GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code

Salvatore Guarnieri
University of Washington
sammyg@cs.washington.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Abstract

The advent of Web 2.0 has lead to the proliferation of client-side code that is typically written in JavaScript. This code is often combined — or mashed-up — with other code and content from disparate, mutually untrusting parties, leading to undesirable security and reliability consequences.

This paper proposes GATEKEEPER, a mostly static approach for soundly enforcing security and reliability policies for JavaScript programs. GATEKEEPER is a highly extensible system with a rich, expressive policy language, allowing the hosting site administrator to formulate their policies as succinct Datalog queries.

The primary application of GATEKEEPER this paper explores is in reasoning about JavaScript widgets such as those hosted by widget portals Live.com and Google/IG. Widgets submitted to these sites can be either malicious or just buggy and poorly written, and the hosting site has the authority to reject the submission of widgets that do not meet the site's security policies.

To show the practicality of our approach, we describe nine representative security and reliability policies. Statically checking these policies results in 1,341 verified warnings in 684 widgets, no false negatives, due to the soundness of our analysis, and false positives affecting only two widgets.

1 Introduction

JavaScript is increasingly becoming the lingua franca of the Web, used both for large monolithic applications and small *widgets* that are typically combined with other code from mutually untrusting parties. At the same time, many programming language purists consider JavaScript to be an atrocious language, forever spoiled by hard-to-analyze dynamic constructs such as `eval` and the lack of static typing. This perception has lead to a situation where code instrumentation and not static program analysis has been the weapon of choice when it comes to enforcing security policies of JavaScript code [20, 25, 29, 35].

As a recent report from Finjan Security shows, widget-based attacks are on the rise [17], making widget security an increasingly important problem to address. The report also describes well-publicised vulnerabilities in the Vista sidebar, Live.com, and Yahoo! widgets. The primary focus of this paper is on statically enforcing security and reliability policies for JavaScript code. These policies include restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, redirect and cross-site scripting detection, preventing global namespace pollution, taint checking, etc. Soundly enforcing security policies is harder that one might think at first. For instance, if we want to ensure a widget cannot call `document.write` because this construct allows arbitrary code injection, we need to either analyze or disallow tricky constructs like `eval("document" + ".write('...')")`, or `var a = document['wri' + 'te']; a('...');` which use reflection or even

```
var a = document;
var b = a.write;
b.call(this, '...')
```

which uses aliasing to confuse a potential enforcement tool. A naïve unsound analysis can easily miss these constructs. Given the availability of JavaScript obfuscators [19], a malicious widget may easily masquerade its intent. Even for this very simple policy, `grep` is far from an adequate solution.

JavaScript relies on heap-based allocation for the objects it creates. Because of the problem of object aliasing alluded to above in the `document.write` example where multiple variable names refer to the same heap object, to be able to soundly enforce the policies mentioned above, GATEKEEPER needs to statically reason about the program heap. To this end, this paper proposes the first points-to analysis for JavaScript. The programming language community has long recognized pointer analysis to be a key building block for reasoning about object-oriented programs. As a result, pointer analy-

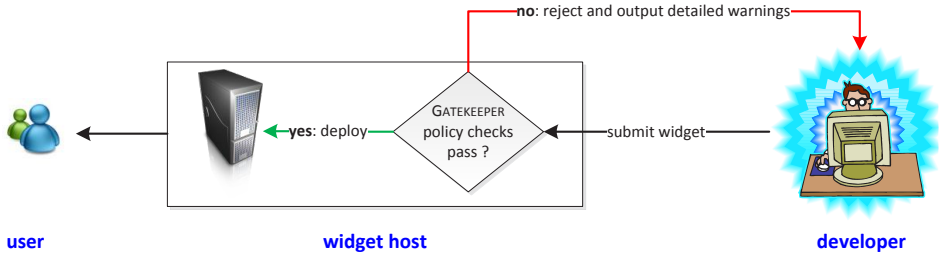

Figure 1: GATEKEEPER deployment. The three principals are: the *user*, the *widget host*, and the *widget developer*.

ses have been developed for commonly used languages such as C and Java, but nothing has been proposed for JavaScript thus far. However, a *sound* and precise points-to analysis of the *full* JavaScript language is very hard to construct. Therefore, we propose a pointer analysis for JavaScript$_{\text{SAFE}}$, a realistic subset that includes prototypes and reflective language constructs. To handle programs outside of the JavaScript$_{\text{SAFE}}$ subset, GATEKEEPER inserts runtime checks to preclude dynamic code introduction. Both the pointer analysis and nine policies we formulate on top of the points-to results are written on top of the same expressive Datalog-based declarative analysis framework. As a consequence, the hosting site interested in enforcing a security policy can program their policy in several lines of Datalog and apply it to all newly submitted widgets.

In this paper we demonstrate that, in fact, JavaScript programs are far more amenable to analysis than previously believed. To justify our design choices, we have evaluated over 8,000 JavaScript widgets, from sources such as Live.com, Google, and the Vista Sidebar. Unlike some previous proposals [35], JavaScript$_{\text{SAFE}}$ is entirely pragmatic, driven by what is found in real-life JavaScript widgets. Encouragingly, we have discovered that the use of `with`, `Function` and other "difficult" constructs [12] is similarly rare. In fact, `eval`, a reflective construct that usually foils static analysis, is only used in 6% of our benchmarks. However, statically unknown field references such `a[index]`, dangerous because these can be used to get to `eval` through `this['eval']`, etc., and `innerHTML` assignments, dangerous because these can be used to inject JavaScript into the DOM, are more prevalent than previously thought. Since these features are quite common, to prevent runtime code introduction and maintain the soundness of our approach, GATEKEEPER inserts dynamic checks around statically unresolved field references and `innerHTML` assignments.

This paper contains a comprehensive large-scale experimental evaluation. To show the practicality of GATEKEEPER, we present nine representative policies for security and reliability. Our policies include restricting widgets capabilities to prevent calls to `alert` and the use of the `XmlHttpRequest` object, looking for global namespace pollution, detecting browser redirects leading to cross-site scripting, preventing code injection, taint checking, etc. We experimented on 8,379 widgets, out of which 6,541 are analyzable by GATEKEEPER [1]. Checking our nine policies resulted in us discovering a total of 1,341 verified warnings that affect 684, with only 113 false positives affecting only two widgets.

1.1 Contributions

This paper makes the following contributions:

- We propose the first points-to analysis for JavaScript programs. Our analysis is the first to handle a prototype-based language such as JavaScript. We also identify JavaScript$_{\text{SAFE}}$, a statically analyzable subset of the JavaScript language and propose lightweight instrumentation that restricts runtime code introduction to handle many more programs outside of the JavaScript$_{\text{SAFE}}$ subset.
- On the basis of points-to information, we demonstrate the utility of our approach by describing nine representative security and reliability policies that are soundly checked by GATEKEEPER, meaning no false negatives are introduced. These policies are expressed in the form of succinct declarative Datalog queries. The system is highly extensible and easy to use: each policy we present is only several lines of Datalog. Policies we describe include restricting widget capabilities, making sure built-in objects are not modified, preventing code injection attempts, etc.
- Our experimental evaluation involves in excess of *eight thousand* publicly available JavaScript widgets from Live.com, the Vista Sidebar, and Google. We flag a total of 1,341 policy violations spanning 684 widgets, with 113 false positives affecting only two widgets.

