FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications

Prateek Saxena[§] Steve Hanna[§] Pongsin Poosankam[‡§] Dawn Song[§]
{*prateeks,sch,ppoosank,dawnsong*}*@eecs.berkeley.edu*
§University of California, Berkeley
‡Carnegie Mellon University

Abstract

The complexity of the client-side components of web applications has exploded with the increase in popularity of web 2.0 applications. Today, traditional desktop applications, such as document viewers, presentation tools and chat applications are commonly available as online JavaScript applications.

Previous research on web vulnerabilities has primarily concentrated on flaws in the server-side components of web applications. This paper highlights a new class of vulnerabilities, which we term client-side validation (or CSV) vulnerabilities. CSV vulnerabilities arise from unsafe usage of untrusted data in the client-side code of the web application that is typically written in JavaScript. In this paper, we demonstrate that they can result in a broad spectrum of attacks. Our work provides empirical evidence that CSV vulnerabilities are not merely conceptual but are prevalent in today's web applications.

We propose dynamic analysis techniques to systematically discover vulnerabilities of this class. The techniques are light-weight, efficient, and have no false positives. We implement our techniques in a prototype tool called FLAX, which scales to real-world applications and has discovered 11 vulnerabilities in the wild so far.

1 Introduction

Input validation vulnerabilities constitute a majority of web vulnerabilities and have been widely studied in the past [4, 8, 24, 28, 30, 35, 42, 43]. However, previous vulnerability research has focused primarily on the server-side components of web applications. This paper focuses on *client-side validation* (or CSV) vulnerabilities, a new class of vulnerabilities which result from bugs in the client-side code.

A typical Web 2.0 application has two parts: a *server-side* component and a *client-side* component. The server-side component processes the user's request and generates an HTML response that is sent back to the browser. The client-side code of the web application, typically written in JavaScript, is sent with the HTML response from the server. The client-side component executes in the web browser and is responsible for processing input data and dynamically updating the view of web page on the client. We define a CSV vulnerability as one which results from unsafe usage of untrusted data in the client-side code of the web application.

CSV vulnerabilities belong to the general class of input validation vulnerabilities, but are different from traditional web vulnerabilities like SQL injection [10, 35] and reflected/stored cross-site scripting [18, 26, 37, 39]. For example, one type of CSV vulnerability involves data that enters the application through the browser's cross-window communication abstractions and is processed completely by JavaScript code, without ever being sent back to the web server. Another type of CSV vulnerability is one where a web application sanitizes input data sufficiently before embedding it in its initial HTML response, but does not sanitize the data sufficiently for its use in the JavaScript component.

CSV vulnerabilities are becoming increasingly likely due to the growing complexity of JavaScript applications. Increasing demand for interactive performance of rich web 2.0 applications has led to rapid deployment of application logic as client-side scripts. A significant fraction of the data processing in AJAX applications (such as Gmail, Google Docs, and Facebook) is done by JavaScript components. JavaScript has several dynamic features for code evaluation and is highly permissive in allowing code and data to be inter-mixed. As a result, attacks resulting from CSV vulnerabilities often result in compromise of the web application's integrity.

Goals. As a first step towards finding CSV vulnerabilities, we aim to develop techniques that analyzes a web application in an end-to-end manner. Since most existing works have targeted their analyses to server-side components (written in PHP, Java, etc.), this paper develops complementary techniques to discover vulnerabilities in client-side code. In particular, we develop a framework for systematic analysis of JavaScript[1] code. Our objective is to build a tool for vulnerability discovery that does *not* require developer annotations, has no false positives and is usable on real-world applications.

Challenges. The first challenge of holistic application analysis is in dealing with the complexity of JavaScript. Many JavaScript programs use code evaluation constructs to dynamically generate code as well as to serialize strings into complex data structures (such as JSON arrays/objects). In addition, the language supports myriad high-level operations on complex data types, which makes the task of practical analysis difficult.

In JavaScript application code, we observe that parsing operations are syntactically indistinguishable from validation checks. This makes it infeasible for automated syntactic analyses to reason about the sufficiency of validation checks in isolation from the rest of the logic. Due to the convenience of their use in the language, developers tend to treat strings as a universal type for exchange, both of code as well as data. Consequently, complex string operations such as regular expression match and replace are pervasively used both for parsing input and for performing custom validation checks.

Third, in many web applications the client-side code periodically sends data to a remote server for processing via browser interfaces such as `XMLHttpRequest`, and then operates on the returned result. We call such a flow of data, to a server and back, a *reflected flow*. Client-side analyses face the inherent difficulty of dealing with hidden processing on remote servers due to reflected flows.

Existing Approaches. Fuzzing or black-box testing is a popular light-weight mechanism for testing applications. However, black-box fuzzing does not scale well with a large number of inputs and is often inefficient in exploration of the input space. A more directed approach used in the past in the context of server-side code analysis is based on *dynamic taint-tracking*. Dynamic taint analysis is useful for identifying a flow of data from an untrusted source to a critical operation. However, dynamic taint-tracking alone alone can not determine if the application sufficiently validates untrusted data before using it, especially when parsing and validation checks are syntactically indistinguishable. If an analysis tool treats all string operations on the input as parsing constructs, it will fail to identify validation checks and will report false positives even for legitimate uses (as shown by our experiments in Section 5). On the other hand, if the analysis treats any use of untrusted data which has been passed through a parsing/validation construct as safe, it is likely to miss many bugs. Static analysis is another approach [14, 17]; however static analysis tools do not directly provide concrete exploit instances and require additional developer analysis to prune away false positives.

Recently, symbolic execution techniques have been used for discovering and diagnosing vulnerabilities in server-side logic [9, 23, 25, 42]. However, web applications pervasively use complicated operations on string and arrays data types, both of which raise difficulties for decision procedures involved in symbolic execution techniques. The power and expressiveness of string decision procedures today is limited. Practical implementations of string decision procedures presently do not deal with the generality of JavaScript string constraints involving common operations (such as `String.replace`, regular expression match, concatenation and equality) expressed together over multi-variable, variable-length inputs [9, 20, 23, 25]. Other approaches have been limited to a subset of input-transformation operations in PHP [4]. The present limitations of symbolic execution tools motivate the need for designing lighter-weight techniques.

Our Approach. We propose a dynamic analysis approach to discover vulnerabilities in web applications called *taint enhanced blackbox fuzzing*. Our technique is a hybrid approach that combines the features of dynamic taint analysis with those of automated random fuzzing. It remedies the limitations of purely dynamic taint analysis (described above), by using random fuzz testing to generate test cases that concretely demonstrate the presence of a CSV vulnerability. This simple mechanism eliminates false alarms that would result from a purely taint-based tool.

The number of test cases generated by vanilla blackbox fuzzing increases combinatorially with the size of the input. In our hybrid approach, we use character-level precise dynamic taint information to prune the input search space significantly. Dynamic taint information extracts knowledge of the type of sink operation involved in the vulnerability, thereby making the subsequent blackbox fuzzing specialized for each sink type (or in other words, be *sink-aware*). Taint enhanced blackbox fuzzing scales well because the results of dynamic taint analysis are used to create independent abstractions of the original application which are small and take fewer inputs, and can be tested efficiently with sink-aware fuzzing. From our experiments (Section 5), we report an average reduction of 55% in the input sizes with the use of dynamic taint information.

Summary of Results. We implement our techniques into a prototype tool called FLAX. So far, FLAX has discovered 11 CSV vulnerabilities in our preliminary study of 40 popular real-world JavaScript-intensive programs in the wild, which includes several third-party iGoogle gadgets, web

[1] Our JavaScript analysis techniques take a blackbox view of the server-side code currently, though in the future these could be be combined with existing whitebox analyses of server-side components

sites, AJAX applications and third-party libraries. These vulnerabilities were unknown to us prior to the experiments. Our findings confirm that CSV vulnerabilities are not merely conceptual but are prevalent in web applications today. Our experimental results also provide a quantitative measurement of the improvements taint enhanced blackbox fuzzing gains over vanilla dynamic taint analysis or random testing in our application.

Summary of Contributions. This paper makes the following contributions:

1. We introduce client-side validation vulnerabilities, a new class of bugs which result from unvalidated usage of untrusted data in JavaScript code. We provide empirical evidence of these vulnerabilities in real-world applications.

