on mitigation and defense strategies against such attacks.

We also tried to load SVG images via a `Canvas` element in an HTML website and steal information by using the `canvas.toDataURL()` feature. This method is capable of freezing the optical state of a canvas element and transforming it into a dataURI for easy saving and later usage. This attack technique to steal data cross domains has been published by Lawrence in 2009 [30], but specifically aimed to steal pixel data cross domain for attacking CAPTCHA mechanisms and similar security instrumentations involving images. We attempted to use this attack technique in a novel context and steal whole website screenshots from SVG images being applied with a `<foreignObject>` tag and cross domain Iframes. Surprisingly, this did not work at all. All tested web browsers reacted with the expected behavior and threw security errors on attempting to execute the `canvas.toDataURL()` method when accessing the SVG with cross domain content.

3.4 Facilitating Filter Bypasses

One feature distinguishing the rendering behavior of HTML from XHTML and XML based websites and documents in browsers is the handling of entities in plain text tags. Plain text tags are HTML elements considered to contain plaintext information (such as `<script>` and `<style>` tags, as well as `<noscript>`, `<noframes>` and `<nostyle>` tags). While in HTML documents entities such as `&#x61;` will be treated as such, XHTML and XML documents will have the entity be treated like its canonical representation (e.g., the character `a`). In practice, this implies that within a XHTML/XML document the code `<script>&#x61;lert(1)</script>` will execute the `alert` method, while an HTML document with the same content causes the script engine to throw an error.

Of course, this behavior applies for SVG files as well, since they are regular XML documents. Interesting in terms of web security, though, is the fact that the same applies for most web browsers to inline SVG. This implies that this behavior can be transported to regular HTML documents as soon as they contain an opening `<svg>` tag somewhere in the markup tree. While the aforementioned `<script>` tag example will not execute in an HTML document, the variation of `<svg><script>&#61;lert(1)<p>` will do.

Note that the browser's parsers are also very tolerant about well-formedness of inline SVG, and neither require attribute delimiters nor balanced tags, nor even closing tags. The `<p>` element at the end of the example shown above indicates to the parser that the inline SVG just ended and an HTML section has started. Thus, the browser automatically closes both the `<svg>` and `<script>` tags, and thereby triggers the `alert` method to execute. This technique, combined with an injection, has been tested against the most common XSS filters and we managed to bypass most of them.

3.5 Active Image Injection

Several ways of abusing SVG files to execute JavaScript in situations where no script execution should happen at all have been fixed after we reported them back to browser vendors. Some vendors even completely restricted access to the DOM from an SVG context. This makes it very hard to execute same-domain JavaScript from within SVG files delivered via CSS, image tags, CSS fonts, or other ways in which browsers deliver images. Even if a web browser can be tricked into executing JavaScript via SVG deployment methods unintended for this, the script will run in the context of `about:blank` and cannot get access to the deploying website's DOM. This effectively disables XSS attacks, since they require their payload to execute on the targeted domain and not on a bogus fully qualified domain name (FQDN) such as `about:blank`.

A yet unsolved problem for several state-of-the-art web browsers is plugin content. The Opera browser, for instance, allows to use SVG files in order to deploy plugin content such as Flash, Java, and PDF files, depending on which plugins have been installed and registered on the user's system. This does not only hold for SVG files embedded via tags such as `<embed>`, `<object>`, or `<iframe>`, but also for `<img>` tags and CSS. This means that an attacker can execute arbitrary plugin code on a victim's machine *without* any user interaction by just having the victim browse a website containing an image tag. The image can be delivered from any arbitrary domain, thus most high-traffic websites and web applications allowing user generated image content are affected by this problem.

The Opera security team has been informed about this issue in summer 2010, but so far this vendor did not address the issue with a sufficient fix. This leaves applications such as Facebook, Google Mail, Yahoo! Mail, and many other websites prone to this kind of *Active Image Injection* attack – in case their users visit the page with Opera version 9 to 11. An example showcase has been set up to demonstrate the severity of the vulnerability, also trying to enforce an urgent fix from the Opera team [22].

Early versions of the Firefox 4 beta browser were prone to AII attacks as well, but these bugs have been spotted and fixed with Firefox 4 Beta 9, and did not surface in the final release version. Nevertheless, all browser vendors should monitor the security boundaries of SVG deployment closely, since small changes can cause vulnerabilities affecting a majority of web applications at once.