[1] Because we cannot ensure soundness for the remaining 1,845 widgets, we reject them without further policy checking.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview of our approach and summarizes the most significant analysis challenges. Section 3 provides a deep dive into the details of our analysis; a reader interested in learning about the security policies may skip this section on the first reading. Section 4 describes nine static checkers we have developed for checking security policies of JavaScript widgets. Section 5 summarizes the experimental results. Finally, Sections 6 and 7 describe related work and conclude.

2 Overview

As a recent report from Finjan Security shows, widget-based attacks are on the rise [17]. Exploits such as those in a Vista sidebar contacts widget, a Live.com RSS widget, and a Yahoo! contact widget [17, 27] not only affect unsuspecting users, they also reflect poorly on the hosting site. In a way, widgets are like operating system drivers: their quality directly affects the perceived quality of the underlying OS. While driver reliability and security has been subject of much work [7], widget security has received relatively little attention. Just like with drivers, however, widgets can run in the same page (analogous to an OS process) as the rest of the hosting site. Because widget flaws can negatively impact the rest of the site, it is out aim to develop tools to ensure widget security and reliability.

While our proposed static analysis techniques are much more general and can be used for purposes as diverse as program optimization, concrete type inference, and bug finding, the focus of this paper is on soundly enforcing security and reliability policies of JavaScript widgets. There are three principals that emerge in that scenario: the widget hosting site such as Live.com, the developer submitting a particular widget, and the user on whose computer the widget is ultimately executed. The relationship of these principals is shown in Figure 1. We are primarily interested in helping the *widget host* ensure that their users are protected.

2.1 Deployment

We envision GATEKEEPER being deployed and run by the widget hosting provider as a mandatory checking step in the online submission process, required before a widget is accepted from a widget developer. Many hosts already use captchas to ensure that the submitter is human. However, captchas say nothing about the quality and intent of the code being submitted. Using GATEKEEPER will ensure that the widget being submitted complies with the policies chosen by the host. A hosting provider has the authority to reject some of the submitted widgets, instructing widgets authors to change their code until it passes the policy checker, not unlike tools like the static driver verifier for Windows drivers [24]. Our policy checker outputs detailed information about why a particular widget fails, annotated with line numbers, which allows the widget developer to fix their code and resubmit.

2.2 Designing Static Language Restrictions

To enable sound analysis, we first restrict the input to be a subset of JavaScript as defined by the EcmaScript-262 language standard. Unlike previous proposals that significantly hamper language expressiveness for the sake of safety [13], our restrictions are relatively minor. In particular, we disallow the `eval` construct and its close cousin, the `Function` object constructor as well as functions `setTimeout` and `setInterval`. All of these constructs take a string and execute it as JavaScript code. The fundamental problem with these constructs is that they introduce new code at runtime that is unseen — and therefore cannot be reasoned about — by the static analyzer. These reflective constructs have the same expressive power: allowing one of them is enough to have the possibility of arbitrary code introduction.

We also disallow the use of `with`, a language feature that allows to dynamically substitute the symbol lookup scope, a feature that has few legitimate uses and significantly complicates static reasoning about the code. As our treatment of prototypes shows, it is in fact possible to handle `with`, but it is only used in 8% of our benchmarks. Finally, while these restrictions might seem draconian at first, they are very similar to what a recently proposed strict mode for JavaScript enforces [14].

We do allow reflective constructs `Function.call`, `Function.apply`, and the `arguments` array. Indeed, `Function.call`, the construct that allows the caller of a function to set the callee's `this` parameter, is used in 99% of Live widgets and can be analyzed statically with relative ease, so we handle this language feature. The prevalence of `Function.call` can be explained by a common coding pattern for implementing a form of inheritance, which is encouraged by Live.com widget documentation, and is found pretty much verbatim in most widgets.

In other words, our analysis choices are driven by the statistics we collect from 8,379 real-world widgets and not hypothetical considerations. More information about the relative prevalence of "dangerous" language features can be found in Figure 3. The most common "unsafe" features we have to address are `.innerHTML` assignments and statically unresolved field references. Because they are so common, we cannot simply disallow them, so we check them at runtime instead.

To implement restrictions on the allowed input, in

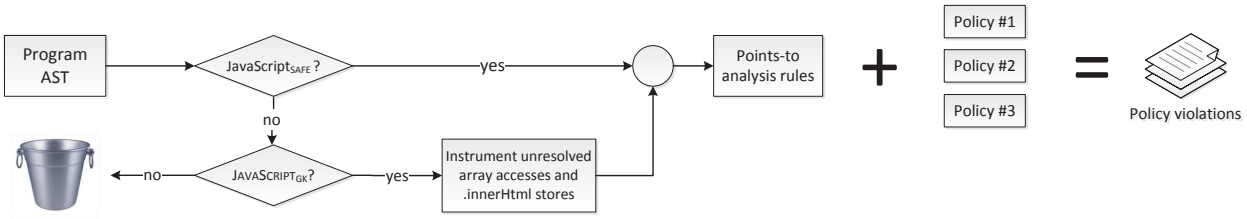

Figure 2: GATEKEEPER analysis architecture.

	Sidebar		Windows Live		Google	
JavaScript Construct	**Affected**	**%**	**Affected**	**%**	**Affected**	**%**
Non-Const Index	1,736	38.6%	176	6.5%	192	16.4%
`with`	422	9.4%	2	.1%	2	.2%
`arguments`	175	3.9%	6	.2%	3	.3%
`setTimeout`	824	18.3%	49	1.8%	65	5.6%
`setInterval`	377	8.4%	16	.6%	13	1.1%
`eval`	353	7.8%	10	.4%	55	4.7%
`apply`	173	3.8%	29	1.1%	6	.5%
`call`	151	3.4%	2,687	99.0%	4	.3%
`Function`	142	3.2%	4	.1%	21	1.8%
`document.write`	102	2.3%	1	0%	108	9.2%
`.innerHTML`	1,535	34.1%	2,053	75.6%	288	24.6%

Figure 3: Statistics for 4,501 widgets from Sidebar and 2,714 widgets from Live, and 1,171 widgets from Google.

our JavaScript parser we flag the use of lexer tokens `eval`, `Function`, and `with`, as well as `setTimeout`, and `setInterval`. We need to disallow all of these constructs because letting one of them through is enough for arbitrary code introduction. The feature we cannot handle simply using lexer token blacklisting is `document.write`. We first optimistically assume that no calls to `document.write` are present and then proceed to verify this assumption as described in Section 4.3. This way our analysis remains sound.