2. We build a framework to systematically discover CSV vulnerabilities called FLAX, which has found 11 previously unknown CSV bugs. Internally, FLAX simplifies JavaScript semantics to an intermediate language that has a simple type system and a small number of operations. This enables dynamic analyses employed in FLAX to be implemented in a robust and scalable way. Additionally, FLAX is designed to analyze applications with reflected flows without the need for a server analysis component.

3. FLAX employs *taint enhanced blackbox fuzzing* : a hybrid, dynamic analysis approach which combines the benefits of dynamic taint analysis and random fuzzing. This technique is light-weight as compared to symbolic execution techniques, has no false positives and is scalable enough to use on real-world applications.

2 Problem Definition

In this section, we outline our threat model, give examples of CSV vulnerabilities and conceptualize them as a class, and define the problem of finding CSV vulnerabilities.

2.1 Threat Model and Problem Definition

We define a CSV vulnerability as a programming bug which results from using untrusted data in a critical *sink* operation without sufficient validation. A critical *sink* is a point in the client-side code where data is used with special privilege, such as in a code evaluation construct, or as an application-specific command to a backend logic or as cookie data.

In our analysis, any data which is controlled by an external web principal is treated as untrusted. Additionally, user data (such as from form fields or text areas) is treated as untrusted as well. Untrusted data could enter the client-side code of a web application in three ways. First, data from an untrusted web attacker could be reflected in the honest web server's HTML response and subsequently read for processing by the client side code. Second, untrusted data from other web sites could be injected via the browser's cross-window communication interfaces. These interfaces include HTML 5's `postMessage`, URL fragment identifiers, and window/frame cross-domain properties. Finally, user data fed in through form fields and text areas is also marked as untrusted.

The first two untrusted sources are concerned with the threat model where the attacker is a remote entity that has knowledge of a CSV vulnerability in an honest (but buggy) web application. The attacker's goal is to remotely exploit a CSV vulnerability to execute arbitrary code, to poison cookie data (possibly inject session identifiers), or to issue web application-specific commands on behalf of the user. The attack typically only involves enticing the user into clicking a link of the attacker's choice (such as in a reflected XSS attack).

We also consider the "user-as-an-attacker" threat model where the user data is treated as untrusted. In general, user data should not be interpreted as web application code. For instance, if user can inject scripts into the application, such a bug can be used in conjunction with other vulnerabilities (such as a login-CSRF vulnerabilities) in which the victim user is logged-in as the attacker while the application behavior is under attacker's control [6]. In our view, FLAX should make developers aware of the existence of errors in this threat model, even though the severity of resulting exploits is usually limited and varies significantly from application to application.

This paper addresses the problem of finding CSV vulnerabilities in the target web application by generating concrete witness inputs. The problem of vulnerability discovery has two orthogonal challenges — exploring the entire functionality of the program, and finding an input that exposes a vulnerability in some explored functionality. In this paper, we focus solely on the second challenge, assuming that our analysis would be driven by an external test harness that explores the large space of the application's functionality. Specifically, the input to our analysis is a web application and an initial benign input. Our analysis aims to find an exploit instance by systematically searching the equivalence class of inputs that force the program execution down the same path as the given benign input.

Running Example. For ease of explanation and concreteness, we introduce a running example of a hypothetical AJAX chat application. The example application consists of two windows. The main window, shown in Figure 1, asynchronously fetches chat messages from the backend

```
1: var chatURL = "http://www.example.com/";
2: chatURL += "chat_child.html";
3: var popup = window.open(chatURL);
4: ...
5: function sendChatData (msg) {
6:   var StrData = "{\"username\": \"joe\", \"message\": \"" + msg + "\"}";
7:   popup.postMessage(StrData, chatURL);
}
```

Figure 1. An example of a chat application's JavaScript code for the main window, which fetches messages from the backend server at `http://example.com/`

server. Another window receives these messages from the main window and displays them, the code for which is shown in Figure 2. The communication between the two windows is layered on `postMessage`[2], which is a string-based message passing mechanism proposed for inclusion in HTML 5. The application code in the display window has two sources of untrusted data — the data received via `postMessage` that could be sent by any browser window, and the `event.origin` property, which is the origin (port, protocol and domain) of the sender.

2.2 Attacks resulting from CSV Vulnerabilities

While some of the vulnerabilities described below have been discussed in previous research literature by leveraging other web vulnerabilities, in this section we show that they can result from CSV vulnerabilities as well.

Origin Mis-attribution. Certain cross-domain communication primitives such as `postMessage` are designed to facilitate sender authentication. Applications using `postMessage` are responsible for validating the authenticity of the domain sending the message. The example in Figure 2 illustrates such an attack on line 13. The vulnerability arises because the application checks the domain field of the origin parameter insufficiently, though the protocol sub-field is correctly validated. The failed check allows any domain name containing "example", including an attacker's domain hosted at "evilexample.com", to send messages. As a result, the vulnerable code naively trusts the received data even though the data is controlled by an untrusted principal. In the running example, for instance, an untrusted attacker can send chat messages to victim users on behalf of benign users.

Code injection. Code injection is possible because JavaScript can dynamically evaluate both HTML and script code using various DOM methods (such as `document.write`) as well as JavaScript native constructs (such as `eval`). This class of attacks is commonly referred to as DOM-based XSS [27,29]. An example of this attack is shown in Figure 2 on line 19. In the example, the display child window uses `eval` to serialize the input string from a JSON format, without validating for its expected structure. Such attacks are prevalent today because popular data exchange interfaces, such as JSON, were specifically designed for use with the `eval` constructs. In Section 5, we outline additional phishing attacks in iGoogle gadgets layered on such XSS vulnerabilities, to illustrate that a wide range of nefarious goals can be achieved once the application integrity is compromised.

Command injection. Many AJAX applications use untrusted data to construct URL parameters dynamically, which are then used to direct `XMLHttpRequest` requests to a backend server. Several of these URL parameters function as application-specific commands. For instance, the chat application in the example sends a confirmation command to a backend script on lines 29-31. The backend server script may take other application commands (such as adding friends, creating a chat room, and deleting history) similarly from HTTP URL parameters. If the HTTP request URL is dynamically constructed by the application in JavaScript code (as done on line 30) using untrusted data without validation, the attacker could inject new application commands by inserting extra URL parameters. Since the victim user is already authenticated, command injection allows the attacker to perform unintended actions on behalf of the user. For instance, the attacker could send '*hi & call=addfriend&name=evil*' as the message which could result in adding the attacker to the buddy list of the victim user.

Cookie-sink vulnerabilities. Web applications often use cookies to store session data, user's history and preferences. These cookies may be updated and used in the client-side code. If an attacker can control the value written to a cookie by exploiting a CSV vulnerability, she may fix the values of the session identifiers (which may result in a session fixation attack) or corrupt the user's preferences and history data.

[2] In the `postMessage` interface design, the browser is responsible for attributing each message with the domain, port, and protocol of the sender principal and making it available as the "origin" string property of the message event [7,40]

```
1:function ParseOriginURL (url) {
2: var re=/(.*?):\/\/(.*?)\.com/;
3: var matches = re.exec(url);
4: return matches;
5:}
6:
7:function ValidateOriginURL (matches)
8:{
9: if(!matches) return false;
10: if(!/https?/.test(matches[1]))
11:        return false;
12: var checkDomRegExp = /example/;
13: if(!checkDomRegExp.test (matches[2])) {
14:        return false;  }
15: return true;     // All Checks Ok
16:}
```

```
17:// Parse JSON into an array object
18:function ParseData (DataStr) {
19:  eval (DataStr);
20:}
21:function receiveMessage(event) {
22: var O = ParseOriginURL(event.origin);
23:  if (ValidateOriginURL (O)) {
24:   var DataStr = 'var new_msg =(' +
25:              event.data + ');';
26:   ParseData(DataStr);
27:   display_message(new_msg);
29:   var backserv = new XMLHttpRequest(); ...;
30:   backserv.open("GET","http://example.com/srv.php?
      call=confirmrcv&msg="+new_msg["message"]);
31:   backserv.send();} ... } ...
32: window.addEventListener("message",
                receiveMessage,...);
```

Figure 2. An example vulnerable chat application's JavaScript code for a child message display window, which takes chat messages from the main window via `postMessage`. The vulnerable child message window code processes the received message in four steps, as shown in the `receiveMessage` function. First, it parses the principal domain of the message sender. Next, it tries to check if the origin's port and domain are "http" or "https" and "example.com" respectively. If the checks succeed, the popup parses the JSON [3] string data into an array object and finally, invokes a function for displaying received messages. In lines 29-31, the child window sends confirmation of the message reception to a backend server script.

