SECURITY

Automating XSS detection in the CI/CD pipeline with XSS-Checkmate

Learn this new security fuzz testing technique that leverages browser capabilities to detect cross-site scripting vulnerabilities before production deployment.

By Binu Ramakrishnan. January 10, 2017

Board Game (source: Prexels via Pixabay)

For more on secure deployments, check out Binu Ramakrishnan's talk "Securing application deployments in CI/CD environments."

Integrating cross-site scripting (XSS) tests into the continuous integration and continuous delivery (CI/CD) pipeline is an effective way for development teams to identify and fix XSS vulnerabilities early in the software development lifecycle. However, due to the

nature of the vulnerability, automating XSS detection in the build pipeline has always been a challenge. A common practice is to employ dynamic web vulnerability scanners that perform blind black-box testing on newly built web application, before production deployment. There is another way to detect XSS vulnerabilities with web applications in CI/CD: XSS-Checkmate is a security fuzz testing technique (not a tool) that complements traditional dynamic scanners by leveraging browser capabilities to detect XSS vulnerabilities. Since modern web applications widely use tools such as Selenium, Cucumber, and Sauce Labs to test user interactions, this technique can be integrated into those test frameworks with less effort.

What is XSS?

XSS is one of the oldest types of security vulnerabilities found in web applications. It enables execution of an attacker-injected malicious script when a victim visits an infected page. The primary reason for XSS is the improper neutralization of user inputs when rendered on a web page. XSS has remained a top threat on OWASP's top ten list since its first publication in 2004.

There are 3 classes of XSS attacks:

1. Reflective XSS: Untrusted input passed as part of an HTTP request is reflected back in the HTTP response from the server. The reflected data is mistakenly treated as code and executed in the browser, causing reflective XSS.
2. Stored XSS: This is similar to reflective XSS, but the untrusted data get stored in the server (e.g., in the database) and get rendered in web pages served to other users, or may affect multiple web pages.
3. DOM XSS: This is direct manipulation of the browser DOM. Here, the untrusted user inputs get directly injected into the page's DOM by a client-side script.

Output encoding is the primary mitigation technique to address content injections in web based applications. Encoding *neutralizes* the input based on the rendering context. For instance, you use HTML encoding to neutralize untrusted data injected into an HTML context.

How XSS-Checkmate works

XSS is a vulnerability that occurs when the data get (mis)interpreted as code and executed on a victim's browser. The idea is to use a headless browser like Selenium WebDriver, and inject XSS payloads along with functional and user interaction tests. For example, to test the compose feature of a web mail, you can inject data into the *From*, *To*, *Subject*, and *Content* fields and compare it with a known end state. To enable XSS testing, you could additionally pass XSS payloads to these fields to detect XSS, if any.

In software testing parlance, the *assert* function is the widely used method to test application features by comparing actual outcome with expected outcome. Though the direct use of *assert* to detect XSS is not new, it is proved to be less effective in detecting XSS. The challenge with the direct *assert* approach is that the injected data flow to many different places. This data often get stored in backend servers and appear in multiple web contexts, asynchronously, making the direct *assert* based comparisons less useful for XSS detection tests.

To overcome the above limitation, you can leverage the basic nature of XSS—execution of JavaScript code. As many of you know, a common proof-of-concept pattern to demonstrate an XSS vulnerability is to execute `javascript:alert()`. XSS-Checkmate

takes a similar approach by embedding `console.error('testid')` in the payload, and watching the browser console. Error messages appearing in the browser console indicate execution of XSS payload in some context. In such cases, the build should fail and the developer must fix the issue to continue the build. XSS-Checkmate relies on the fact that the browser is the best tool to test content injection vulnerabilities. By injecting carefully crafted XSS payloads into the web application executed on a headless browser, we can detect XSS vulnerabilities.

In general, security testing should be considered part of application quality testing. Unlike dynamic web vulnerability scanners that perform black-box testing, XSS-Checkmate is part of the user interaction tests performed during the build and deployment (CI/CD) phase. Developers can enable XSS detection tests for features they develop. By moving these tests closer to regular application tests, all flows are likely to get tested with XSS payloads, including complex user interaction flows on modern web applications. The unique identifier used in the payload helps developers to track down vulnerable code in less time and with less effort.