3.6 Browser Vulnerabilities

So far we covered attacks utilizing malicious SVG files to be instrumented in attacks against websites. We demonstrated how SVG files can be used to facilitate XSS attacks that bypass existing and well-configured filter mechanisms and security best practices, such as encoding user-generated data into HTML entities. SVG files can have other purposes for attackers, though, and be used to leverage attacks against the browser itself, or even against the underlying operating system. During our tests, it became evident that especially complex SVGs or SVG chameleons containing executable plugin code have the potential of easily crashing web browsers. We observed and reported several cases of memory corruption occurring in state-of-the-art browsers, which were caused by faulty or incomplete implementation of SVG features, or interference effects between the browser components delivering the SVG data and other components delivering embedded plugin code and Iframes.

One significant example showing the dangers of SVG files when used as attack tools against browsers is a specific bug in Opera version 11.50 24581. This version marked a turning point in the Opera browser history, since it was the first officially released version supporting inline SVG so far. We investigated an attack scenario where an attacker creates a website providing an SVG image as *favicon*. This SVG image deployed malicious content in form of a Flash file and a Java applet, as well as, an embedded PDF file. When opening the malicious website, the Opera browser attempted to load the *favicon* to decorate the loaded page's tab and address bar, as shown in Figure 2. Although this context should never execute plugin content or JavaScript at all, the browser started to play the Flash video we used for testing, and delivered the applet and PDF file – within the address bar as can be seen in Figure 2. The code was executed in the browser context, thus an exploit like this could easily haven been escalated from a proof of concept to a full attack, demonstrating how arbitrary vulnerabilities in browser plugins can be triggered via image files.

Furthermore, SVG-based attack vectors should be considered relevant for another category of software as well: mail clients such as Thunderbird and Opera Mail make use of the same (or only slightly modified) rendering and layout

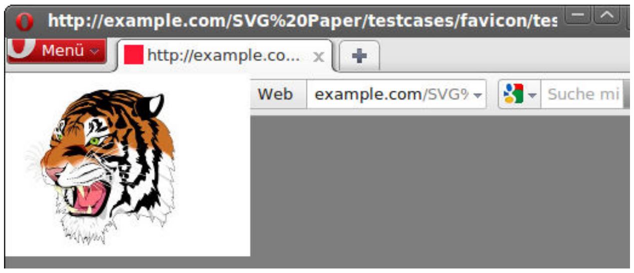

Figure 2: An SVG containing plugin content delivered via favicon

engines as their respective browser counterparts. We tested the latest versions of Opera Mail and Thunderbird 3.3, and discovered that both products allow usage of inline SVG inside HTML mails. One attack vector we crafted caused Thunderbird 3.3 alpha 3 to automatically store an SVG file in the temporary folder, open it, and execute JavaScript in the *file://* context. The only required user interaction is a click on an arbitrary part of the displayed mail body. Attacks like this can be used to place malicious software or to steal sensitive information from the */tmp* folder or other directories, depending on the location of the SVG file and the post-exploitation techniques used by the attacker.

4. MITIGATION TECHNIQUES

In the following section, we cover mitigation techniques to fend the attacks presented in the previous part. We start with a discussion of the common XML filtering and sanitization techniques, point out which problems occur when SVG is being used in modern web browsers, and list arguments as to why the classic and formerly approved approaches cannot be applied to SVGs used on the Internet. Based on these insights, we introduce and discuss *SVGPurifier*, a PHP-based, server-side SVG filter software we have developed to mitigate the identified attacks. What is more, we outline a set of recommendations for browser vendors addressing non-standard behavior that causes security problems with malicious SVG images and inline SVG parsing. Some of the listed issues have already been adopted by browser vendors during the preparatory phase of this paper completion.

4.1 XML Sanitization

The most common approach for verifying an XML document's validity is the use of *Document Type Definitions* (DTDs), *XML Schema*, or *RelaxNG* descriptions. All of these languages authorize a precise specification of which XML elements, attributes, and other tokens may be used within a specific XML document. For instance, the SVG Tiny specification provides a RelaxNG description of all XML elements and attributes that may be used in "Tiny"-compliant SVG documents. We have investigated these descriptions for practical usage in terms of removing malicious contents from SVG files. Unfortunately, we determined that their capabilities of restricting SVG contents are more focused on the XML documents' *structure* rather than restricting the content values of the SVG elements and attributes. Though both XML Schema and RelaxNG provision the means to restrict an SVG attribute value's data type to integer, string, URI, or other data types, we concluded this is insufficient for effectively filtering malicious SVG contents, such as those exemplified above.