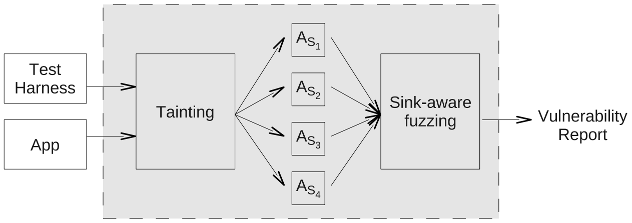


Figure 3. Approach Overview

3 Approach

In this section, we present the key design points of our approach and explain our rationale for employing a hybrid dynamic analysis technique into FLAX.

3.1 Approach and Architectural Overview

Figure 3 gives a high-level view of our approach – the boxed, shaded part represents the primary technical contribution of this work. The input to our analysis is an initial benign input and the target application itself. The technique explores the equivalence class of inputs that execute the same program path as the initial benign input and finds a flow of untrusted data into a critical sink without sufficient validation.

Approach. In the first step, we execute the application with the initial input $\mathcal{I}$ and perform character-level dynamic taint analysis. Dynamic taint analysis identifies all uses of untrusted data in critical sinks. This analysis identifies two pieces of information about each potentially dangerous data flow: the type of critical sink, and, the fractional part of the input that is influences the data used in the critical sink. Specifically, we extract the range of input characters $\mathcal{I}_{\mathcal{S}}$ that on which data arguments of a sink operation $\mathcal{S}$ are directly dependent. All statements that operate on data that is directly dependent on $\mathcal{I}_{\mathcal{S}}$, including path conditions, are extracted into an executable slice of the original application which we term as an *acceptor slice* (denoted as $\mathcal{A}_{\mathcal{S}}$). $\mathcal{A}_{\mathcal{S}}$ is termed so because it is a stand-alone program that accepts all inputs in the equivalence class of $\mathcal{I}$, in the sense that they execute the same program path as $\mathcal{I}$ up to the sink point $\mathcal{S}$. As the second step, we fuzz each $\mathcal{A}_{\mathcal{S}}$ to find an input that exploits a bug. Our fuzzing is sink-aware because it uses the details of the sink node exposed by the taint analysis step. Fuzz testing on $\mathcal{A}_{\mathcal{S}}$ semantically simulates fuzzing on the original application program. Using an acceptor slice to link the two high-level steps has two advantages:

- *Program size reduction.* $\mathcal{A}_{\mathcal{S}}$ can be executed as a program on its own, but is significantly smaller in size than the original application. From our experiments in Section 5, $\mathcal{A}_{\mathcal{S}}$ is typically smaller than the executed instruction sequence by a factor of 1000. Thus, fuzzing on a concise acceptor slice instead of the original complex application is a practical improvement. It avoids application restart, decouples the two high-level steps, and allows testing of multiple sinks to proceed in parallel.

- *Fuzzing search space reduction.* Sink-aware fuzzing focuses only on $\mathcal{I}_S$ for each $\mathcal{A}_S$, rather than the entire input. Additionally, our sink-aware fuzzer has custom rules for each type of critical sink because each sink results in different kinds of attacks and requires a different attack vector. As an example, it distinguishes `eval` sinks(which allow injection of JavaScript code) from DOM sinks(which allow HTML injection). Our sink-aware fuzzing employs input mutation strategies that are based on grammars such as the HTML syntax, JavaScript syntax, or URL syntax grammars.

3.2 Technical Challenges and Design Points

One of our contributions is to design a framework that simplifies JavaScript analysis and explicitly models reflected flows and path constraints. We explain each of these design points in detail below.

Modeling Path Constraints. The running example in Figure 2 shows how validation checks manifest as conditional checks, affecting the choice of execution path in the program. Saner, an example of previous work that precisely analyzes server-side code, has considered only input-transformation functions as sanitization operations in its dynamic analysis, thereby ignoring branch conditions [4]. Our techniques improve on Saner's by explicitly modelling path constraints, thereby enabling FLAX to capture the validation checks as branch conditions, as shown in the running example in the $\mathcal{A}_S$.

Simplifying JavaScript. There are two key problems in designing analyses for JavaScript code.

- *Rich data types and complex operations*. JavaScript supports complex data types such as string and array, with a variety of native operations on them. The ECMA-262 specification defines over 50 operations on string and array data types alone [1]. JavaScript analysis becomes complex because there are several syntactic constructs that can perform the same semantic operations. As a simple indicative example, there are several ways to split a string on a given separator (such as by using String.split, String.match, String.indexOf, and String.substring).

  In our approach, we canonicalize JavaScript operations and data references into a simplified intermediate form amenable for analysis, which we call *JASIL* (JAvascript Simplified Instruction Language). JASIL has a simpler type system and a smaller set of instructions which are sufficient to faithfully express the semantics of higher-level operations relevant to the applications we study. As a result, JASIL serves as a robust platform for simplified implementation of dynamic taint analysis and other analyses.

- *Aliasing*. There are numerous ways in which two different syntactic expressions can refer to the same object at runtime. This arises because of the dynamic features of JavaScript, such as reflection, prototype-based inheritance, complex scoping rules, function overloading, as well as due to numerous exposed interfaces to access DOM elements. Reasoning about such a diverse set of syntactic variations is difficult. Previous static analysis techniques applied to this problem area required complex points-to analyses [14, 17].

  This forms one of the main motivations for designing FLAX as a dynamic analysis tool. FLAX dynamically translates JavaScript operations to JASIL, and by design each operand (an object, variable or data element) in JASIL is identified by its allocated storage address. With appropriate instrumentation of the JavaScript interpreter, we identify element accesses regardless of the syntactic complexity of the access pattern used in the references.

Dealing with reflected flows. In this paper, we consider data flows of two kinds: *direct* and *reflected*. A direct flow is one where there is a direct data dependency between a source operation and a critical sink operation in script code. Dynamic taint analysis identifies such flows as potentially dangerous. A reflected flow occurs when data is sent by the JavaScript application to a backend server for processing and the returned results are used in further computation on the client. Our dynamic taint analysis identifies untrusted data propagation across a reflected flow using a common-substring based content matching algorithm[3]. During a reflected flow, data could be transformed on the server. The exact data transformation/sanitization on the server is hidden from the client-side analysis. To address this, we compositionally test the client-side code in two steps. First, we test the client-side code independently of the server-side code by generating candidate inputs that make simple assumptions about the transformations occurring in reflected flows. Subsequently, it verifies the assumption by running the candidate attack concretely, and reports a vulnerability if the concrete test succeeds.

4 Design and Implementation

We describe our algorithm for detecting vulnerabilities and present details about the implementation of our prototype tool FLAX.

[3] It is possible to combine client-side taint tracking with taint tracking on the server; however, in present work we take a blackbox view of the web server.