Polyglot payloads

The effectiveness of an XSS payload is dependent on its ability to execute on the contexts where the payload is rendered/injected on the browser. The data injected to an input field may appear multiple places on a web page. To cover all rendering contexts, you need to use a wide variety of payloads. But the more input you add, the more time it takes to complete the tests. XSS polyglots can come in handy here to save a lot of time and effort. By using polyglots you can significantly reduce the number of XSS payloads and improve the effectiveness of the testing. So what is it? A polyglot payload is an XSS vector which can be executed in multiple injection contexts.

Here is an example of an XSS polyglot payload:

```
"><script>alert(1)</script>"
```

The above polyglot payload works at least in the following two injection contexts:

```
<div></div>                    [HTML Body context]
<div class="">text</div>       [HTML Attribute value context]
```

To learn more about polyglot XSS payloads, see the links in the XSS Payloads box below.

XSS payloads

Carefully crafted XSS payloads are important to get better coverage with fewer tests. It is fairly easy to add a large number of attack vectors, but the the downside is that your test can take more time to complete, often with less than desired coverage.

XSS polyglot payloads:

- https://blog.bugcrowd.com/xss-polyglots-the-context-contest
- http://polyglot.innerht.ml/
- https://github.com/jhaddix/tbhm/blob/master/5_XSS.markdown
- https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot
- https://github.com/danielmiessler/SecLists/tree/master/Fuzzing/Polyglots

XSS payloads:

- http://hackingforsecurity.blogspot.com/2013/11/xss-cheat-sheet-huge-list.html
- https://deadliestwebattacks.com/html-injection-quick-reference/

Measuring success

For dynamic web vulnerability scanners, the success criteria are mostly dependent on the ability to discover XSS on a site. But for XSS-Checkmate, the success is based on two criteria below:

1. Code and functionality coverage
2. XSS payload coverage

Since XSS detection tests are not different from other user interaction tests, code and functionality coverage is an important factor determining the effectiveness of this technique. Similarly, payload coverage also plays an important role. Careful selection of XSS polyglots boosts payload coverage, while reducing the number of test payloads.

Examples

To demonstrate XSS-Checkmate in action, I'll use Gruyere XSS codelab. Gruyere provides a vulnerable web application, which can be used to learn penetration testing by exploiting vulnerabilities. A Selenium Python program is provided below.

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.desired_capabilities import
DesiredCapabilities
from string import Template
import unittest, json

CHROMEDRIVER = './chromedriver'
GRUYERE_ID = '932695469192'
TESTID = '4441'
urls = [ """http://google-gruyere.appspot.com/${GID}/${INPUT}""" ]
payloads = [ """<script>console.error(${TID})</script>""",
"""console.error(${TID})//<svg/onload=console.error(${TID})>'-
console.error(${TID})-'""" ]

class XSSTest(unittest.TestCase):
   def setUp(self):
       d = DesiredCapabilities.CHROME
       d['loggingPrefs'] = { 'browser':'ALL' }
       chrome_options = Options()
       chrome_options.add_argument("--disable-web-security")
       chrome_options.add_argument("--disable-xss-auditor")
       self.driver = webdriver.Chrome('./chromedriver',
chrome_options=chrome_options)

   def tearDown(self):
```

```
        self.result()
        self.driver.quit()

    def test_run(self):
        id = int(TESTID)*10;
        for url in urls:
            for xss in payloads:
                xss1 = Template(xss).substitute(TID=str(id))
                url1 = Template(url).substitute(GID=GRUYERE_ID,
INPUT=xss1)
                print url1
                id = id + 1
                self.driver.get(url1)

    def result(self):
      n = 0
      for cmsg in self.driver.get_log('browser'):
          if -1 != str(cmsg).find(" " + TESTID):
            print(json.dumps(cmsg))
            n += 1

      self.assertEqual(0, n)

# entry point
if __name__ == "__main__":
 unittest.main()
```

The program makes calls to Gruyere URL endpoints with XSS payloads and watches the browser console for the execution of injected script. Violation logs appearing in the browser console are an indication of XSS vulnerability. Optionally, the violation logs can be sent to systems like Splunk for metrics and to gain visibility.

The console log output for the above program is shown below:

```
{ "source": "console-api",
 "message": "http://google-
gruyere.appspot.com/932695469192/%3Cscript%3Econsole.error(4440)%3C/script%3
 141:55 4440",
```

```
 "timestamp": 1480881814713,
 "level": "SEVERE"
}


{ "source": "console-api",
 "message": "http://google-
gruyere.appspot.com/932695469192/console.error(4441)//%3Csvg/onload=console.
console.error(4441)-' 141:107 4441",
 "timestamp": 1480881815456,
 "level": "SEVERE"
}
E
======================================================================
ERROR: test_run (__main__.XSSTest)
----------------------------------------------------------------------
Traceback (most recent call last):
 File "xsscheckmate-gruyere.py", line 39, in tearDown
   self.result()
 File "xsscheckmate-gruyere.py", line 59, in result
   self.assertEqual(0, n)
AssertionError: 0 != 2

----------------------------------------------------------------------
Ran 1 test in 3.256s

FAILED (errors=1)
```

You may find the complete source code here.

xss-polyglots

xss-polyglots is a Node Packaged Module (npm) package that provides a set of malicious payloads for XSS testing. This package exports a function that returns an array of polyglot payloads.

```
import getPayloads from 'xss-polyglots';

const xssPayloads = getPayloads();
xssPayloads.forEach(payload => {
   testRenderingComponent(payload);
});
```

By default, these payloads are embedded with `console.error` call. However, you can also specify a custom JavaScript function by passing it as the first argument of the `getPayloads` function.

```
import getPayloads from 'xss-polyglots';
window.testInvalidCallback = sinon.stub();