For once, there is no feasible way to restrict the `xlink:href` attribute of `URI` data type to point to same-domain locations only, which would have been essential for resisting certain XSS attacks. The only viable way to perform such a verification while using XML Schema or RelaxNG capabilities comes down to setting the attribute's data type to a restricted string value that must conform to a given regular expression pattern. Expressing a value restriction as stated above would thus require a regular expression to be crafted for each type of restriction necessary for fending off the attacks. Although this might be (somehow and somewhat) feasible for same-domain URIs, it is easy to imagine this approach to fail for complete CSS declarations, Base64-encoded contents, and the like.

```
<!doctype html><svg><style>
&lt;img src=null onerror=alert(1)&gt;<p>
```

Listing 3: XML Entity Resolution leading to element injection

Another example that demonstrates the impossibility of XML Schema to remove suspicious content from a given file is shown in Listing 3. When processing this code fragment, Firefox 4 resolves the `&lt;` entity automatically, resulting in the alert being triggered. The crux is that the XML entity introduces an additional HTML element "on the fly" during parsing. An XML Schema validator would seen a `<style>` element with odd contents, but nothing to be alerted about (note that the missing closing tags and attribute value quotations are added automatically by the parser engine.). However, the HTML renderer resolves the entities, hence introducing the additional `<img>` tag, and triggering the `onerror` event due to the missing `null` file.

To summarize, we established that common XML validator techniques, like XML Schema or RelaxNG validation are not capable of fending the specific SVG attack vectors described above. Especially in the case of inline SVG, where HTML, CSS, JavaScript, and SVG elements are mixed arbitrarily, the approach of XML Schema validation must be revoked as completely ineffective in practice.

4.2 SVG Purification

Due to the limitations discussed above, we require another way to prevent the attacks introduced in Section 3. The basic idea of our methodology is to *purify* SVG images, i.e., remove all suspicious content from a given file and preserve as much content as possible. As a result of this transformative process, the visual impact is minimized and the suspicious content is removed. Next, we discuss the overall design and present some implementation details.

The open source market provides a lot of tools claiming to possess the skills to filter user generated input for web applications in order to eliminate active markup and script code. Their main purpose is usually XSS mitigation and markup sanitation, as well as restructuring for validity and well-formedness' sake. For PHP-based web applications, several filtering solutions are available and those most commonly used include:

- *kses* [21], that has been incorporated into a highly customized version by the popular WordPress software.
- *htmlLawed* [37], which claims to be the fastest and most compact, yet complete solution.
- *HTMLPurifier* [43], which not only sanitizes data from possibly malicious code fragments, but also generates valid and well-formed XHTML output.

We analyzed all three XSS filters and have managed to bypass each of them, demonstrating that even the most sophisticated filtering software can never be able to fully protect against malicious markup. Some of the bypasses worked only for injections into SVG files, some even in a HTML/X-HTML context.

Despite these drawbacks of the server-side filtering approaches, we have decided to choose *HTMLPurifier* as the foundation for our SVG attack mitigation tool. One major reason for this determination was the fact that *HTMLPurifier* is very well maintained, receives frequent updates and security fixes. Another reason is the quality of filtering: we have only identified a few bypasses for this tool and every single one of them was fixed very quickly upon having contacted the developers. Still, most importantly, *HTMLPurifier's* internal API allows to filter arbitrary XML data and is not limited to HTML by design, unlike the other tested tools. This knowledge allowed us to create an *SVGPurifier* branch, a software that is using the *HTMLPurifier* API, but is not touching the core components. Our *SVGPurifier* has been supplied with a large array of data based on the SVG specifications defining which tags and attributes should be allowed in the user-generated SVG files. We explicitly *whitelist* tags and attributes, as well as the tag-attribute combinations and specific value ranges for attributes. Uncommon sources for cross-site scripting attacks, such as the `<set>` tag on older Webkit-based browsers, are limited by *SVGPurifier.*

The `<set>` tag can be used in SVG files, similarly to the timing driven equivalent `<animate>`, although the `attributeName` value can only consist of a limited amount of values. Specifically, we only allow those values which cannot be used to initiate or overwrite event handlers, change `xlink:href` values on the fly, or reposition elements and element groups on a website. Our tests showed that the `<set>` and `<animate>` tags can be used to assign `javascript:` and `data:` URIs to existing elements. This can either enable attacks by initiating malicious remote inclusions, or apply malicious URL schemes to the existing elements.