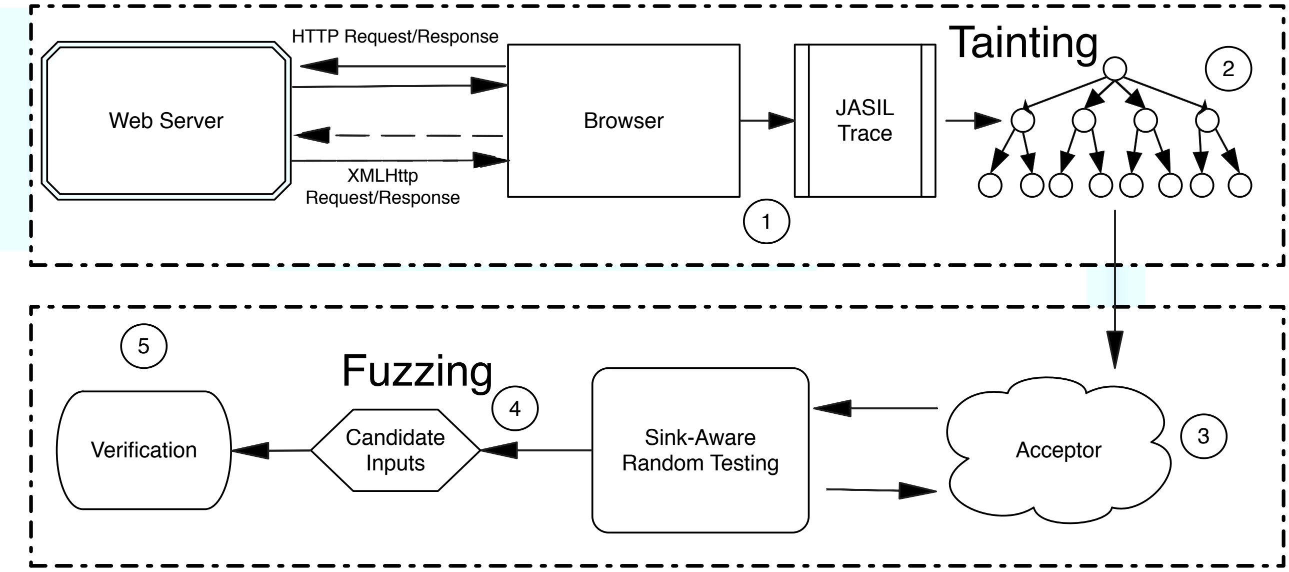


Figure 4. System Architecture for FLAX

4.1 Algorithm

Figure 4 shows the architectural overview of our taint enhanced blackbox fuzzing algorithm. The pseudocode of the algorithm is described in Figure 5. At a high level, it consists of 5 steps:

1. *Dynamic trace generation and conversion to JASIL.* Run the application concretely in our instrumented web browser to record an execution trace in JASIL form.

2. *Dynamic taint analysis.* Perform dynamic taint analysis on the JASIL trace to identify uses of external data in critical sinks. For each such potentially dangerous data flow into a sink $\mathcal{S}$, our analysis computes the part of the untrusted input ($\mathcal{I}_{\mathcal{S}}$) which flows into the critical sink.

3. *Generate an acceptor slice.* For each sink $\mathcal{S}$ and the given associated information about $\mathcal{S}$ from the previous step, the analysis extracts an executable slice, $\mathcal{A}_{\mathcal{S}}$, as defined in Section 3.1.

4. *Sink-aware Random testing.* Apply random fuzzing to check if sufficient validation has been performed along the path to a given sink operation. For a given $\mathcal{A}_{\mathcal{S}}$, our fuzzer generates random inputs according to sink-specific rules and custom attack vectors.

5. *Verification of candidate inputs.* Randomized testing of $\mathcal{A}_{\mathcal{S}}$ generates candidate vulnerability inputs assuming a model of the transformation operations on the

```
Input: $\mathcal{T}$ : Trace
Output: $\mathcal{V}$ : AttackString List
type $Flow$ : {
            var Sink, Source : Int List,
            var TaintedInsList : Int List,
            var InputOffset : (Int,Int) List
            };
var $FlowList$ : Flow List;
$FlowList = CalculateFlows\ (\mathcal{T})$;
var $Candidates$ = InputString List;
var $\mathcal{V}$ = InputString List;
foreach $\mathcal{F}$ in $FlowList$ do
    $\mathcal{A}_{\mathcal{S}} = GenAutomaton(\mathcal{F}, \mathcal{T})$;
    $Candidates = Fuzz\ (\ \mathcal{A}_{\mathcal{S}}$
    $, max\_length, max\_iters)$;
    foreach $\mathcal{C}$ in $Candidates$ do
        $\mathcal{C}_{\mathcal{T}} = ExecuteOnInput(\mathcal{C})$
        var $Result = VerifyAttack(\mathcal{T}, \mathcal{C}_{\mathcal{T}})$
        if Result then
            $\mathcal{V}.append([\mathcal{F}, \mathcal{C}_{\mathcal{T}}.input])$;
        end
    end
end
return $\mathcal{V}$;
```

Figure 5. Algorithm for FLAX

x : τ ::=	v : τ	(Assignment, Type Conversion)
x : τ ::=	$*$ (v : $Ref(\tau)$)	(Dereference)
x : Int ::=	v1 : Int op v1 : Int	(Arithmetic)
x : $Bool$ ::=	v1 : τ op v1 : τ	(Relational)
x : $Bool$ ::=	v1 : $Bool$ op v1 : $Bool$	(Logical)
x : PC ::=	if (testvar : $Bool$) then (c : Int) else (c : Int)	(Control Flow)
x : $String$ ::=	substring(s : $String$, startpos : Int, len : Int)	(String Ops)
x : $String$ ::=	concat(s1 : $String$, s2 : $String$)	(String Ops)
x : $String$ ::=	fromArray(s1 : $Ref(\tau)$)	(String Ops)
x : $String$ ::=	convert(s1 : $String$)	(String Ops)
x : $Char * \kappa$ ::=	convert(i : Int)	(Character Ops)
x : Int ::=	convert(i : $Char * \kappa$)	(Character Ops)
x : τ ::=	$\mathcal{F}$ (i_1 : τ, . . ., i_n : τ)	(Uninterpreted Function Call)

Figure 6. Simplified operations supported in JASIL intermediate representation

$\tau :=$	$\eta \mid \beta[\eta] \mid Bool \mid Null \mid Undef \mid PC$
$\eta :=$	$Int \mid \beta$
$\beta :=$	$Ref(\tau) \mid String(\kappa) \mid Char(\kappa)$
$\kappa :=$	$UTF8 \mid UTF7 \mid \ldots$

Figure 7. Type system of JASIL intermediate representation

server that may occur in reflected flow. This final step verifies that the assumptions hold, by testing the attacks concretely on the web application and checking that the attack succeeds by using a browser-based oracle.

4.2 JASIL

To simplify further analysis, we lower the semantics of the JavaScript language to a simplified intermediate representation which we call JASIL. JASIL is designed to have a simple type system with a minimal number of operations on the defined data types. A brief summary of its type system and categories of operations are outlined in Figure 7 and Figure 6 respectively. JavaScript interpreters already perform some amount of semantic lowering in converting to internal bytecode. However, the semantics of typical JavaScript bytecode are not substantially simpler, because most of the complexity is hidden in the implementation of the rich native operations that the interpreter's runtime supports.

JASIL has a substantially smaller set of operations, shown in Figure 6. In our design, we have found JASIL to be sufficient to express the operational semantics of a subset of JavaScript commonly used in real applications. Our design is implemented using WebKit's JavaScript interpreter, the core of the Safari web browser, and is faithful to the semantics of the operations as implemented therein. In our work, we lower all the native string operations, array operations, integer operations, regular expression based operations, global object functions, DOM functions, and the operations on native window objects. Lowering to JASIL simplifies analyses. For instance, consider a `String.replace` operation in JavaScript. Intuitively, a replace operation retains some parts of its input string in its output while transforming the other parts with specified strings. An execution of the replace operation can be replaced by a series of substring operations followed by a final concatenation of substrings. With JASIL, subsequent dynamic taint analysis is greatly simplified because the tainting engine only needs to reason about simple operations like substring extraction and concatenation.

In addition to lowering semantics of complex operations, JASIL explicitly models procedure call/return semantics, parameter evaluation, parameter passing, and object creation and destruction. Property look-ups on JavaScript objects and accesses to native objects such as the DOM or window objects are converted to operations on a functional map in JASIL (denoted by $\beta[\eta]$ in its type system). This canonicalization of references makes further analysis easier.