We consider two subsets of the JavaScript language, JavaScript$_{SAFE}$ and JavaScript$_{GK}$. The two subsets are compared in Figure 4. If the program passes the checks above *and* lacks statically unresolved array accesses and `innerHTML` assignments, it is declared to be in JavaScript$_{SAFE}$. Otherwise, these dangerous accesses are instrumented and it is declared in the JavaScript$_{GK}$ language subset. To resolve field accesses, we run a local dataflow constant propagation analysis [1] to identify the use of constants as field names. In other words, in the following code snippet

```
var fieldName = 'f';
a[fieldName] = 3;
```

the second line will be correctly converted into `a.f` $=$ `3`.

2.3 Analysis Stages

The analysis process is summarized in Figure 2. If the program is outside of JavaScript$_{GK}$, we reject it right away. Otherwise, we first traverse the program representation and output a database of facts, expressed in Datalog notation. This is basically a declarative database representing what we need to know about the input JavaScript program. We next combine these facts with a representation of the native environment of the browser discussed in Section 3.4 and the points-to analysis rules. All three are represented in Datalog and can be easily combined. We pass the result to bddbddb, an off-the-shelf declarative solver [33], to produce policy violations. This provides for a very agile experience, as changing the policy usually only involves editing several lines of Datalog.

2.4 Analyzing the JavaScript$_{SAFE}$ Subset

For a JavaScript$_{SAFE}$ program, we normalize each function to a set of statements shown in Figure 5. Note that the JavaScript$_{SAFE}$ language, which we shall extend in Section 3 is very much Java-like and is therefore amenable to inclusion-based points-to analysis [33]. What is not made explicit by the syntax is that JavaScript$_{SAFE}$ is a prototype-based language, not a class-based one. This means that objects do not belong to explicitly declared classes. Instead, a object creation can be based on a function, which becomes that object's prototype. Furthermore, we support a restricted form of reflection including `Function.call`,

Feature	JavaScript$_{SAFE}$	JavaScript$_{GK}$
UNCONTROLLED CODE INJECTION		
Unrestricted `eval`	✗	✗
`Function` constructor	✗	✗
`setTimeout`, `setInterval`	✗	✗
`with`	✗	✗
`document.write`	✗	✗
Stores to code-injecting fields `innerHTML`, `onclick`, etc.	✗	✗
CONTROLLED REFLECTION		
`Function.call`	✓	✓
`Function.apply`	✓	✓
`arguments` array	✓	✓
INSTRUMENTATION POINTS		
Non-static field stores	✗	✓
`innerHTML` assignments	✗	✓

Figure 4: Support for different dynamic EcmaScript-262 language features in JavaScript$_{SAFE}$ and JavaScript$_{GK}$ language subsets.

$s ::=$	
ϵ | $s; s$	[EMPTY] [SEQUENCE]
$v_1 = v_2$ | $v = \bot$	[ASSIGNMENT] [PRIMASSIGNMENT]
`return` $v;$ | $v =$ `new` $v_0(v_1, ..., v_n);$	[RETURN] [CONSTRUCTOR]
$v = v_0(v_{this}, v_1, v_2, \ldots, v_n);$	[CALL]
$v_1 = v_2.f;$ | $v_1.f = v_2;$	[LOAD] [STORE]
$v =$ `function`$(v_1, ..., v_n)\ \{s;\};$	[FUNCTIONDECL]

Figure 5: JavaScript$_{SAFE}$ statement syntax in BNF.

CALLS$(i : I, h : H)$	indicates when call site i invokes method h
FORMAL$(h : H, z : Z, v : V)$	records formal arguments of a function
METHODRET$(h : H, v : V)$	records the return value of a method
ACTUAL$(i : I, z : Z, v : V)$	records actual arguments of a function call
CALLRET$(i : I, v : V)$	records the return value for a call site
ASSIGN$(v_1 : V, v_2 : V)$	records variable assignments
LOAD$(v_1 : V, v_2 : V, f : F)$	represents field loads
STORE$(v_1 : V, f : F, v_2 : V)$	represents field stores
PTSTO$(v : V, h : H)$	represents a points-to relation for variables
HEAPPTSTO$(h_1 : H, f : F, h_2 : H)$	represents a points-to relations for heap objects
PROTOTYPE$(h_1 : H, h_2 : H)$	records object prototypes

Figure 6: Datalog relations used for program representation.

`Function.apply`, and the `arguments` array. The details of pointer analysis are shown in the Datalog rules Figure 8 and discussed in detail in Section 3.

One key distinction of our approach with Java is that there is basically no distinction of heap-allocation objects and function closures in the way the analysis treats them. In other words, at a call site, if the base of a call "points to" an allocation site that corresponds to a function declaration, we statically conclude that that function might be called. While it may be possible to recover portions of the call graph through local analysis, we interleave call graph and points-to analysis in our approach.

We are primarily concerned with analyzing objects or references to them in the JavaScript heap and not primitive values such as integers and strings. We therefore do not attempt to faithfully model primitive value manipulation, lumping primitive values into PRIMASSIGNMENT statements.

2.5 Analysis Soundness

The core static analysis implemented by GATEKEEPER is sound, meaning that we statically provide a conservative approximation of the runtime program behavior. Achieving this for JavaScript with all its dynamic features is far from easy. As a consequence, we extend our soundness guarantees to programs utilizing a smaller subset of the language. For programs within JavaScript$_{SAFE}$, our analy-

$v_1 = v_2$	ASSIGN(v_1, v_2).	[ASSIGNMENT]
$v = \bot$		[BOTASSIGNMENT]
`return` v	CALLRET(v).	[RETURN]
$v =$ `new` $v_0(v_1, v_2, ..., v_n)$	PTSTO(v, d_{fresh}). PROTOTYPE(d_{fresh}, h) :– PTSTO(v_0, m), HEAPPTSTO$(m,$ `"prototype"`$, h)$. for $z \in \{1..n\}$, generate ACTUAL(i, z, v_z). CALLRET(i, v).	[CONSTRUCTOR]
$v = v_0(v_{this}, v_1, v_2, \ldots, v_n)$	for $z \in \{1..n, this\}$, generate ACTUAL(i, z, v_z). CALLRET(i, v).	[CALL]
$v_1 = v_2.f$	LOAD(v_1, v_2, f).	[LOAD]
$v_1.f = v_2$	STORE(v_1, f, v_2).	[STORE]
$v =$ `function`$(v_1, ..., v_n)$ $\{s\}$	PTSTO(v, d_{fresh}). HEAPPTSTO$(d_{fresh},$ `"prototype"`$, p_{fresh})$. FUNCDECL(d_{fresh}). PROTOTYPE(p_{fresh}, h_{FP}). for $z \in \{1..n\}$, generate FORMAL(d_{fresh}, z, v_z). METHODRET(d_{fresh}, v).	[FUNCTIONDECL]

Figure 7: Datalog facts generated for each JavaScript$_{\text{SAFE}}$ statement.

sis is sound. For programs within GATEKEEPER, our analysis is sound *as long as no code introduction is detected with the runtime instrumentation we inject*. This is very similar to saying that, for instance, a Java program is not going to access outside the boundaries of an array as long as no `ArrayOutOfBoundsException` is thrown. Details of runtime instrumentation are presented in Section 3.2. The implications of soundness is that GATEKEEPER is guaranteed to flag all policy violations, at the cost of potential false positives.