Attacks like the one just portrayed, as well as those shown in Listing 4, can be no longer carried out. Take account of the fact that the `<set>`/`<animate>` functionality is not removed completely, only the remote includes and the assignment of malicious URL handlers have been blocked. Furthermore, the resulys of our evaluation show that only two files out of more than 100,000 tested instances from the Wikipedia servers made use of the `<set>`/`<animate>` feature at all (see Section 5 for details).

```
<svg xmlns="http://www.w3.org/2000/svg">
  <set
    attributeName="onmouseover"
    to="alert(1)"/>
  <animate
    attributeName="onunload"
    to="alert(1)"/>
</svg>
```

Listing 4: Initiating JavaScript execution via set/animate elements

SVGPurifier completely forbids and removes `<script>` as well as `<foreignObject>` tags and event handler usage. Later versions of our purification approach might prove to add a supplementary scripting layer to allow basic JavaScript execution, but hinder scripts from reading, and overwriting sensitive data, or conduct other activities capable of leaking sensitive data or deploying malicious code [26]. As further elaborated on in Section 5, our tests showed that none of the analyzed and purified SVG used actual `<script>` tags. Surprisingly, a large percentage of the test files were making use of `<foreignObject>` tags. The reason behind it is that the software *Adobe Illustrator* uses this tag to hide proprietary meta-info in the SVG images generated by this tool [7]. Removing these tags does not affect the visual information provided by the SVG file.

Similar problems can be caused by maliciously crafted *SVG Cascading Stylesheets* (SVGCSS). SVG styles support more properties than classic CSS for (X)HTML documents and specifically extend the feature set with font formatting, typographic features, extended pointer event behavior, and the possibility to reference to other SVG elements containing definitions and visual effects. Arbitrary SVG elements can constitute reference to other SVG elements or even let external SVG files to borrow visual information or functionality including event handling. Those references can be defined via *FunctionIRI* or the fully qualified paths via protocol schemes such as data, HTTP and others. *SVGPurifier* guarantees that no external references can be loaded by elements allowing script execution. The `<use>` tag on modern Opera browser versions is conversely problematic. This tag can be utilized to include external resources executing JavaScript or providing links with potentially malicious URL handlers.

Currently, *SVGPurifier* scans style elements and attributes of the purified SVG for potentially malicious patterns and neutralizes them by overwriting certain parts of the payload. This includes replacing strings indicating the use of CSS expressions, Opera link and link target properties, as well as, data binding approaches with the placeholder `INVALID`. Listing 5 demonstrates an example for a purification result. Be assured that we do not forbid dangerous tags such as `<set>`, but analyze the attribute values and remove them in case an attack could be initiated by their contents.

```
// before
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" fill="red" cx="30" cy="30
      ">
    <set attributeName="onclick"
      to="javascript:alert(1)//">
    <set attributeName="fill" to="green">
  </circle>
</svg>


// after
<svg xmlns="http://www.w3.org/2000/svg">
<circle r="50" fill="red" cx="30" cy="30">
<set to="INVALID"></set><set attributeName
="fill" to="green"></set></circle></svg>
```

Listing 5: A malicious SVG before and after purification

Section 5 will further build upon the results of this purification process and furnish insight into how far this (for XML data unconventional) approach affects the visual information provided by the SVG test set.

SVGPurifier itself has undergone substantial testing from the security community during a public demonstration over a time-frame of several months [24]. The results helped us to refine the filtering mechanism and spot all the less obvious and difficult to find browser behaviors requiring dedicated fixes to deliver effective filtering and keep the security promise that the tool poses. During the testing phase we logged about 500 attempts targeted to break the filter functionality of the *SVGPurifier* and inject malicious content. Of those, about 15 were successful and resulted in the refinement of our algorithms. The *SVGPurifier* performance scales with the number of SVG tags and elements to sanitize, but can be considered uncritical since the main use case for the tool is on the server-side (e.g., each uploaded SVG image is transformed to remove suspicious content).