In JASIL, each object, variable or data element is identified by its allocated storage address, which obviates the need to reason about most forms of aliasing. As one example of how this simplification allows robust reasoning, consider the case of prototype-based inheritance in JavaScript. In JavaScript, whenever an object $\mathcal{O}$ is created, the object inherits all the properties of a *prototype object* corresponding to the constructor function, accessible through the `.prototype` property of the function (functions are first-class types in JavaScript and behave like normal objects). The prototype object of the constructor function could in turn inherit from other prototype objects depending on how they are created. When a reference $\mathcal{O}$`.f` is resolved, the field

Sources
document.URL document.URLUnencoded document.location.* document.referrer.* window.location.* event.data event.origin textbox.value forms.value

Critical Flow Sinks	Resulting Exploit
eval(), window.execScript(), window.setInterval(), window.setTimeout()	Script injection
document.write(...), document.writeln(...), document.body.innerHtml, document.cookie document.forms[0].action, document.create(), document.execCommand(), document.body.*, window.attachEvent(), document.attachEvent()	HTML code injection
document.cookie	Session fixation attacks
XMLHttpRequest.open(,url,), document.forms[*].action,	Command Injection and parameter injection

Figure 8. (Left) Sources of untrusted data. (Right) Critical sinks and corresponding exploits that may result if untrusted data is used without proper validation.

`f` is first looked up in the object $\mathcal{O}$. If it is not found, it is looked up in the prototype object of $\mathcal{O}$ and in the subsequent objects of the prototype chain. Thus, determining which object is referenced by $\mathcal{O}$ statically requires a complex alias analysis. In simplifying to JASIL, we instrumented the interpreter to record the address identifier for each variable used after the reference resolution process (including the scope and prototype chain traversals) is completed. Therefore, further analysis does not need any further reasoning about prototypes or scopes.

To collect a JASIL trace of a web application for analysis we instrumented the browser's JavaScript interpreter to translate the bytecode executed at runtime to JASIL. This required extensive instrumentation of the JavaScript interpreter, bytecode compiler and runtime, resulting in a patch of 6032 lines of C++ code to the vanilla WebKit browser. To facilitate recovering JavaScript source form from the JASIL representation, auxiliary information mapping the dynamic allocation addresses to native object types is embedded as metadata in the JASIL trace.

4.3 Dynamic taint analysis

Character-level precise modeling of string operation semantics. JavaScript applications are array- and string- centric; lowering of JavaScript to JASIL is a key factor in reasoning about complex string operations in our target applications. Dynamic taint analysis has been used with success in several security applications outside of the realm of JavaScript applications [31, 32, 43]. For JavaScript, Vogt *et al.* have previously developed taint-tracking techniques to detect confidentiality attacks resulting from cross-site scripting vulnerabilities [39]. In contrast to their work, our techniques model the semantics of string operations and are character-level precise.

We list the taint sources and sinks used by default in FLAX in Figure 8. FLAX models only direct data dependencies for this step; additional control dependencies for path conditions are introduced during $\mathcal{A}_{\mathcal{S}}$ construction. It performs taint-tracking offline on the JASIL execution trace, which reduces the intrusiveness of the instrumentation by not requiring transformation of the interpreter's core semantics to support taint-tracking. In our experience, this has resulted in a more robust implementation than our previous work on online taint-tracking [29]. Taint propagation rules are straight-forward — assignment and arithmetic operations taint the destination operand if one of the input operands is tainted, while preserving character-level precision. The JASIL string `concatenation` and `substring` operations result in a merge and slicing operation over the ranges of tainted data in the input operands, respectively. The `convert` operation, which implements character-to-integer and integer-to-character conversion, typically results from simplifying JavaScript encode/decode operations (such as `decodeURI`). Taint propagation rules for `convert` are similar: the output is tainted if the input is tainted. Other native functions that are not explicitly modeled are treated as uninterpreted transfer functions, acting merely to transfer taint from input parameters to output parameters in a conservative way.

Tracking data in reflected flow. During this analysis data may be sent to a backend server via the `XMLHttpRequest` object. We approximate taint propagation across such network data flows by using an exact substring match algorithm, which is a simplified form of black-box taint inference techniques proposed in the previous literature [33, 34]. We record all tainted data sent in a reflected flow, and perform a longest common substring

```
function acceptor (input) {
  var path_constraints = true;
  var re = /(.*?):\/\/(.*?)\.com/;
  var matched = re.exec(input);
  if (matched == null) {
  path_constraints = path_constraints & false;
  }
  if (!path_constraints) return false;
  var domain = matched[2];
  var valid = /example/.test(domain);
  path_constraints = path_constraints & valid;
  if (!path_constraints) return false;
  var port = matched[1];
  valid = /https?/.test(port);
  path_constraints = path_constraints & valid;
  if (!path_constraints) return false;
  return true;
}
```

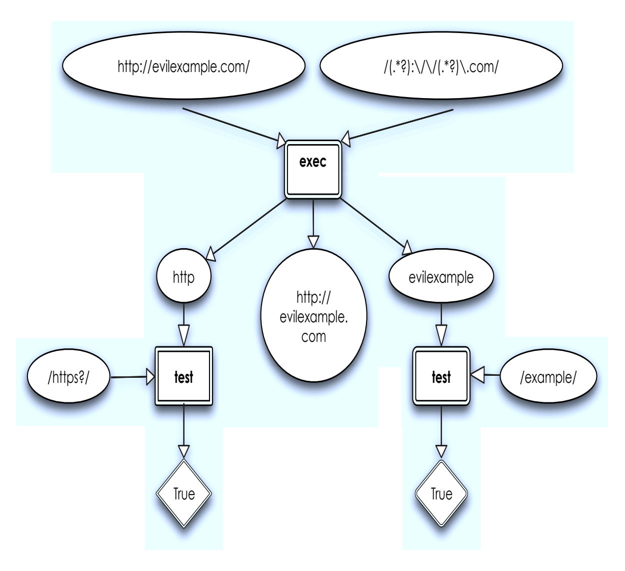


Figure 9. (Left) Acceptor Slice showing validation and parsing operations on `event.origin` field in the running example. (Right) Execution of the Acceptor Slice on a candidate attack input, namely `http://evilexample.com/`

match on the data returned. Any matches that are above a threshold length are marked as tainted, and the associated taint metadata is propagated to the reflected data. This technique has proved sufficient for the AJAX applications in our experiments.

Implicit Sinks. Certain source operations do not have explicit sink operations. For instance, in our running example (Figure 2) the `event.origin` field has no explicit sink. However, this field must be sanitized before any use of `event.data`. We model this case of implicit dependence between two fields by introducing an *implicit sink* node for `event.origin` at any use of `event.data` in critical sink operation. This has the effect that for any use of `event.data`, the path constraint checks on `event.origin` are implicitly included in the acceptor slice.

4.4 Acceptor Slice Construction

After dynamic taint analysis identifies a sink point, FLAX extracts a dynamic executable slice from the program, by walking backwards from the critical sink to the source of untrusted data. In order to fuzz the slice, the JASIL slice is converted back to a stand-alone JavaScript function. This results in an executable function that retains the operations on $\mathcal{I}_\mathcal{S}$, and returns `true` for any input that executes the same path as the original run. The slicing operation captures (a) *data dependencies*, i.e., all operations directly processing $\mathcal{I}_\mathcal{S}$ and (b) a limited form of control dependencies, i.e., all path constraints, conditions of which are directly data dependent on $\mathcal{I}_\mathcal{S}$. Path constraints are conditional checks corresponding to each branch point which force the execution to take the same path as $\mathcal{I}_\mathcal{S}$. Data values which are not directly data dependent (marked tainted) in the original execution, are replaced with their concrete constant values observed during the program execution.

Acceptor Slice for the Running Example. The instructions operating on the `event.origin` in the running example that influences the implicit `eval` sink is shown in Figure 9. It shows the $\mathcal{A}_\mathcal{S}$ for the the `event.origin` field of our example, after certain optimizations, like dead-code elimination. This program models all the validation checks performed on that field, until its use in the implicit sink node at `eval`.

4.5 Sink-aware fuzzing

This step in our analysis performs randomized testing on each $\mathcal{A}_\mathcal{S}$. Note that each critical sink operation can result in a different kind of vulnerability. Therefore, it is useful to target each sink node ($\mathcal{S}$) with a set of specialized attack vectors. For instance, an unchecked flow that writes to the `innerHTML` property of a DOM element can result in HTML code injection and our fuzzer attempts to inject an HTML tag into such a sink. For `eval` sink, our testing targets the injection of JavaScript code. We incorporate a large corpus of publicly available attack vectors for XSS [19] in our fuzzing.