We should also point out that the GATEKEEPER analysis is inherently a *whole-program analysis*, not a modular one. The need to statically have access to the entire program is why we work so hard to limit language features that allow dynamic code loading or injection. We also generally model the runtime — or *native* — environment in which the JavaScript code executes. Our approach is sound, assuming that our native environment model is conservative. This last claim is similar to asserting that a static analysis for Java is sound, as long as `native` functions and libraries are modeled conservatively, a commonly used assumption. We also assume that the runtime instrumentation we insert is able to handle the relevant corner cases a deliberately malicious widget might try to exploit, admittedly a challenging task, as further explained in Section 3.2.

3 Analysis Details

This section is organized as follows. Section 3.1 talks about pointer analysis in detail[2]. Section 3.2 discusses the runtime instrumentation inserted by GATEKEEPER. Section 3.3 talks about how we normalize JavaScript AST to fit into our intermediate representation. Section 3.4 talks about how we model the native JavaScript environment.

3.1 Pointer Analysis

In this paper, we describe how to implement a form of inclusion-based Andersen-style flow- and context-sensitive analysis [3] for JavaScript. It remains to be seen whether flow and context sensitivity significantly improve analysis precision; our experience with the policies in Section 4 has not shown that to be the case. We use allocation sites to approximate runtime heap objects. A key distinction of our approach in the lack of a call graph to start with: our technique allows call graph inference and points-to analysis to be interleaved. As advocated elsewhere [21], the analysis itself is expressed declaratively: we convert the program into a set of facts, to which we

[2] We refer the interested reader to a companion technical report [22] that discusses handling of reflective constructs `Function.call`, `Function.apply`, and `arguments`.

% Basic rules		
$\textsc{PtsTo}(v, h)$	$:-$	$\textsc{Alloc}(v, h).$
$\textsc{PtsTo}(v, h)$	$:-$	$\textsc{FuncDecl}(v, h).$
$\textsc{PtsTo}(v_1, h)$	$:-$	$\textsc{PtsTo}(v_2, h), \textsc{Assign}(v_1, v_2).$
$\textsc{DirectHeapStoresTo}(h_1, f, h_2)$	$:-$	$\textsc{Store}(v_1, f, v_2), \textsc{PtsTo}(v_1, h_1), \textsc{PtsTo}(v_2, h_2).$
$\textsc{DirectHeapPointsTo}(h_1, f, h_2)$	$:-$	$\textsc{DirectHeapStoresTo}(h_1, f, h_2).$
$\textsc{PtsTo}(v_2, h_2)$	$:-$	$\textsc{Load}(v_2, v_1, f), \textsc{PtsTo}(v_1, h_1), \textsc{HeapPtsTo}(h_1, f, h_2).$
$\textsc{HeapPtsTo}(h_1, f, h_2)$	$:-$	$\textsc{DirectHeapPointsTo}(h_1, f, h_2).$
% Call graph		
$\textsc{Calls}(i, m)$	$:-$	$\textsc{Actual}(i, 0, c), \textsc{PtsTo}(c, m).$
% Interprocedural assignments		
$\textsc{Assign}(v_1, v_2)$	$:-$	$\textsc{Calls}(i, m), \textsc{Formal}(m, z, v_1), \textsc{Actual}(i, z, v_2), z > 0.$
$\textsc{Assign}(v_2, v_1)$	$:-$	$\textsc{Calls}(i, m), \textsc{MethodRet}(m, v_1), \textsc{CallRet}(i, v_2).$
% Prototype handling		
$\textsc{HeapPtsTo}(h_1, f, h_2)$	$:-$	$\textsc{Prototype}(h_1, h), \textsc{HeapPtsTo}(h, f, h_2).$

Figure 8: Pointer analysis inference rules for JavaScript$_{\text{SAFE}}$ expressed in Datalog.

apply inference rules to arrive at the final call graph and points-to information.

Program representation. We define the following *domains* for the points-to analysis GATEKEEPER performs: heap-allocated objects and functions H, program variables V, call sites I, fields F, and integers Z. The analysis operates on a number of relations of fixed arity and type, as summarized in Figure 6.

Analysis stages. Starting with a set of initial input relation, the analysis follows inference rules, updating intermediate relation values until a fixed point is reached. Details of the declarative analysis and BDD-based representation can be found in [32]. The analysis proceeds in stages. In the first analysis stage, we traverse the normalized representation for JavaScript$_{\text{SAFE}}$ shown in Figure 5. The basic facts that are produced for every statement in the JavaScript$_{\text{SAFE}}$ program are summarized in Figure 7. As part of this traversal, we fill in relations ASSIGN, FORMAL, ACTUAL, METHODRET, CALLRET, etc. This is a relatively standard way to represent information about the program in the form of a database of facts. The second stage applies Datalog inference rules to the initial set of facts. The analysis rules are summarized in Figure 8. In the rest of this section, we discuss different aspects of the pointer analysis.

3.1.1 Call Graph Construction

As we mentioned earlier, call graph construction in JavaScript presents a number of challenges. First, unlike a language with function pointers like C, or a language with a fixed class hierarchy like Java, JavaScript does *not* have any initial call graph to start with. Aside from local analysis, the only conservative default we have to fall back to when doing static analysis is "any call site calls every declared function," which is too imprecise.

Instead, we chose to combine points-to and call graph constraints into a single Datalog constraint system and resolve them at once. Informally, intraprocedural data flow constraints lead to new edges in the call graph. These in turn lead to new data flow edges when we introduce constraints between newly discovered arguments and return values. In a sense, function declarations and object allocation sites are treated very much the same in our analysis. If a variable $v \in V$ may point to function declaration f, this implies that call $v()$ may invoke function f. Allocation sites and function declarations flow into the points-to relation PTSTO through relations ALLOC and FUNCDECL.

3.1.2 Prototype Treatment

The JavaScript language defines two lookup chains. The first is the lexical (or static) lookup chain common to all closure-based languages. The second is the prototype chain. To resolve `o.f`, we follow `o`'s prototype, `o`'s prototype's prototype, etc. to locate the first object associated with field `f`.

Note that the object prototype (sometimes denoted as [[`Prototype`]] in the ECMA standard) is different from the `prototype` field available on any object. We model [[`Prototype`]] through the PROTOTYPE relation in our static analysis. When $\textsc{Prototype}(h_1, h_2)$ holds, h_1's internal [[`Prototype`]] may be h_2 [3].