A server-side solution has the advantage that a website owner can performantly protect its users from attacks using SVGs and not requiring all users to upgrade their client-software. Our evaluation showed that the SVGPurifier was capable of removing malicious code in all of the discussed test cases. We examined possibilities to craft a purely client-side SVG filter combined with the possibility of limiting down DOM elements and their capabilities. Our initial research showed that this is feasible and considered as future work.

4.3 Unexpected Browser Behavior

We also found several cases of unusual and (depending on the execution context) often problematic browser behaviors that forced us to adapt *SVGPurifier* to address them:

- The Opera AII attacks mentioned in Section 3.5 have not been fixed by the vendor despite several bug reports from our side. This problem complicates the implementation of *SVGPurifier* since basically any external image resource loaded by an SVG file could contain suspicious plugin code and initiate an attack.

- Most browsers support the SVG `<use>` tag, but so far only Opera allows to include external SVG resources containing script code to execute, or links to show and point to possibly malicious URIs via URL handlers such as `javascript` and `data`. Most browsers tested permit utilizing the `<set>` as well as the `<animate>` tag to transform `xlink:href` attributes and set them with *JavaScript* and *data* URIs, too. This should be restricted by browsers for the sake of avoiding injection attacks via `<use>`, `<set>`, and `<animate>`. However, none of the over 100.000 SVG files we have tested during our evaluation actually used this feature.

- Plain text tags inside SVG images such as `<script>`, `<style>`, `<noscript>`, and similar tags allow to use HTML entities, giving them an equivalent syntactical meaning as their canonical forms. This was relevant for several of the XSS filter bypasses we described in Section 3.4. Especially the automated decoding of entities such as `&lt;` and `&gt;` could be used to bypass XSS filters and common protection mechanisms. Browsers therefore need to be more selective in determining which entities get automatically decoded and which do not. For example, Google Chrome went as far as to completely disable the automated decoding.

Some of these behaviors forced us to customize *SVGPurifier* and reduce the available feature set. Later versions of the tool might be less restrictive, assuming that the browser bugs get fixed and the market share of the affected browser versions shrinks to an insignificant level. Future versions of the tool will also be capable of handling external entities and place their proper value at the desired locations.

5. EVALUATION

We have implemented a prototype of *SVGPurifier*, consisting of 5,663 lines of PHP code. To evaluate the tool, we compiled a test set of SVG images obtained from Wikipedia. We chose this platform for several reasons: SVG images are widely used within Wikipedia, the content of the platform consists of the contributions from a large community, and among the contributors, the employment of a diverse set of tools to create the images can be observed. As a result, we have a heterogeneous test set that enables us to study the robustness and versatility of *SVGPurifier*.

To download the files, we used *wikix*, a tool that uses a snapshot of Wikipedia (exported as XML) to generate the URLs of the SVG files hosted at `upload.wikimedia.org`. The latest snapshot of the english Wikipedia at the time of writing has referenced a total of 112,646 SVG images. 105,509 were actually available for download at that time and we used those files for our evaluation.

5.1 Evaluation Setup

In order to minimize the impact of *SVGPurifier* on usability, the tool should not alter the visual appearance of an image since this would decrease the user experience. Our implementation only removes elements from an SVG file, thus (by construction) no new image element will appear in a purified image. However, the cleaning process might be too aggressive, i.e., cases where we remove elements that have a visual impact on the resulting image might occur. To evaluate this effect, we compare the original image with the purified one and determine if the file was altered during the process. Since the size of our test set is too large for a manual evaluation, we developed an approach to compare SVG files in an automated way.

Comparing two SVG images for similarity is difficult, resulting from the fact that there are countless ways of achieving the same visual appearance. Consequently, contrasting only the XML markup does not enable us to determine if a pair of images has the same visual appearance. Therefore, we decided to convert the SVG files to *Portable Network Graphics* (PNG) format and then perform the comparative step. PNG is a raster graphics format providing lossless data compression. Each pixel is defined in an 24 bit RGB colorspace with an optional 8 bit alpha channel (32 bit RGBA). Comparing two PNG files for similarity can thus be achieved by matching the value of each channel for each pair of pixels, which in turn results in a numerical value representing the absolute error a. A value of $a = 0$ indicates that no difference between the two images was found, while $a > 0$ denotes some discrepancy. Note that this evaluation measures the visual impact of our tool and approximates the deviation caused by the transformation process.