While testing for an attack input that causes $\mathcal{A}_S$ to return true, our fuzzer utilizes the aforementioned attack vectors and a grammar-aware strategy. Starting with the initial benign input, the fuzzer employs a mutation-based strategy to transform, prepend and appends language nonterminals. For each choice, the fuzzer first selects terminal characters based on the knowledge of surrounding text (such as HTML tags, JavaScript nonterminals) and finally resorts to random characters if the grammar-aware strategy fails to find a vulnerability.

To check if a candidate attack input succeeds we use a browser-based oracle. Each candidate input is executed in $\mathcal{A}_S$ and the test oracle determines if the specific attack vector is evaluated or not. If executed, the attack is verified as being a concrete attack instance. For instance, in our running example, the `event.origin` acceptor slice returns true for any URL principal which is not a subdomain of `http://example.com`[4]. Our fuzzer tries string mutations of the original domain `http://example.com` and quickly discovers that there are other domains that circumvent the validation checks.

5 Evaluation

Our primary objective is to determine if taint enhanced blackbox fuzzing is scalable enough to be used on real-world applications to discover vulnerabilities. As a second objective, we aim to quantitatively measure the benefits of taint enhanced blackbox fuzzing over vanilla taint-tracking and purely random testing. In our experiments, FLAX discovers 11 previously unknown vulnerabilities in real applications and our results show that our design of taint enhanced blackbox fuzzing offers significant practical gains over vanilla taint-tracking and fuzzing. We also investigate the security implications of the vulnerabilities by constructing proof-of-concept exploits and we discuss their varying severity in this section.

5.1 Test Subjects

We selected a set of 40 web applications consisting of iGoogle gadgets and other AJAX applications for our experiments. Of these, FLAX observed untrusted data flows into critical sinks for only 18 of the cases, consisting of 13 iGoogle gadgets and 5 web applications. We report detailed results for only these 18 applications in Table 1. We tested each subject application manually to explore its functionality, giving benign inputs to seed our automated testing. For instance, all of the iGoogle gadgets were tested by visiting the benign URL used by the iGoogle web page to embed the gadget in its page. To explore each application's functionality, we entered data into text boxes, clicked buttons and hyperlinks, simulating the behavior of a normal user.

Google gadgets constitute the largest fraction of our study because they are representative of third-party applications popular among internet users today. Most gadgets are reported to have thousands of users with one of the vulnerable gadgets having over 1,350,000 users, as per the data available from the iGoogle gadget directory on December 17^{th} 2009 [2]. The other AJAX applications consist of social networking sites, chat applications and utility libraries which are examples of the trend towards increasing code sharing via third-party libraries. All tests were performed using our FLAX framework running on a Ubuntu 8.04 platform with a 2.2 GHz, 32-bit Intel dual-core processor and 2 GB of RAM.

5.2 Experimental Results

FLAX found several distinct taint sinks in the applications, only a small fraction of which are deemed vulnerable by the tool. Column 2 and 3 of Table 1 reports the number of distinct sinks and number of vulnerabilities found by FLAX respectively. The use of character-level precise taint tracking in FLAX prunes a significant fraction of the input in several cases for further testing. To quantitatively measure this saving we observe the average sizes of the original input and the reduced input size in the acceptor slices (used for subsequent fuzzing), which is reported in columns 4 and 5 of Table 1 respectively. We measure the reduction in the acceptor size, which results in substantial practical efficiencies in subsequent black-box fuzzing. We find that the acceptor slices are small enough to often enable manual analysis for a human analyst. Columns 6 and 7 report the size of dynamic execution trace and the average size of the acceptor slices respectively[5]. The last two columns in Table 1 show the number of test cases it takes to find the first vulnerability in each application and the kinds of vulnerability found.

5.2.1 Prevalence of CSV vulnerabilities

Of the 18 applications in which FLAX observed a dangerous flow, it found a total of 11 vulnerabilities which we report in the third column of Table 1. The vulnerabilities are evidence of a broad range of attack possibilities, as conceptualized in Section 2, though code injection vulnerabilities were the highest majority. FLAX reported 8 code injection vulnerabilities, 1 origin mis-attribution vulnerability, 1 cookie-sink vulnerability and 1 application command

[4] Recall that the running example acceptor does not have an explicit sink, therefore only return true on success and false otherwise.

[5] In our implementation, the acceptor slices are converted back to JavaScript form for further analysis: the size of acceptor slices increases as a result of this conversion by a factor of 4 at most in our implementation, as compared to the numbers reported in column 7

Name	# of Taint Sinks	**Verified Vuln.**	Size of Total Inputs	Size of Acceptor Inputs	Trace Size (# of insns)	Avg. size of $\mathcal{A}_{\mathcal{S}}$	# of Tests to Find 1st Vuln.	Vulnerability Type
Plaxo	178	**0**	119	60	557,442	36	-	-
Academia	1	**1**	334	21	156,621	286	16	Origin Mis-attribution
Facebook Chat	44	**0**	127	127	6,460,591	1,151	-	-
ParseURI	1	**1**	78	62	55,179	638	6	Code injection
AjaxIM	20	**2**	28	28	223,504	517	93	Code injection , Application Command Injection
AskAWord	3	**1**	26	26	59,480	611	93	Cookie Sink
Block Notes	1	**1**	474	96	11,539	766	28	Code injection
Birthday Reminder	6	**0**	632	246	2,178,927	664	-	-
Calorie Watcher	3	**0**	681	20	449,214	733	-	-
Expenses Manager	6	**0**	1,137	65	522,788	1,454	-	-
MyListy	1	**1**	578	47	17,054	1,468	4	Code injection
Notes LP	5	**0**	740	30	144,829	3,327	-	-
Progress Bar	151	**0**	496	264	118,108	475	-	-
Simple Calculator	1	**1**	27	27	72,475	4	93	Code injection
Todo List	1	**0**	632	40	647,849	1,181	-	-
TVGuide	2	**1**	586	66	24,144,843	188	8,366	Code injection
Word Monkey	1	**1**	26	26	237,837	99	93	Code injection
Zip Code Gas	5	**1**	412	69	410,951	248	2	Code injection

Table 1. Applications for which FLAX observed untrusted data flow into critical sinks. The top 5 subject applications are websites and the rest are iGoogle gadgets.

injection vulnerability. We confirmed that all vulnerabilities reported were true positives by manually inspecting the JavaScript code and concretely evaluating them with exploit inputs. The severity of the vulnerabilities varied by application and source of untrusted input, which we discuss in section 5.2.3.

5.2.2 Effectiveness

We quantitatively measure the benefits of taint enhanced blackbox fuzzing over vanilla taint-tracking and random fuzzing from our experimental results.

False Positives Comparison. The second column in Table 1 shows the number of distinct flows of untrusted data into critical sink operations observed; only a fraction of these are true positives. Each of these distinct flows is an instance where a conservative taint-based tool would report a vulnerability. In contrast, the subsequent step of sink-aware fuzzing in FLAX eliminates the spurious alarms, and a vulnerability is reported (column 3 of Table 1) only when a witness input is found. It should be noted that FLAX can have false negatives and could have missed bugs, but completeness is not an objective for FLAX.

We manually analyzed the taint sinks reported as safe by FLAX and, to the best of our ability, found them to be true negatives. For instance, we determined that most of the sinks reported for the Plaxo case were due to code which output the length of the untrusted input to the DOM, which executed repeatedly each time the user typed a character in the text box. Many of the true negatives we manually analyzed employed sufficient validation – for instance, Facebook Chat application correctly validates the origin property of every `postMessage` event it received in the execution. Several other applications validate the structure of the input before using it in a JavaScript `eval` statement or strip dangerous characters before using it in HTML code evaluation sinks.

Efficiency of sink-aware fuzzing. Table 1 (column 8) shows the number of test cases FLAX generated before it found the vulnerability for the cases it deems unsafe. Part of the reason for the small number of cases on average, is that our fuzzing leverages knowledge of the sink operations. Column 4 of the Table 1 shows that the size of the original inputs for most applications is in the range of 100-1000 characters. Slicing on the tainted data prunes away a significant portion of the input space, as seen from column 5 of Table 1. We report an average reduction of 55% from the original input size to the size of test input used in acceptor slices.

Further, the average size of an acceptor slice (reported in column 7 of Table 1) is smaller than the original execution trace by approximately 3 orders of magnitude. These reductions in test program size for sink-aware fuzzing allow sink-aware fuzzing to work with much smaller abstractions of the original application, thereby significantly improving the efficiency of this step.