[3] We follow the EcmaScript-262 standard; Firefox makes

Two rules in Figure 7 are particularly relevant for prototype handling: [CONSTRUCTOR] and [FUNCTIONDECL]. In the case of a constructor call, we allocate a new heap variable d_{fresh} and make the return result of the call v point to it. For (every) function m the constructor call invokes, we make sure that m's `prototype` field is connected with d_{fresh} through the PROTOTYPE relation. We also set up ACTUAL and CALLRET values appropriately, for $z \in \{1..n\}$. In the regular [CALL] case, we also treat the `this` parameter as an extra actual parameter.

In the case of a [FUNCTIONDECL], we create two fresh allocation site, d_{fresh} for the function and p_{fresh} for the newly create `prototype` field for that function. We use shorthand notion h_{FP} to denote object `Function.prototype` and create a PROTOTYPE relation between p_{fresh} and h_{FP}. We also set up HEAPPTSTO relation between d_{fresh} and p_{fresh} objects. Finally, we set up relations FORMAL and METHODRET, for $z \in \{1..n\}$.

Example 1. The example in Figure 9 illustrates the intricacies of prototype manipulation. Allocation site a_1 is created on line 2. Every declaration creates a declaration object and a prototype object, such as d_T and p_T. Rules in Figure 10 are output as this code is processed, annotated with the line number they come from. To resolve the call on line 4, we need to determine what `t.bar` points to. Given $\text{PTSTO}(t, a_1)$ on line 2, this resolves to the following Datalog query:

$$\text{HEAPPTSTO}(a_1, \texttt{"bar"}, X)?$$

Since there is nothing d_T points to *directly* by following the `bar` field, the PROTOTYPE relation is consulted. $\text{PROTOTYPE}(a_1, p_T)$ comes from line 2. Because we have $\text{HEAPPTSTO}(p_T, \texttt{"bar"}, d_{\texttt{bar}})$ on line 3, we resolve X to be $d_{\texttt{bar}}$. As a result, the call on line 4 may correctly invoke function `bar`. Note that our rules do not try to keep track of the order of objects in the prototype chain. □

3.2 Programs Outside JavaScript$_{\text{SAFE}}$

The focus of this section is on runtime instrumentation for programs outside JavaScript$_{\text{SAFE}}$, but within the JavaScript$_{\text{GK}}$ JavaScript subset that is designed to prevent runtime code introduction.

3.2.1 Rewriting `.innerHTML` Assignments

`innerHTML` assignments are a common dangerous language feature that may prevent GATEKEEPER from statically seeing all the code. We disallow it in JavaScript$_{\text{SAFE}}$, but because it is so common, we still allow it in the JavaScript$_{\text{GK}}$ language subset. While in many cases the right-hand side of `.innerHTML` assignments is a constant, there is an unfortunate coding pattern encouraged by Live widgets that makes static analysis difficult, as shown in Figure 11. The `url` value, which is the result concatenating of a constant URL and `widgetURL` is being used on the right-hand side and could be used for code injection. An assignment $v_1.\texttt{innerHTML} = v_2$ is rewritten as

```
if (__IsUnsafe(v2)) {
    alert("Disguised eval attempt at <file>:<line>");
} else {
    v1.innerHTML = v2;
}
```

where `__IsUnsafe` disallows all but very simple HTML. Currently, `__IsUnsafe` is implemented as follows:

```
function __IsUnsafe(data)) {
    return (toStaticHTML(data)===data);
}
```

`toStaticHTML`, a built-in function supported in newer versions of Internet Explorer, removes attempts to introduce script from a piece of HTML. An alternative is to provide a parser that allows a subset of HTML, an approach that is used in WebSandbox [25]. The call to `alert` is optional — it is only needed if we want to warn the user. Otherwise, we may just omit the statement in question.

3.2.2 Rewriting Unresolved Heap Loads and Stores

That syntax for JavaScript$_{\text{GK}}$ supported by GATEKEEPER has an extra variant of LOAD and STORE rules for associative arrays, which introduce Datalog facts shown below:

$$v_1 = v_2[*] \quad \text{LOAD}(v_1, v_2, \_) \quad \text{[ARRAYLOAD]}$$
$$v_1[*] = v_2 \quad \text{STORE}(v_1, \_, v_2) \quad \text{[ARRAYSTORE]}$$

When the indices of an associative array operation cannot be determined statically, we have to be conservative. This means that any field that may be reached can be accessed. This also means that to be conservative, we must consider the possibility that *any* field may be affected as well: the field parameter is unconstrained, as indicated by an _ in the Datalog rules above.

Example 2. Consider the following motivating example:

```
1. var a = {
2.     'f' : function(){...},
3.     'g' : function(){...}, ...};
5. a[x + y] = function(){...};
6. a.f();
```

If we cannot statically decide which field of object `a` is being written to on line 5, we have to conservatively assume

[[`Prototype`]] accessible through a non-standard field `__proto__`.

```
1. function T(){ this.foo = function(){ return 0}};   d_T, p_T
2. var t = new T();                                    a_1
3. T.prototype.bar = function(){ return 1; };          d_bar, p_bar
4. t.bar(); // return 1
```

Figure 9: Prototype manipulation example.

1. PTSTO(T, d_T). HEAPPTSTO(d_T, "prototype", p_T). PROTOTYPE(p_T, h_{FP}).
2. PTSTO(t, a_1). PROTOTYPE(a_1, p_T).
3. HEAPPTSTO(p_T, "bar", d_{bar}). HEAPPTSTO(d_{bar}, "prototype", p_{bar}). PROTOTYPE($p_{\text{bar}}, h_{\text{FP}}$).

Figure 10: Rules created for the prototype manipulation example in Figure 9.

that the assignment could be to field `f`. This can affect which function is called on line 6. □

Moreover, any statically unresolved store may introduce additional code through writing to the `innerHTML` field that will be never seen by static analysis. We rewrite statically unsafe stores $v_1[i] = v_2$ by blacklisting fields that may lead to code introduction:

```
if (i==="onclick" ||  i==="onkeypress" || ...) {
    alert("Disguised eval attempt at <file>:<line>");
} else
if(i==="innerHTML" && __IsUnsafe(v2)){
   alert("Unsafe innerHTML at <file>:<line>");
} else {
    v1[i] = v2;
}
```

Note that we use $===$ instead of $==$ because the latter form will try to coarse `i` to a string, which is not our intention. Also note that it's impossible to introduce a TOCTOU vulnerability of having `v2` change "underneath us" after the safety check because of the single-threaded nature of JavaScript.

Similarly, statically unsafe loads of the form $v_1 = v_2[i]$ can be restricted as follows:

```
if (i==="eval" || i==="setInterval" ||
    i==="setTimeout" || i==="Function" ||...)
{
    alert("Disguised eval attempt at <file>:<line>");
} else {
    v1 = v2[i];
}
```

Note that we have to check for unsafe functions such as `eval`, `setInterval`, etc. While we reject them as tokens for JavaScript$_{\text{SAFE}}$, they may still creep in through statically unresolved array accesses. Note that to preserve the soundness of our analysis, care must be taken to keep the blacklist comprehensive.