We tested the following four tools regarding their capability to convert SVG images to PNG format:

- Apache *batik* (`http://xml.apache.org/batik/`)
- *rSVG* (`http://librsvg.sourceforge.net/`)
- *GIMP* (`http://www.gimp.org/`)
- *Inkscape* (`http://inkscape.org/`)

Based on our test set, we have empirically found that the Apache *batik* toolkit was able to convert the largest number of files: only 23 of the 105,509 files could not be converted by the tool, prompting us to remove them from the test set. All files from the resulting evaluation set were converted to a PNG image with a fixed width, assuring the aspect ratio's preservation. As *batik* does not fully support declarative animations, we create static PNG images from the SVG files. In this manner, if an image was animated beforehand, we will only consider the visual state it is in before the animation begins. There are five different elements within a SVG file to accomplish animation: `<set>`, `<animate>`, `<animateMotion>`, `<animateColor>`, and `<animateTransform>`. A close analysis showed that only two SVG files in our test set actually contained one of these elements, which indicates that this feature is not (yet) widely used. We therefore consider comparison of static images exclusively to be only a minor limitation of our evaluation.

After converting images to PNGs, we compare the original and the purified image using the *ImageMagick* toolkit, which provides methods for a pixel-wise comparison of raster images, as well as, for gathering statistics on the amount of aberration. Furthermore, the tool is capable of creating *difference images* that visually indicate what regions of an image contain an error, which eventually enables us to manually examine cases in which actual changes occur. Other than determining the absolute number of pixels that differ between the two images, we calculate the normalized mean absolute error for each pair of images as metrics. We consider the normalized mean absolute error to be more relevant in our scenario than the root mean square error, as the former weights every aberration equally. Moreover, we log both the size of the original file, and that of the purified one, factually determining the compression ratio resulting from the purification process.

5.2 Evaluation Results

Based on the procedure outlined above, we analyzed all 105,486 files belonging to the evaluation set. For 98.5% of the samples, no visual difference exists in the appearance of the original versus the purified image (i.e., absolute error a is 0 and, therefore, all other error metrics are 0 as well). The 99th percentile of the normalized mean absolute error is 0.00000474877, where 0 represents no error at all and 1 indicates completely dissimilar images.

When investigating the error cases and the corresponding different images, we often found that although the image contained some kind of aberration that can be expressed numerically, a visual difference cannot be easily found by a human observer. Specifically, we manually analyzed 1,000 test cases in which the absolute error was larger than zero. We tried to determine in how many cases a human observer would notice an aberration. While this is not believed to be an approach that is valid overall (i.e., specific circumstances like medical applications require precise conversion), a visual impact in the context of a website exists only if a user can actually spot it.

During the manual inspection process, we spent about 10 seconds on each pair of the images to compare them (aided by the *difference image* to support the user). The results of this manual examination indicate that only 46.3% of the erroneous samples were perceived as different from their original. Since user experience may differ, we provide a website where we present all defective images complete with their original and the *difference images*, inviting others to freely inspect these cases [6].

Several side effects were observed during the purification process and the evaluation of this process' results. What was an additional positive side effect we found, was that due to the removal of elements not contributing to the visual appearance of the image, *SVGPurifier* actually compresses files with an average compression ratio of 2.6. Some files had a slightly larger file size after the purification process, which was caused by the transformation the tool has performed on broken files. In most cases, the increase in file size was based on the addition of missing closing tags by *SVGPurifier*.

1.59% of the files in our test set contained one or more instances of the `<foreignObject>` element. In the majority of the 1,686 cases, this element was used as what had appeared to be an artifact of *Adobe Illustrator* to store a base64-encoded representation of its proprietary AI file format within the SVG file. *SVGPurifier* deleted these elements and no visual impact resulted from this removal. However, the image size was reduced significantly.