Qualitative comparison to other approaches. Figure 10 shows one of the several examples that FLAX generates which can not be directly expressed to the languages

```
function acceptor(input) {
 //input = ’{"action":"","val":""}’;
 must_match = ’{]:],]:]}’;
 re1 =/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g;
 re2 =/"[^"\\\n\r]*"|true|false|null|
      -?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g;
 re3 = /(?:^|:|,)(?:\s*\[)+/g;
 rep1 = input.replace(re1, "@");
 rep2 = rep1.replace(re2, "]");
 rep3 = rep2.replace(re3,"");
 if(rep3 == must_match) { return true; }
 return false;
}
```

Figure 10. An example of a acceptor slice which uses complex string operations for input validation, which is not directly expressible to the off-the-shelf string decision procedures available today.

supported by off-the-shelf existing string decision procedures [21,25], which FLAX deems as safe. We believe that even human analysis for such cases is tedious and error-prone.

5.2.3 Security Implication Evaluation and Examples

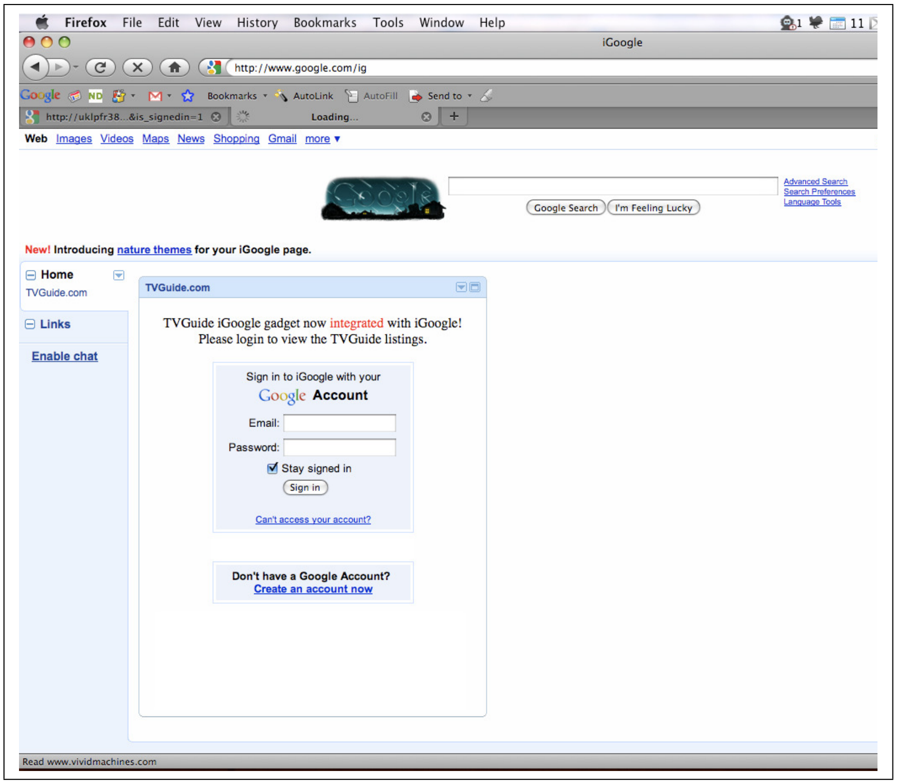
To gain insight into their severity we further analyzed the vulnerabilities reported by FLAX and created proof-of-concept exploits for a few of them to validate the threat. All vulnerabilities were disclosed to the developers either through direct communication or through CERT.

Origin Mis-attribution in Facebook Connect. FLAX reported an origin mis-attribution vulnerability for *academia.edu*, a popular academic collaboration and document sharing web site used by several academic universities. FLAX reported that the application was vulnerable due to a missing validation check on the `origin` property of a received postMessage event. We manually created a proof-of-concept exploit which demonstrates that any remote attacker could inject arbitrary script code into the vulnerable web application. On further analysis, we found that the vulnerability existed in the code for Facebook Connect library, which was used by academia.edu as well as several other web applications. We disclosed the vulnerability to Facebook developers on December 15th 2009 and they released a patch for the vulnerability within 6 hours of the disclosure.

Code Injection. FLAX reported 8 code injection vulnerabilities (DOM-based XSS) in our target applications, where untrusted values were written to code evaluation constructs in JavaScript (such as `eval`, `innerHTML`). One DOM-based XSS vulnerability was found on each of the following: 6 distinct iGoogle gadgets, an AJAX chat application (AjaxIM), and one URL parsing library's demonstration page. We manually verified that all of these were true positives and resulted in script execution in the context of the vulnerable domains, when the untrusted source was set with a malicious value. Four of the code injection vulnerabilities were exploitable when remote attackers entice the user into clicking a link of an attacker's choice. The affected web applications were also available as iGoogle gadgets and we discuss an a *gadget overwriting* attack using the CSV vulnerabilities below. The remaining 4 code injection vulnerabilities were self-XSS vulnerabilities as the untrusted input source was user-input from a form field, a text box, or a text area. As explained in section 2.1, these vulnerabilities do not directly empower a remote attacker without additional social engineering (such as enticing users into copy-and-pasting text). All gadget developers we were directly able to communicate with positively acknowledged the concern and agreed to patch the vulnerabilities.

Gadget Overwriting Attacks. In a gadget overwriting attack, a remote attacker compromises a gadget and replaces it with the content of its choice. We assume the attacker is an entity which controls a web-site and has the ability to entice the victim user into clicking a malicious link. We describe a gadget overwriting attack with an example of how it can be used to create a phishing attack layered on the gadget's CSV vulnerability. In a gadget overwriting attack, the victim clicks an untrusted link, just as in a reflected XSS attack, and sees a page such as the one shown in Figure 11 in his browser. The URL bar of the page points to the legitimate iGoogle web site, but the gadget has been compromised and displays attacker's contents: in this example, a phishing login box which tempts the user to give away his credentials for Google. If the user enters his credentials, they are sent to the attacker rather than Google or the gadget's web site. The attack mechanics are as follows. First, the victim visits the attacker's link which points to the vulnerable gadget domain (typically hosted at a subdomain of *gmodules.com*). The link exploits a code injection CSV vulnerability in the gadget and the attack payload is executed in the context of the gadget's domain. The attacker's payload then spawns a new window which points to the full iGoogle web page (`http://www.google.com/ig`) containing several gadgets including the vulnerable gadget in separate `iframes`. Lastly, the attacker's payload replaces the content of the vulnerable gadget's iframe in the new window with contents of its choice. This cross-window scripting is permitted by browser's same-origin policy because the attacker's payload and the gadget's iframe principal are the same.

We point out that Google/IG is designed such that each iGoogle gadget runs as a separate security principal hosted at a subdomain of *http://gmodules.com*. This mitigation prevents an attacker who compromises a gadget from hav-



New! Introducing nature themes for your iGoogle page.

TVGuide iGoogle gadget now integrated with iGoogle!
Please login to view the TVGuide listings.

Sign in to iGoogle with your
Google Account

Email:

Password:

☑ Stay signed in

Sign in

Can't access your account?

Don't have a Google Account?
Create an account now

Figure 11. A gadget overwriting attack layered on a CSV vulnerability. The user clicks on an untrusted link which shows the iGoogle web page with an overwritten iGoogle gadget. The URL bar continues to point to the iGoogle web page.

ing any access to the sensitive data of the *google.com* domain. In the past, Barth *et al.* described a related attack, called a *gadget hijacking* attack, which allows attackers[6] to steal sensitive data by navigating the gadget frame to a malicious site [7]. Barth *et al.* proposed new browser frame navigation policies to prevent these attacks. Gadget overwriting attacks resulting from CSV vulnerabilities in vulnerable gadgets can also allow attacker to achieve the same attack objectives as those remedied by the defenses proposed by Barth *et al.* [7].

Cookie-sink Vulnerabilities. FLAX reported a cookie corruption vulnerability in one of AskAWord iGoogle gadgets which provide the AskAWord.com dictionary and spell checker service. FLAX reported that the cookie data could be corrupted with arbitrary data and additional cookie attributes could be injected, which is a low severity vulnerability. However, on further analysis, we found that the gadget used the cookie to store the user's history of previous searches which was echoed back on the server's HTML response without any client-side or server-side validation. We subsequently informed the developers about the cookie attribute injection and the reflected XSS vulnerability through the cookie channel, and the developers patched the vulnerability on the same day.