While we currently use a blacklist and do our best to keep it as complete as we can, ideally blacklist design and browser runtime design would go hand-in-hand. We really could benefit from a browser-specified form of runtime safety, as illustrated by the `use strict` pragma [14]. A conceptually safer, albeit more restrictive, approach is to resort to a whitelist of allowed fields.

3.3 Normalization Details

In this section we discuss several aspects of normalizing the JavaScript AST. Note that certain tricky control flow and reflective constructs like `for...in` are omitted here because our analysis is flow-insensitive.

Handling the global object. We treat the global object explicitly by introducing a variable `global` and then assigning to its fields. One interesting detail is that global variable reads and writes become *loads* and *stores* to fields of the global object, respectively.

Handling of `this` argument in function calls. One curious feature of JavaScript is its treatment of the `this` keyword, which is described in section 10.2 of the EcmaScript-262 standard. For calls of the form `f`$(\texttt{x}, \texttt{y}, ...)$, the `this` value is set by the runtime to the global object. This is a pretty surprising design choice, so we translate syntactic forms `f`$(\texttt{x}, \texttt{y}, ...)$ and `o.f`$(\texttt{x}, \texttt{y}, ...)$ differently, passing the global object in place of `this` in the former case.

3.4 Native Environment

The browser embedding of the JavaScript engine has a large number of pre-defined objects. In addition to `Array`, `Date`, `String`, and other objects defined by the EcmaScript-262 standard, the browser defines objects such as `Window` and `Document`.

Native environment construction. Because we are doing whole-program analysis, we need to create stubs for

```
this.writeWidget = function(widgetURL) {
    var url = "http://widgets.clearspring.com/csproduct/web/show/flash?
        opt=-MAX/1/-PUR/http%253A%252F%252Fwww.microsoft.com&url="+widgetURL;

    var myFrame = document.createElement("div");
    myFrame.innerHTML = '<iframe id="widgetIFrame" scrolling="no"
        frameborder="0" style="width:100%;height:100%;border:0px" src="'+
        url+'"></iframe>';
    ...
}
```

Figure 11: `innerHTML` assignment example

the native environment so that calls to built-in methods resolve to actual functions. We recursively traverse the native embedding. For every function we encounter, we provide a default stub `function(){return undefined;}`. The resulting set of declarations looks as follows:

```
var global = new Object();
// this references in the global namespace refer to global
var this = global;
global.Array = new Object();
global.Array.constructor = new function(){return undefined;}
global.Array.join = new function(){return undefined;}
 ...
```

Note that we use an explicit `global` object to host a namespace for our declarations instead of the implicit `this` object that JavaScript uses. In most browser implementations, the global `this` object is aliased with the `window` object, leading to the following declaration: `global.window = global;`.

Soundness. However, as it turns out, creation of a *sound* native environment is more difficult than that. Indeed, the approach above assumes that the built-in functions return objects that are never aliased. This fallacy is most obviously demonstrated by the following code:

```
var parent_div = document.getElementById('header');
var child_div = document.createElement('div');
parent_div.appendChild(child_div);
var child_div2 = parent_div.childNodes[0];
```

In this case, `child_div` and `child_div2` are aliases for the same DIV element. if we pretend they are not, we will miss an existing alias. We therefore model operations such as `appendChild`, etc. in JavaScript code, effectively creating *mock-ups* instead of native browser-provided implementations.

In our implementation, we have done our best to ensure the soundness of the environment we produce by starting with an automatically generated collection of stubs and augmenting them by hand to match what we believe the proper browser semantics to be. This is similar to modeling `memcpy` in a static analysis of C code or `native` methods in a static analysis for Java. However, as with two instance of foreign function interface (FFI) modeling above, this form of manual involvement is often error-prone. It many also unfortunately compromise the soundness of the overall approach, both because of implementation mistakes and because of browser incompatibilities. A potential alternative to our current approach and part of our future work is to consider a standards-compliant browser that that implements some of its library code in JavaScript, such as Chrome. With such an approach, because libraries become amenable to analysis, the need for manually constructed stubs would be diminished.

When modeling the native environment, when in doubt, we tried to err on the side of caution. For instance, we do not attempt to model the DOM very precisely, assuming initially that any DOM-manipulating method may return any DOM node (effectively all DOM nodes are statically modeled as a single allocation site). Since our policies in Section 4 do not focus on the DOM, this imprecise, but sound modeling does not result in false positives.

4 Security and Reliability Policies

This section is organized as follows. Sections 4.1–4.4 talk about six policies that apply to widgets from all widgets hosts we use in this paper (Live, Sidebar, and Google). Section 4.5 talks about host-specific policies, where we present two policies specific to Live and one specific to Sidebar widgets. Along with each policy, we present the Datalog query that is designed to find policy violations. We have run these queries on our set of 8,379 benchmark widgets. A detailed discussion of our experimental findings can be found in Section 5.

4.1 Restricting Widget Capabilities

Perhaps the most common requirement for a system that reasons about widgets is the ability to restrict code capabilities, such as disallowing calling a particular function, using a particular object or namespace, etc. The Live Widget Developer Checklist provides many such examples [34]. This is also what systems like Caja and WebSandbox aim to accomplish [25, 29]. We can achieve the same goal statically.

Pop-up boxes represent a major annoyance when using

web sites. Widgets that bring up popup boxes, achieved by calling function `alert` in JavaScript, can be used for denial-of-service against the user. In fact, the `alert` box prevention example below comes from a widget sample that asynchronously spawns new alert boxes; this widget is distributed with WebSandbox [26]. The following query ensures that the `alert` routine is never called:

Query output: $AlertCalls(i : I)$

$$\begin{array}{lll} GlobalSym(m,h) & :- & \textsc{PtsTo}(\texttt{"global"}, g), \\ & & \textsc{HeapPtsTo}(g, m, h). \\ AlertCalls(i) & :- & GlobalSym(\texttt{"alert"}, h), \\ & & \textsc{Calls}(i, h). \end{array}$$

To define $AlertCalls$, we first define an auxiliary query $GlobalSym : F \times H$ used for looking up global functions such as `alert`. On the right-hand side, $g \in H$ is the explicitly represented global object pointed to by variable `global`. Following field m takes us to the heap object h of interest. $AlertCalls$ instantiates this query for field `alert`. Note that there are several references to it in the default browser environment such as `window.alert` and `document.alert`. Since they all are aliases for the same function, the query above will spot all calls, independently of the the reference being used.

