WHITE PAPER

DOM-Based Cross-Site Scripting

Travis Biehn, Ksenia Peguero and John Steven

Table of contents

Executive summary

DOM-based attacks are a misunderstood, serious, and pervasive source of risk in contemporary web applications. The language that drives the web, JavaScript, is easy to understand and hard to master; junior and senior developers routinely make mistakes. Mix difficulty to master with an enormous attack surface, and you have the perfect storm for widespread vulnerability. These risks expose web applications to threats similar to well-understood cross-site scripting (XSS) vulnerabilities.

The nebulous and imprecise definition of DOM-based XSS makes discovery and management of these issues harder. Recently proposed naming changes, such as “client-side reflected” or “Type 0,” still miss the point. At their simplest, DOM-based attacks are those that, due to the way browser-based application code is written, allow malicious code to steal or manipulate user data and execute application functionality. This leads to serious attacks, including the ability to impersonate users without their knowledge.

DOM-based vulnerabilities cannot effectively be discovered through standard white box or black box testing. Unlike traditional XSS, DOM-based problems are not effectively discovered using some of today’s most widely deployed tools. This is a blind spot that shows up as a lack of findings in assessments.

Finding and fixing individual instances, or through integration with “magic bullet” security APIs, is not enough to secure an application’s use of browser-based DOM. Organizations also must use reinforcing approaches to prevent and mitigate risk. This includes specifying Content Security Policy (CSP) for an application, which enables a browser-enforced policy to proactively restrict places that application code and other resources may originate from. Another approach is using freely available libraries like Caja, which, when deployed with an application, restrict that application’s use of the myriad of unsafe JavaScript capabilities that bootstrap and run malware.

CSP dramatically reduces the opportunity to expose an application to injection. Enforcing safe-language subsets provided by Caja nearly eliminates an application’s vulnerable attack surface. Used in concert, these security controls serve to inoculate an application against attack without imposing the large amount of programming work other methods require. As a result, this approach reduces the opportunity for programmers to ignore or fail to correctly apply security guidance. Secure patterns for identity access management (IAM), such as adaptive access control, perform risk-appropriate authentication and fine-grained, context-aware authorization based on historical data and real-time analytics of the user’s behavior. Used together with CSP and safe-language subsets, including Caja, they provide protection across an application as well as a high degree of confidence that only proper users can access targeted sensitive data or privileged transactions.

Exposure

When popularized in the early 2000s, cross-site scripting attacks were named and classified by many practitioners in terms of how exposure occurred—for example, reflected XSS (#1) vs. stored XSS (#2). This nomenclature continues with the so-called DOM-based, or more recently named "client-side reflected," attacks. At the time, researchers focused on attacks where a threat agent previously injected a payload into a database through SQL injection or a simple lack of validation (so-called stored attacks). They also focused on attacks in which a payload received through web requests was initially delivered to a victim by schemes such as phishing or clickjacking (so-called reflected attacks). Figure 1 shows this naïve view.