6. RELATED WORK

Not surprisigly, being one of the most common problems in the area of web security, the Cross Site Scripting (XSS) problem has received a lot of attention during the last decade. [11, 12, 20, 27–29, 31, 36, 39, 40, 42]. On the offensive side, several different kinds of attacks were studied [12, 28, 31]. Approaches to prevent XSS attacks include information flow and taint tracking [20, 36, 39, 40], and analysis on the client- or server-side [11, 29, 40]. John's dissertation elaborated on the attack and defense techniques in detail [27], while Phung et al. presented specific defense techniques against client side and JavaScript based attacks [38]. Nevertheless, none of these works dealt with the threat of malicious image files in the JavaScript execution context. In this paper, we introduce new innovative attacks that highlight the fact that in the era of HTML5, even a previously unsuspicious `<img>` tag may introduce security vulnerabilities due to the tight integration of SVG images into the modern browsers.

One exception is the work by Barth et al., who discussed attacks and mitigations around faulty and jaunty content sniffing [9]. Deprecated browsers such as Opera 9 and Internet Explorer 6 allowed to execute JavaScript by combining image tags with JavaScript URIs, but none of the tested modern browsers supports this kind of render behavior anymore, as this particular attack vector has been recently fixed. In contrast, the risks of Cross Site Scripting and related attacks against browsers induced by SVG images have not yet been investigated.

SVG as a subject itself surfaced rarely in the scientific security community. One notable exception is given by Damiani et al. [18], who dealt with access control requirements of parts of SVG files. Their assumption was that SVG files containing sensitive personal information should be rendered differently for different viewers, hence requiring some parts of an SVG document to be deleted (or kept encrypted). However, they did not manage to cope with the threat of misusing SVG files as attack vectors. In the same line of work, Mohammed et al. [33–35] investigated the use of SVG images in medical contexts, where certain security guarantees have to be granted for sensitive information contained in an SVG image. Again, their publication did not resolve the offensive use of SVG files.

One area of research closely related to the results presented in this paper deals with the problem of code embedded in document formats. For example, Backes et al. showed that maliciously prepared PostScript files can be used as an attacker vector [8], and Checkoway et al. discussed malicious TEX files that can, among other consequences, lead to an arbitrary code execution and data exfiltration based on TEX's Turing-complete macro language [13]. Even pure text files might contain shellcode as shown by Mason et al. [32]. We continue this line of scientific enquiry and present attacks related to SVG images.

An orthogonal area of research are alternative browser designs [10,15,19,41]. These browsers explore how the security of state-of-the-art browsers can be improved, for example by creating separate protection domains. The results presented in this paper need to be taken into account when designing more secure browser and especially the fact that `<img>` tags might lead to suspicious content have to be considered.

7. CONCLUSION

In this paper, we provide an overview of Scalable Vector Graphics (SVG) and their security impact on the World Wide Web as based on the new HTML5 specification drafts. We show that this image format (which exists for more than a decade), significantly changes the browser and web security landscape. We introduce several novel attacks against modern browsers and show that this phenomenon can have major impact on web applications that allow their users to post images. In particular, we illustrate that SVG images embedded via `<img>` tag and CSS can execute arbitrary JavaScript code and similar attacks. Subsequently, the discussed XSS filter bypasses, which work against several browsers, can have a similarly high impact on a targeted attack scenario.

To mitigate the attacks presented, we proposed *SVGPurifier* as a first practical solution available and capable of removing potentially malicious code from SVG files. We have empirically shown that the software is usable for real-world scenarios such as a purification of the SVG files stored by Wikipedia. We are in a discussion with the Wikipedia team, who might adopt *SVGPurifier* to their infrastructure. Furthermore, many of the identified attacks have already been fixed by major browser vendors.

8. REFERENCES

[1] National vulnerability database (NVD) (CVE-2007-1765). `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-1765`, Mar. 2007.

[2] National vulnerability database (NVD) (CVE-2008-3702). `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-3702`, Aug. 2008.

[3] Fonts – SVG 1.1 (Second Edition). `http://www.w3.org/TR/SVG/fonts.html`, June 2010.

[4] National vulnerability database (NVD) (CVE-2010-3113). `http://web.nvd.nist.gov/view/vuln/detail?vulnId=cve-2010-3113`, Aug. 2010.

[5] Scalable vector graphics (SVG) 1.1 (Second edition). `http://www.w3.org/TR/SVG11/`, June 2010.