4.2 Detecting Writes to Frozen Objects

We disallow changing properties of built-in objects such as `Boolean`, `Array`, `Date`, `Function`, `Math`, `Document`, `Window`, etc. to prevent environment pollution attacks such as prototype hijacking [9]. This is similar to *frozen objects* proposed in EcmaScript 4. The query in Figure 12 looks for attempts to add or update properties of JavaScript built-in objects specified by the auxiliary query $BuiltInObject$, including attempts to change their prototypes: The rules above handle the case of assigning to properties of these built-in objects directly. Often, however, a widget might attempt to assign properties of the `prototype` of an object as in `Function.prototype.apply = function(){...}`. We can prevent this by first defining a recursive heap reachability relation $Reaches$:

$$\begin{array}{lll} Reaches(h_1, h_2) & :- & \textsc{HeapPtsTo}(h_1, \_, h_2). \\ Reaches(h_1, h_2) & :- & \textsc{HeapPtsTo}(h_1, \_, h'), \\ & & Reaches(h', h_2). \end{array}$$

and then adding to the $FrozenViolation$ definition:

$$\begin{array}{lll} FrozenViolation(v) & :- & \textsc{Store}(v, \_, \_), \\ & & \textsc{PtsTo}(v, h'), \\ & & BuiltInObject(h), \\ & & Reaches(h, h'). \end{array}$$

An example of a typical policy violation from our experiments is shown below:

```
Array.prototype.feed = function(o, s){
    if(!s){s=o;o={};}
    var k,p=s.split(":");
    while(typeof(k=p.shift())!="undefined")
        o[k]=this.shift();
    return o;
}
```

4.3 Detecting Code Injection

As discussed above, `document.write` is a routine that allows the developer to output arbitrary HTML, thus allowing code injection through the use of <script> tags. While verbatim calls to `document.write` can be found using `grep`, it is easy to disguise them through the use of aliasing:

```
var x = document;
var y = x.write;
y("<script>alert('hi');</script>");
```

The query below showcases the power of points-to analysis. In addition to finding the direct calls, the query below will correctly determine that the call to `y` invokes `document.write`.

Query output: $DocumentWrite(i : I)$

$$\begin{array}{lll} DocumentWrite(i) & :- & GlobalSym(\texttt{"document"}, d), \\ & & \textsc{HeapPtsTo}(d, \texttt{"write"}, m), \\ & & \textsc{Calls}(i, h). \\ DocumentWrite(i) & :- & GlobalSym(\texttt{"document"}, d), \\ & & \textsc{HeapPtsTo}(d, \texttt{"writeln"}, m), \\ & & \textsc{Calls}(i, h). \end{array}$$

4.4 Redirecting the Browser

JavaScript in the browser has write access to the current page's location, which may be used to redirect the user to a malicious site. Google widget `Google_Calculator` performing such redirection is shown below:

```
window.location =
        "http://e-r.se/google-calculator/index.htm"
```

Allowing such redirect not only opens the door to phishing widgets luring users to malicious sites, redirects within an iframe also open the possibility of running code that has not been adequately checked by the hosting site, potentially circumventing policy checking entirely. Another concern is cross-site scripting attacks that involve stealing cookies: a cross-site scripting attack may be mounted by assigning a location of the form `"http://www.evil.com/" + document.cookie`. Of

Query output: $FrozenViolation(v : V)$

$BuiltInObject(h) :- GlobalSym(\texttt{"Boolean"}, h).$
$BuiltInObject(h) :- GlobalSym(\texttt{"Array"}, h).$
$BuiltInObject(h) :- GlobalSym(\texttt{"Date"}, h).$
$BuiltInObject(h) :- GlobalSym(\texttt{"Function"}, h).$
$BuiltInObject(h) :- GlobalSym(\texttt{"Math"}, h).$
$BuiltInObject(h) :- GlobalSym(\texttt{"Document"}, h).$
$BuiltInObject(h) :- GlobalSym(\texttt{"Window"}, h).$

$FrozenViolation(v) :- \textsc{Store}(v, \_, \_), \textsc{PtsTo}(v, h), BuiltInObject(h).$

Figure 12: FrozenViolations query

course, `grep` is not an adequate tool for spotting redirects, both because of the aliasing issue described above and because *read access* to `window.location` is in fact allowed. Moreover, redirects can take many forms, which we capture through the queries below. Direct `location` assignment are found by the following query:

Query output: $LocationAssign(v : V)$

$LocationAssign(v) :- GlobalSym(\texttt{"window"}, h), \textsc{PtsTo}(v, h), \textsc{Store}(\_, \texttt{"location"}, v).$

$LocationAssign(v) :- GlobalSym(\texttt{"document"}, h), \textsc{PtsTo}(v, h), \textsc{Store}(\_, \texttt{"location"}, v).$

$LocationAssign(v) :- \textsc{PtsTo}(\texttt{"global"}, h), \textsc{PtsTo}(v, h), \textsc{Store}(\_, \texttt{"location"}, v).$

Storing to location object's properties are found by the following query:

$LocationAssign(v) :- GlobalSym(h, \texttt{"location"}), \textsc{PtsTo}(v, h), \textsc{Store}(v, \_, \_).$

Calling methods on the location object are found by the following query:

Query output: $LocationChange(i : I)$

$LocationChange(i) :- LocationObject(h), \textsc{HeapPtsTo}(h, \texttt{"assign"}, h'), \textsc{Calls}(i, h').$

$LocationChange(i) :- LocationObject(h), \textsc{HeapPtsTo}(h, \texttt{"reload"}, h'), \textsc{Calls}(i, h').$

$LocationChange(i) :- LocationObject(h), \textsc{HeapPtsTo}(h, \texttt{"replace"}, h'), \textsc{Calls}(i, h').$

```
var SearchTag = new String ("Home");
var SearchTagStr = new String(
  "meta%3ASearch.tag%28%22beginTag+" +
  SearchTag +"endTag%22%29");
var QnaURL = new String(
  SearchHostPath /*+ SearchQstateStr */+
  SearchTagStr +"&format=rss") ;

// define the constructor for your Gadget
Microsoft.LiveQnA.RssGadget =
  function(p_elSource, p_args, p_namespace) { ... }
```

Figure 13: Example of a global namespace pollution violation (Section 4.5.2) in a Live widget.

Function `window.open` is another form of redirects, as the following query shows:

Query output: $WindowOpen(i : I)$

$WindowOpen(i) :- WindowObject(h), \textsc{HeapPtsTo}(h, \texttt{"open"}, h'), \textsc{Calls}(i, h').$

4.5 Host-specific Policies

The policies we have discussed thus far have been relatively generic. In this section, we give examples of policies that are specific to the host site they reside on.

4.5.1 No `XMLHttpRequest` Use in Live Widgets

The first policy of this sort comes directly from the Live Web Widget Developer Checklist [34]. Among other rules, they disallow the use of `XMLHttpRequest` object in favor of function `Web.Network.createRequest`. The latter makes sure that the network requests are properly proxied so they can work cross-domain:

Query output: $XMLHttpRequest(i : I)$

$XMLHttpRequest(i) :- GlobalSym(\texttt{"XMLHttpRequest"}, h), \textsc{Calls}(i, h).$

Query output: $ActiveXExecute(i : I)$		
$ActiveXObjectCalls(i)$	:–	$GlobalSym(\texttt{"ActiveXObject"}, h'), \textsc{Calls}(i, h').$
$ShellExecuteCalls(i)$	:–	$\textsc{PtsTo}(\texttt{"global"}, h_1), \textsc{HeapPtsTo}(h_1, \texttt{"System"}, h_2),$ $\textsc{HeapPtsTo}(h_2, \texttt{"Shell"}, h_3), \textsc{HeapPtsTo}(h_3, \texttt{"execute"}, h_4), \textsc{Calls}(i, h_4).$
$ActiveXExecute(i)$	:–	$ActiveXObjectCalls(i), \textsc{CallRet}(i, v), \textsc{PtsTo}(v, h),$ $\textsc{HeapPtsTo}(h, \_, m), \textsc{Calls}(i^\star, m), \textsc{CallRet}(i^\star, r), \textsc{PtsTo}(r, h^\star),$ $ShellExecuteCalls(i'), \textsc{Actual}(i', \_, v'), \textsc{PtsTo}(v', h^\star).$

Figure 14: Query for finding information flow violations in Vista Sidebar widgets.