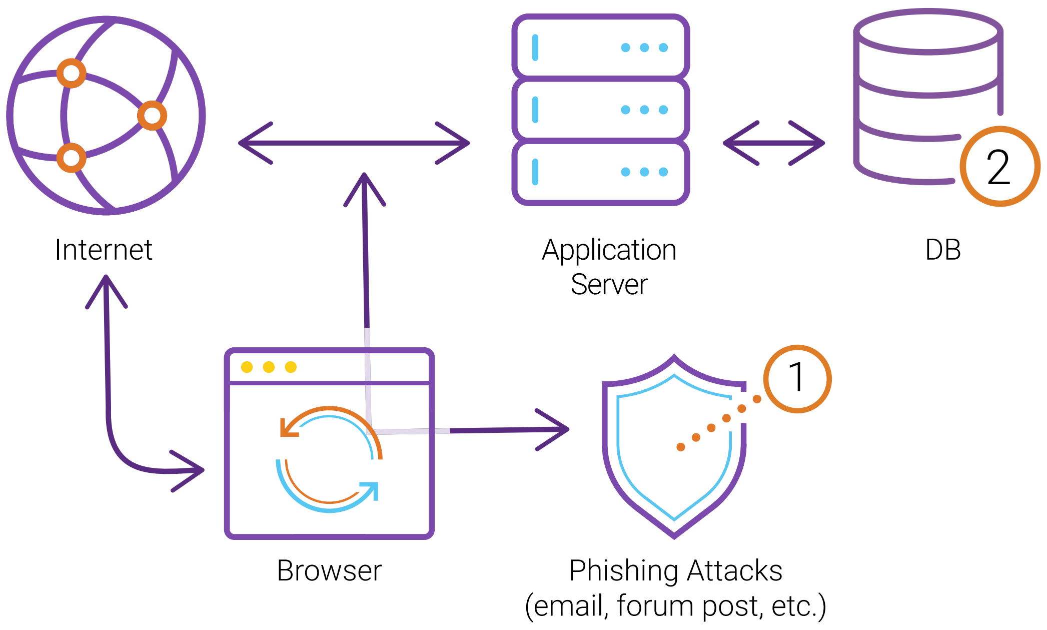

Figure 1. Early application security view of XSS

DOM-based XSS exposure is injecting a malicious script into application functionality that is already manipulating a browser DOM. Defending against this attack proves challenging because the targeted constructs, such as `document.write()` and `innerHTML`, appear normal and exist prevalently in contemporary web applications. It is important to note that DOM-based XSS attacks can also be reflected or stored. In the case of a reflected DOM-based XSS, the payload is usually sent as part of the URL (for example, as the fragment identifier after the # sign) and is immediately processed by the JavaScript running in the browser. However, the Web Storage functionality introduced by HTML5 now allows developers to save data on the client side. This allows an attacker to save a malicious payload in a localStorage object, for example. This payload can be retrieved from the localStorage at a later time and written into the webpage by JavaScript. This may result in a cross-site scripting attack in which the payload is stored on the client side. Therefore, we can depict the types of cross-site scripting in a quadrant, where one axis determines if the payload was stored or reflected right away, and another if the exposure occurred on the server or on the client side.

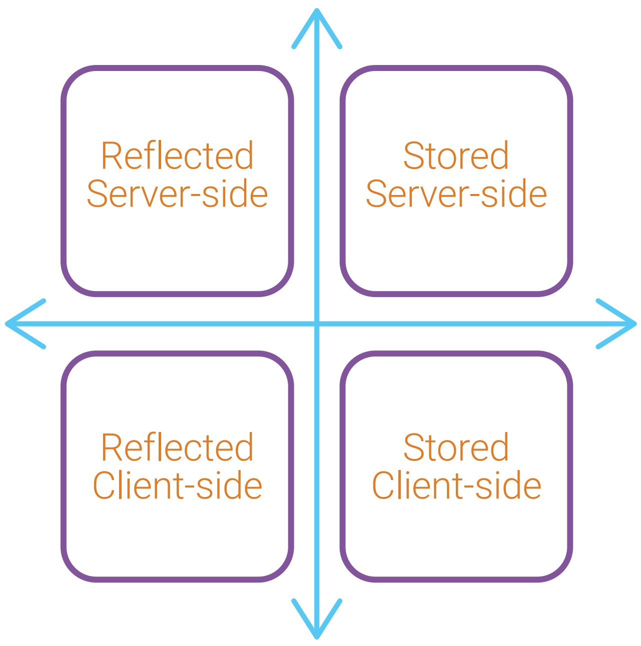

Figure 2. Types of XSS

Many other sources of exposure also exist. For instance, attackers can trick third-party web application services into serving malicious payloads (see Figure 3, #3 and #4). New browser-based data stores are introduced with HTML5 (Figure 3, #5). Web services communicating using JSON-based and XML-based calls provide other opportunities for code injection (JSON injection, XML entity expansion, XML external entity attack, etc.).

Code that is mobile and crosses trust boundaries is commonplace in contemporary web applications. In order to be successful, organizations must deal with the problem of code injection at all points of exposure rather than focusing on one—only to be surprised by the next.