[6] Svgpurifier: inaccurately converted images. `http://svgpurifier.nds.rub.de/`, May 2011.

[7] Adobe Systems Inc. Illustrator 10 XML Extensions Guide, Sept. 2001.

[8] M. Backes, M. Durmuth, and D. Unruh. Information Flow in the Peer-Reviewing Process. In *IEEE Symposium on Security and Privacy*, 2007.

[9] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *IEEE Symposium on Security and Privacy*, 2009.

[10] A. Barth, C. Jackson, C. Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser, 2008. `http://seclab.stanford.edu/websec/chromium/`.

[11] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2008.

[12] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross Channel Scripting and its Impact on Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[13] S. Checkoway, H. Shacham, and E. Rescorla. Are Text-only Data Formats Safe? or, Use This LaTeX Class File to Pwn Your Computer. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2010.

[14] J. Clark. XSL transformations (XSLT). `http://www.w3.org/TR/xslt`, Nov. 1999.

[15] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[16] A. Dabirsiaghi. The OWASP AntiSamy project. `http://code.google.com/p/owaspantisamy/`, Apr. 2011.

[17] E. Dahlström. SVG and HTML. `http://dev.w3.org/SVG/proposals/svg-html/svg-html-proposal.html`, July 2008.

[18] E. Damiani, S. De Capitani di Vimercati, E. Fernandez-Medina, and P. Samarati. An access control system for SVG documents. *King's College, University of Cambridge, UK*, pages 29–31, 2002.

[19] C. Grier, S. Tang, and S. T. King. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy*, 2008.

[20] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[21] U. Harnhammar. kses - PHP HTML/XHTML filter. `http://sourceforge.net/projects/kses/`, Mar. 2010.

[22] M. Heiderich. Opera SVG AII testcase. `http://heideri.ch/opera/`, 2011.

[23] M. Heiderich. SVG chameleon via XSLT - HTML5 Security Cheatsheet. `http://html5sec.org/#125`, Mar. 2011.

[24] M. Heiderich and T. Frosch. SVGpurifier smoketest. `http://heideri.ch/svgpurifier/SVGPurifier/`, Apr. 2011.

[25] I. Hickson. HTML standard — the map element. `http://whatwg.org/specs/web-apps/current-work/multipage/the-map-element.html#svg-0`, Apr. 2011.

[26] L. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from Cross-Origin CSS attacks. In *ACM Conference on Computer and Communications Security (CCS) 2010)*, 2010.

[27] M. Johns. *Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting*. PhD thesis, University of Passau, Passau, July 2009.

[28] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic Pharming Attacks and Locked Same-Origin Policies for Web Browsers. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[29] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *ACM Symposium On Applied Computing (SAC)*, 2006.

[30] E. Lawrence. Same origin policy part 1: No peeking. `http://blogs.msdn.com/b/ieinternals/archive/2009/08/28/explaining-same-origin-policy-part-1-deny-read.aspx`, Aug. 2009.

[31] M. Martin and M. S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking. In *USENIX Security Symposium*, 2008.

[32] J. Mason, S. Small, F. Monrose, and G. MacManus. English Shellcode. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[33] S. Mohammed, L. Chamarette, J. Fiaidhi, and S. Osborn. A Safe RSS Approach for Securely Sharing Mobile SVG Biomedical Images for Web 2.0. In *12th IEEE International Conference on Computational Science and Engineering*, 2009.

[34] S. Mohammed, J. Fiaidhi, H. Ghenniwa, and M. Hahn. Developing a Secure Web Service Architecture for SVG Image Delivery. *Journal of Computer Science*, 2(2):171–179, 2006.

[35] S. M. A. Mohammed and J. A. W. Fiadhi. Developing Secure Transcoding Intermediary for SVG Medical Images within Peer-to-Peer Ubiquitous Environment. In *CNSR '05 Proceedings of the 3rd Annual Communication Networks and Services Research Conference*, 2005.

[36] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[37] S. Patnaik. htmLawed. `http://www.bioinformatics.org/phplabware/internal_utilities/htmLawed/`.

[38] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.

[39] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[40] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2007.

[41] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium*, 2009.

[42] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2008.

[43] E. Z. Yang. HTML Purifier. `http://htmlpurifier.org/`, Mar. 2011.