const xssPayloads = getPayloads('testInvalidCallback');
expect(testInvalidCallback).to.not.have.been.called;
```

Content Security Policy

Content Security Policy (CSP) is a browser mechanism to detect and block content injection attacks on web applications. Since the CSP violation reports also appear in the browser console, it can be handled the same way as XSS detection tests violation logs (see my CSP talk here). When CSP is *enforced* on a webpage that contains an XSS vulnerability, it blocks the execution of the payload and generates a CSP violation report. The error message from payload execution doesn't show up in the browser console because its execution is blocked by CSP—instead you see the CSP violation report in the browser console. In practice, adopting XSS-Checkmate with CSP means you should be prepared to handle both message types in your browser console.

Enhancements

You can take automation to the next level by recording all user interaction performed as part of UI tests, and replay the recorded steps with XSS payloads. If successful, this will provide you a tool that automatically runs after UI tests, and relieves developers from writing XSS detection tests.

Recording user interactions on a page object is possible with browser extensions, but with few drawbacks. Many web applications generate HTML dynamically and use random names for tag identifiers, making it hard for web drivers to locate those tags and elements from page objects. The solution is to locate tags and elements by leveraging XPath or CSS. The universal applicability of such options are not verified yet.

Conclusion

XSS-Checkmate is an effective mechanism for web applications to detect XSS vulnerabilities as part of their user interaction tests in CI/CD pipeline. By using carefully crafted XSS polyglot payloads you can improve the effectiveness of the testing and significantly shorten the testing time by reducing the number of XSS payloads. The current model is based on developers writing XSS detection tests along with their regular UI tests, and the additional effort required for writing those tests is marginal. Future improvements are directed towards replacing developer-written XSS detection tests with browser extensions that can record and replay user interaction tests with XSS payloads.

Article image: Board Game (source: Prexels via Pixabay).

Binu Ramakrishnan

Binu Ramakrishnan is a principal security engineer at Yahoo with over a decade of experience in building Internet-scale systems, anti-abuse systems, and application security. He currently leads security engagements in Yahoo mail, working closely with product engineers and leaders to help define and implement strategic security programs. Binu is an active participant in the industry-wide initiative to secure mail delivery infrastructure and contributed to the recent SMTP MTA-STS efforts. Follow him on Twitter @securitysauce.

Cory Doctorow on the problems with Encrypted Media Extensions

By Courtney Nash

Confronting the World Wide Web Consortium on the new digital rights management specification.

SECURITY

HTTPS is coming. Are you prepared?

By Zack Tollman

Zack Tollman explores the key aspects of HTTPS to help developers to take control of their HTTPS-only sites.

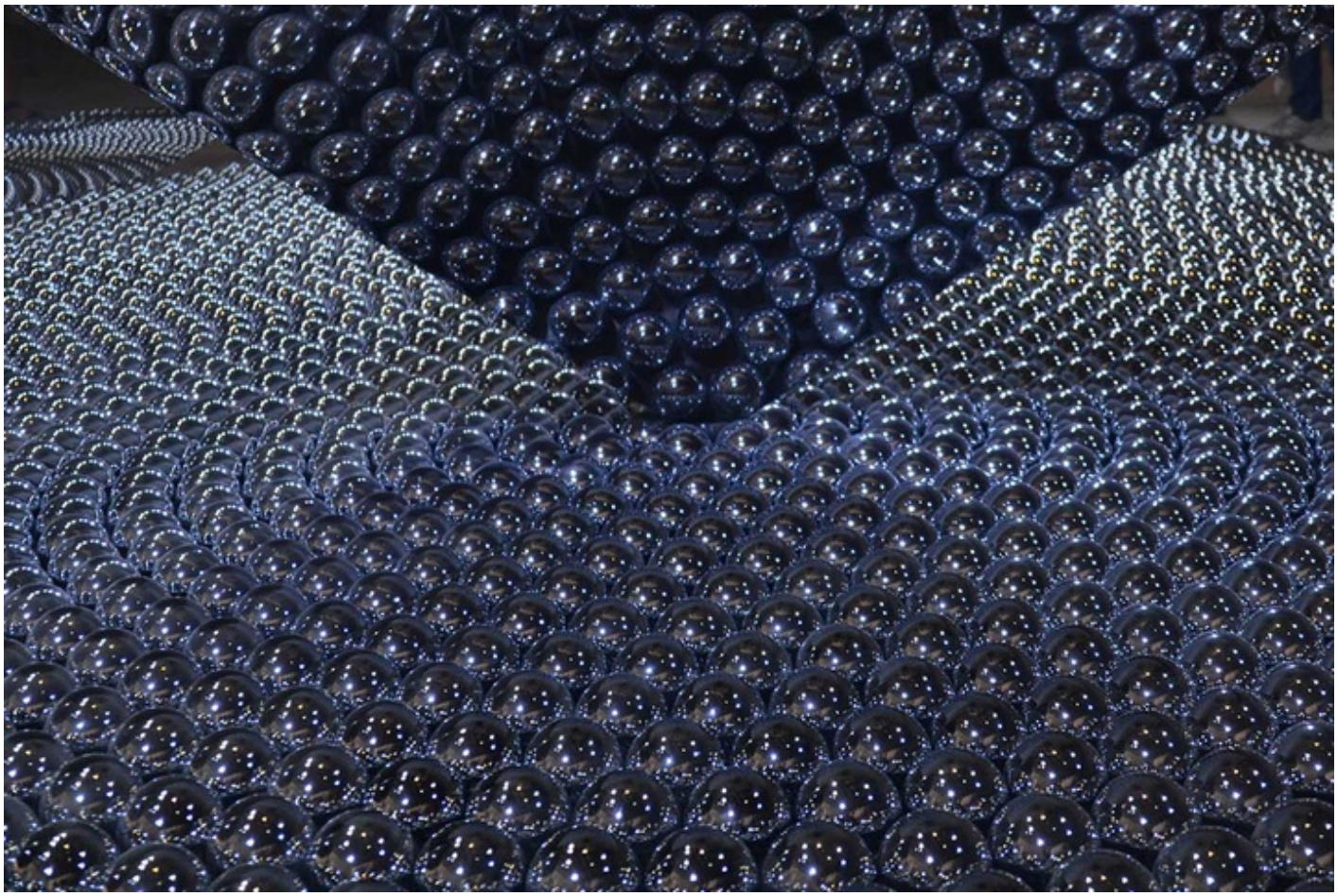
SECURITY

Injecting security into Continuous Delivery

By Jim Bird

How to build security in as an essential part of your workflow.

SECURITY



Balancing frontend security and performance with HTTP Strict Transport Security

By Sabrina Burney and Sonia Burney

What is HTTP Strict Transport Security and why should you use it?