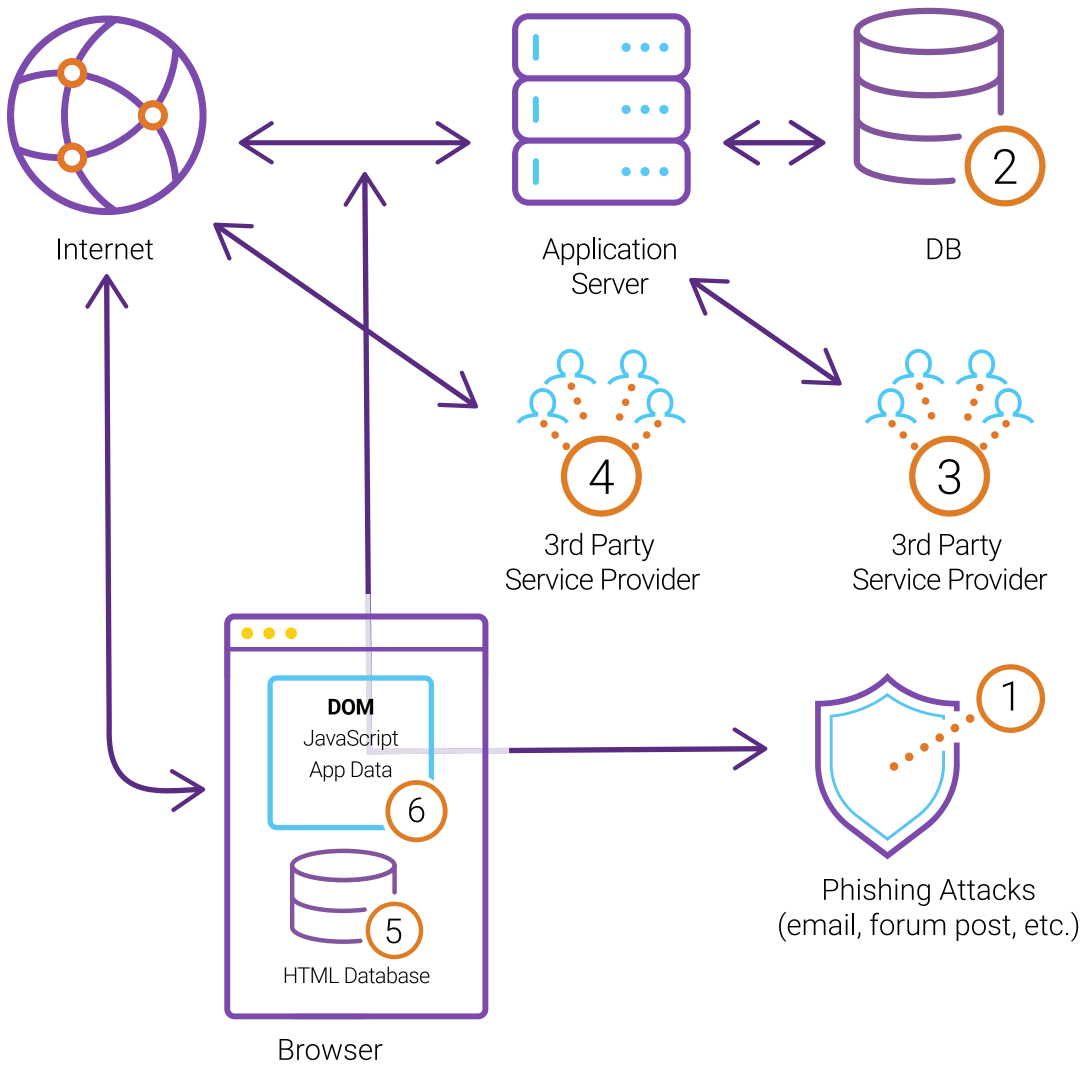
Internet
Application Server
DB
2
4
3rd Party Service Provider
3
3rd Party Service Provider
DOM
JavaScript
App Data
6
5
HTML Database
Browser
1
Phishing Attacks (email, forum post, etc.)