Application Command Injection. One vulnerability reported by FLAX for AjaxIM chat application indicated that such bugs can result in practice. FLAX reported that untrusted data from an input text box could be used to inject application commands. AjaxIM uses untrusted data to construct a URL that directs application-specific commands to its backend server using `XMLHttpRequest`. These commands include adding/deleting chat rooms, adding/deleting friends and changing the user's profiles. FLAX discovered a vulnerability where an unsanitized input from an input-box is used to construct the URL that sends a GET request command to join a chat room. An attacker can exploit this vulnerability by injecting new parameters (key-value pairs) to the URL. A benign command request URL to join a chat room named *'friends'* in AjaxIM is of the form *ajaxim.php?call=joinroom&room=friends*. We confirmed that by providing a room name as

[6] A gadget attacker described by Barth *et al.* requires the privilege that the integrator embeds a gadget of the attacker choice, which is different from the attacker model in a gadget overwriting attack

‘*friends&call=addbuddy&buddy=evil*’ results in overriding the value of the `call` command from ‘joinroom’ to a command that adds an untrusted user (called “evil”) to the victim’s friend list.

The severity of this vulnerability is very limited as it does not allow a remote attacker to exploit the bug without additional social engineering. However, we informed the developers and they acknowledged the concern agreeing to fix the vulnerability.

6 Related Work

CSV vulnerabilities constitute attack categories that have similar counterparts in server-side application logic — this has driven a majority of the research on web vulnerabilities to analysis of server-side logic written in languages such as PHP. First, we discuss the techniques employed in these and compare it our taint enhanced blackbox fuzzing. Next, we compare the benefits of our approach with purely taint-based analysis approaches, and other semi-random testing based approaches. Finally, we discuss the recent frameworks proposed for analysis of JavaScript applications.

Server-side vulnerabilities. XSS, SQL injection, directory traversal, cross-site request forgery and command injection have been the most important kind of web vulnerabilities in the last few years [36]. Techniques including static analyses [22, 24], model checking [28], mixed static-dynamic analyses [4], as well as decision procedure based automated analyses [21,25] have been developed for server-side applications written in PHP and Java. Of these techniques, only a few works have aimed to precisely analyze custom validation routines. Balzarotti *et al.* were the first to identify that the use of custom sanitization could be an important source of both false positives and negatives for analysis tools in their work on Saner [4]. The proposed approach used static techniques for reasoning about multiple paths effectively. However, the sanitization analysis was limited to a subset of string functions and ignored validation checks that manifest as conditional constraints on the execution path. Though an area of active research, the more recent string decision procedures do not yet support the full generality of constraints we practically observed in our JavaScript subject applications [9,21,25].

Dynamic taint analysis approaches. Vogt *et al.* have developed taint-analysis techniques for JavaScript to study the problem of confidentiality attacks resulting from XSS vulnerabilities [39]. In addition to the features provided by their work, our taint-tracking techniques are character-level precise and accurately model the semantics of string operations as our application domain requires such precision. Purely dynamic taint-based approaches have been used for runtime defense against web attacks [18, 29, 32, 35, 37, 38, 43]. However, applying these to discover attacks is difficult because reasoning about validation checks is important for precision. Certain tools such as PHPTaint [38] approximate this by implicitly clearing the taint when data is sanitized using a built-in sanitization routine.

Directed random testing. Our taint enhanced blackbox fuzzing technique shares some of the benefits of a related technique called taint-based directed whitebox fuzzing [15]. Both techniques use taint information to narrow down the space of inputs that are relevant; however, our technique uses the knowledge of the sink to perform a directed blackbox analysis for the vulnerability as opposed to their whitebox analysis due to the limitation of current decision procedures in our application domain. Techniques developed in this paper are related to dynamic symbolic execution based approaches [11,12,16,21] which use decision procedures to explore the program space of the application. As discussed earlier, automated decision procedures for theory of strings today do not support the expressiveness to directly solve practical constraints we observe in real JavaScript applications. In comparison, our taint enhanced blackbox fuzzing algorithm is a lighter-weight mechanism which, in practice, efficiently combines the benefits of taint-based analyses with randomized testing to overcome the limitations of decision-procedure based tools.

JavaScript analysis frameworks. Several works have recently applied static analysis on JavaScript applications [14, 17]. In contrast, we demonstrate the practical effectiveness of a complimentary dynamic analysis technique and we explain the benefits of our analyses over their static counterparts. GateKeeper enforces a different set of policies using static techniques which may lead to false positives. Recent frameworks for dynamic analyses [44] have been proposed for source-level instrumentation for JavaScript; however, source-level transformations are much harder to reason about in practice due to the complexity of the JavaScript language.

Browser vulnerabilities. CSV vulnerabilities are related to, but significantly different from browser vulnerabilities [5, 7, 13, 41]. Research on these vulnerabilities has largely focused on better designs of interfaces that could be used securely by mutually untrusted principals. In this paper, we showed how web application developers use these abstractions, such as inter-frame communication interfaces, in an insecure way.

7 Conclusion

This paper presents a new class of vulnerabilities, which we call CSV vulnerabilities. We proposed a hybrid approach to automatically test JavaScript applications for the presence of these vulnerabilities. We implemented our approach in a prototype tool called FLAX. FLAX has discovered several real-world bugs in the wild, which suggests that such tools are valuable resources for security analysts and developers of rich web applications today. Results from running FLAX provide key insight into the prevalence of this class of CSV vulnerabilities with empirical examples, and point out several implicit assumptions and programming errors that JavaScript developers today make.

8 Acknowledgments

We thank Adam Barth, Stephen McCamant, Adrian Mettler, Joel Weinberger, Matthew Finifter, Devdatta Akhawe, Juan Caballero and Min Gyung Kang for helpful feedback on the paper at various stages. We are also thankful to our anonymous reviewers for suggesting improvements to our work. This work is being done while Pongsin Poosankam is a visiting student researcher at University of California, Berkeley. This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, and by the Army Research Office under Grant No. DAAD19-02-1-0389. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, or the Army Research Office.

References

[1] Ecmascript language specification, 3rd edition. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

[2] iGoogle Gadget Directory. `http://www.google.com/ig/`.

[3] Introducing JSON. `http://www.json.org/`.

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[5] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers *or* how to stop papers from reviewing themselves. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, 2008.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.

[8] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *5th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, 2008.

[9] N. Bjorner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[10] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[11] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, 2006.

[13] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 2–11, New York, NY, USA, 2007. ACM.

[14] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI*, 2009.

[15] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[17] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[18] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*, 2009.

[19] R. Hansen. XSS cheat sheet. `http://ha.ckers.org/xss.html`.

[20] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 188–198, June 2009.

[21] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, 2009.

[22] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. *The 13th International Conference on World Wide Web*, 2004.

[23] S. Jha, S. A. Seshia, and R. Limaye. On the computational complexity of satisfiability solving for string theories. *CoRR*, abs/0903.2825, 2009.

[24] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[25] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[26] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the ACM Symposium on Applied Computing*, 2006.

[27] A. Klein. DOM based cross site scripting or XSS of the third kind. Technical report, Web Application Security Consortium, 2005.

[28] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *17th USENIX Security Symposium*, 2008.

[29] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[30] S. Nanda, L.-C. Lam, and T. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware companion*, 2007.

[31] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[32] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. *20th IFIP International Information Security Conference*, 2005.

[33] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[34] E. Stinson and J. C. Mitchell. Characterizing bots' remote control behavior. In *Botnet Detection*, volume 36, pages 45–64. 2008.

[35] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2006.

[36] Symantec Corp. Symantec internet security threat report. Technical report, Apr. 2008.

[37] M. Ter Louw and V. N. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[38] W. Venema. Taint support for PHP. `http://wiki.php.net/rfc/taint`, 2007.

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

[40] W3C. HTML 5 specification. `http://www.w3.org/TR/html5/`.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *SOSP*, 2007.

[42] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, 2008.

[43] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*, 2006.

[44] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007.