Figure 3. Modern application security view of XSS

Pathology

As with other varieties of XSS, exposure to DOM-based attacks allows malicious scripts to execute within the application running in a web browser. Once exposed, a malicious script can manipulate an application's state in addition to making requests. An application's browser-based DOM embodies so much application functionality, state, and sensitive customer data that simply manipulating the DOM can be as impactful as traditional attacks on HTTP requests, such as classical cross-site request forgery. Just as a request's parameters can be tampered with, data within the DOM can be changed in order to replace user IDs, prices, and balances or lift sensitive customer data. However, these attacks may be more challenging to detect and prevent simultaneously.

Assessment techniques

The dynamic nature of JavaScript prevents us from exhaustively finding injection vulnerabilities using standard static and dynamic methods. Let's take a closer look at the two traditional methods of application testing.

Dynamic application security testing (DAST), also known as "black box testing," is traditionally based on the concept of sending numerous HTTP requests with different payloads to the server and observing the changes in the responses. For example, in a traditional, server-side reflected XSS attack, the payload is sent as one of the request parameters, and the vulnerability is detected by finding that same payload in the body of the response without any modifications. This method of testing is not effective for DOM-based XSS, because the payload may not even reach the server (if it's part of the fragment identifier), or it will not be written directly to the webpage in the response. On the contrary, it may be contained in some other place, such as a cookie, an HTTP header, or another URL, which is later processed by the JavaScript in the browser and executed.

Since JavaScript is loaded to the client to be executed, pure black box testing does not make sense. Some tools perform something called "gray box testing," a combination of dynamic and static analysis. However, let's first look at the problems of performing static analysis, or "white box testing," on JavaScript.

Many **static application security testing** (SAST) tools claim to perform automatic secure code review on JavaScript; however, the results are usually worse than we expect. In most cases, static analysis tools perform pattern-based analysis looking for JavaScript sinks, like `document.write()`, `innerHTML, eval()`, `execScript()`, and others. Some tools can also identify sources of data, such as `document.cookie`, `window.name`, and `event.data`.

However, performing the actual dataflow and linking tainted sources with the sinks is an extremely complicated task. JavaScript analysis continues to be an unsolved problem. So far, researchers in this area have had incomplete results, and little has been released. Existing commercial tools, like AppScan Source and Fortify, and free JavaScript-specific tools, like ScanJS and JSPrime, help to identify possible problematic places, like JavaScript sinks, but leave the in-depth dataflow analysis to the human. Moreover, most tools, like ScanJS and AppScan Source, only support native JavaScript and don't take into consideration widely used frameworks, like jQuery, YUI, and AngularJS. In turn, these frameworks introduce new sources and sinks into the code. Therefore, tools will return many false positives and true negatives. In other words, they'll miss real vulnerabilities.

Another complication of scanning JavaScript is the presence of obfuscated and compressed files, leading static tools to then face the challenge of attempting to virtualize or execute code. Further compounded by the notion that JavaScript code can come from the server in various contexts and the client in numerous contexts, the code a static analysis tool is expected to audit may not even exist. Overall, static analysis tools won't be able to provide any helpful findings.

Several traditional dynamic scanners perform **gray box testing**. These scanners perform usual dynamic testing of the HTTP requests with malicious payloads and download the application's JavaScript files to perform basic static analysis on them. Unfortunately, the static analysis engines used by these tools are relatively weak. For example, Burp Suite includes an engine for static analysis of JavaScript code. In reality, the tool identifies JavaScript sources and sinks via a pattern-based search and performs limited dataflow analysis within one page to find places where tainted data ends up in a sink. This is extremely inefficient because data often travels across pages, and JavaScript files and other pages may access data stored on the client side.

Both white box and gray box testing encounter a few additional difficulties.

1. The task of scanning the JavaScript in a web application is not trivial, because aside from .js files, JavaScript may be executed in numerous other contexts. It can run inline between <script> tags, in event handlers, or in CSS or SVG files or be dynamically loaded from Ajax calls.
2. Applications often include large JavaScript libraries and frameworks, such as jQuery, Backbone.js, AngularJS, and others. These libraries include large obfuscated codebases, use esoteric and dynamic JavaScript functionality, and have their own specific syntax.

At the same time, JavaScript analysis implemented by dynamic tools is very basic and can be described as a simple grep search for sources and sinks. Therefore, the incomplete analysis engines present in free and commercial software lack support for dynamic and edge-case JavaScript syntax, which is heavily used by modern JavaScript frameworks. As a result, JavaScript static analysis engines face the common problem of static analysis—understanding and supporting language frameworks.

Scanning JavaScript code requires a new approach—**interactive application security testing** (IAST). There are two parts of IAST.

1. The tool must be present in the application's runtime through some kind of injection or instrumentation so that it tracks the code as it is being executed.
2. Then there is the interactive part, where the user or tester needs to manually interact with the application, while the dataflows are being tracked.

In addition, there can be an artificial intelligence element, where the tool creates dynamic tests based on the static analysis results. For example, the tool finds a sink through static analysis and then creates test cases sending payloads to this sink.

For most languages, to perform IAST, the tool must be injected into the application runtime—for example, a web application server such as Apache, JBoss, or IIS. However, since JavaScript is executed on the client side within a browser, injection must happen in the browser. The browser's interpreter is the JavaScript's runtime. Therefore, testers need instrumented browsers that are able to track dataflow as the code is being run. One tool that implements this approach is DOMinator. A modified version of Mozilla Firefox is turned into a JavaScript VM to add support for DOMinator's scanning/tracking engine and rulepack. The tool has some dynamic testing capabilities for JavaScript, but it provides better results in "interactive" or "manual" mode, when the tester is actually using the application. This is similar to the "manual explore" functionality in many dynamic scanners.

Conclusion

The widespread use of contemporary web applications, which are heavily dependent on client-side JavaScript, and mobile applications using JavaScript to achieve cross-platform compatibility has led to the emergence of a new type of cross-site scripting. DOM-based XSS doesn't introduce a new way the exposure can occur (reflected or stored) but rather a new location where the payload is constructed. By the case of DOM-based XSS, the exposure happens entirely on the client side. By contrast, in traditional XSS, JavaScript is injected into the response on the server side. Frequently, the DOM-based JavaScript payload does not even travel to the server. This new type of cross-site scripting is as serious as server-side reflected or stored XSS, and in many cases, it can be more severe due to the existing visibility gap and the fact that traditional protection tools like web application firewalls (WAFs) cannot catch it at the network boundary. At the same time, an attacker who gains access to the DOM has the same level of access to user information and web application functionality as with traditional XSS.

Detecting DOM-based XSS is not a trivial task, because traditional DAST and SAST tools and approaches do not yield promising results. DAST tools are ineffective due to the location of the exposure. Since the payload is injected and executed on the client side and does not even travel to the server, the vulnerability cannot be detected by tools sending HTTP requests and analyzing HTTP responses. SAST tools are ineffective due to the complexity and dynamic nature of JavaScript being an interpreted language. Therefore, new testing approaches and tools are needed for detecting DOM-based XSS. One example approach is IAST, where the detection mechanism is integrated into the browser's interpreter. However, new reliable tools implementing IAST are yet to emerge. In the meantime, finding DOM-based vulnerabilities remains largely a manual effort. The question of preventing DOM-based XSS and protecting applications from it remains open.

The Synopsys difference

Synopsys helps development teams build secure, high-quality software, minimizing risks while maximizing speed and productivity. Synopsys, a recognized leader in application security, provides static analysis, software composition analysis, and dynamic analysis solutions that enable teams to quickly find and fix vulnerabilities and defects in proprietary code, open source components, and application behavior. With a combination of industry-leading tools, services, and expertise, only Synopsys helps organizations optimize security and quality in DevSecOps and throughout the software development life cycle.

For more information, go to www.synopsys.com/software.

Synopsys, Inc.
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

Contact us:
U.S. Sales: 800.873.8193
International Sales: +1 415.321.5237
Email: sig-info@synopsys.com

©2020 Synopsys, Inc. All rights reserved. Synopsys is a trademark of Synopsys, Inc. in the United States and other countries. A list of Synopsys trademarks is available at www.synopsys.com/copyright.html. All other names mentioned herein are trademarks or registered trademarks of their respective owners. June 2020