4.5.2 Global Namespace Pollution in Live Widgets

Because web widgets can be deployed on a page with other widgets running within the same JavaScript interpreter, polluting the global namespace, leading to name clashes and unpredictable behavior. This is why hosting providers such as Facebook, Yahoo!, Live, etc. strongly discourage pollution of the global namespace, favoring a module or a namespace approach instead [11] that avoids name collision. We can easily prevent stores to the global scope:

Query output: $GlobalStore(h : H)$		
$GlobalStore(h)$	:–	$\textsc{PtsTo}(\texttt{"global"}, g),$ $\textsc{HeapPtsTo}(g, \_, h).$

An example of a violation of this policy from a Live.com widget is shown in Figure 13. Because the same widget can be deployed twice within the same interpreter scope with different values of `SearchTag`, this can lead to a data race on the globally declared variable `SearchTagStr`.

Note that our analysis approach is radically different from proposals that advocate language restrictions such as AdSafe or Cajita [12, 13, 29] to protect access to the global object. The difficulty those techniques have to overcome is that the `this` identifier in the global scope will point to the global object. However, disallowing `this` completely makes object-oriented programming difficult. With the whole-program analysis GATEKEEPER implements, we do not have this problem. We are able to distinguish references to `this` that point to the global object (aliased with the `global` variable) from a local reference to `this` within a function.

4.5.3 Tainting Data in Sidebar Widgets

This policy ensures that data from ActiveX controls that may be instantiated by a Sidebar widget does not get passed into System.Shell.execute for direct execution on the user's machine. This is because it is common for ActiveX controls to retrieve unsanitized network data, which is how a published RSS Sidebar exploit operates [27]. There, data obtained from an ActiveX-based RSS control was assigned directly to the `innerHTML` field withing a widget, allowing a cross-site scripting exploit. What we are looking for is demonstrated by the pattern:

```
var o = new ActiveXObject();
var x = o.m();
System.Shell.Execute(x);
```

The Datalog query in Figure 14 looks for instances where the tainted result of a call to method `m` on an ActiveX object is directly passed as an argument to the "sink" function `System.Shell.Execute`.

Auxiliary queries $ActiveXObjectCalls$ and $ShellExecuteCalls$ look for source and sink calls and $ShellExecuteCalls$ ties all the constraints together, effectively matching the call pattern described above. As previously shown for the case of Java information flow [23], similar queries may be used to find information flow violations that involve cookie stealing and location resetting, as described in Chugh et al. [10].

5 Experimental Results

For our experiments, we have downloaded a large number of widgets from widget hosting sites' widget galleries. As mentioned before, we have experimented with widgets from Live.com, the Vista Sidebar, and Google. We automated the download process to save widgets locally for analysis. Once downloaded, we parsed through each widget's manifesto to determine where the relevant JavaScript code resides. This process was slightly different across the widget hosts. In particular, Google widgets tended to embed their JavaScript in HTML, which required us to develop a limited-purpose HTML parser. In the Sidebar case, we had to extract the relevant JavaScript code out of an archive. At the end of this process, we ended up with a total of 8,379 JavaScript files to analyze.

Figure 15 provides aggregate statistics for the widgets we used as benchmarks. For each widget source,

Widget Source	Avg. LOC	Count	Widget counts JavaScript$_{GK}$		JavaScript$_{SAFE}$	
Live.com	105	2,707	2,643	97%	643	23%
Vista sidebar	261	4,501	2,946	65%	1,767	39%
Google.com/ig	137	1,171	962	82%	768	65%

Figure 15: Aggregate statistics for widgets from Live portal, Windows Sidebar, and Google portal widget repositories (columns 2–3). Information about widget distribution for different JavaScript language subsets (columns 4–7).

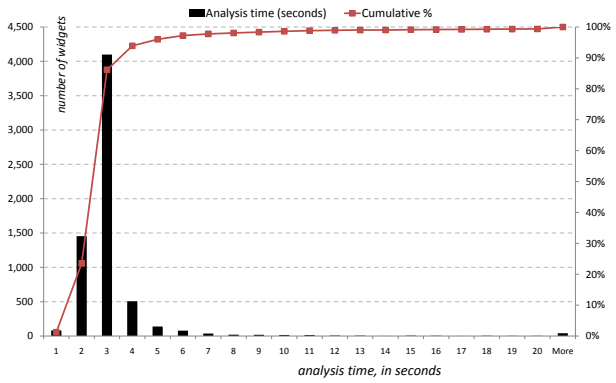

Figure 17: Histogram showing GATEKEEPER processing times.

we specify the total number of widgets we managed to obtain in column 2. Column 3 shows the average lines-of-code count for every widget. In general, Sidebar widgets tend to be longer and more involved than their Web counterparts, as reflected in the average line of code metric. Note that in addition to every widget's code, at the time of policy checking, we also prepend the native environment constructed as described in Section 3.4. The native environment constitutes 270 lines of non-comment JavaScript code (127 for specifying the the browser embedding and 143 for specifying built-in objects such as `Array` and `Date`).

5.1 Result Summary

A summary of our experimental results in presented in Figure 16. For each policy described in Section 4, we show the the total number of violations across 8,379 benchmarks, and the number of violating benchmarks. The latter two may be different because there could be several violations of a particular query per widget. We also show the percentage of benchmarks for which we find policy violations. As can be seen from the table, overall, policy violations are quite uncommon, with only several percent of widgets affected in each case. Overall, a total of 1,341 policy violations are reported.

As explained in Section 4.5, we only ran those policies on the appropriate subset of widgets, leaving other table cells blank. To validate the precision of our analysis, we have examined all violations reported by our policies. For examination, GATEKEEPER output was cross-referenced with widget sources. Luckily for us, most of our query results were easy to spot-check by looking at one or two lines of corresponding source code, which made result checking a relatively quick task. Encouragingly, for most inputs, GATEKEEPER was quite precise.

```
function MM_preloadImages() {
  var d=m_Doc;
  if(d.images){
    if(!d.MM_p) d.MM_p=new Array();
    var i,j=d.MM_p.length,
      a=MM_preloadImages.arguments;
    for(i=0; i<a.length; i++)
      if (a[i].indexOf("#")!=0){
        d.MM_p[j]=new Image;
        d.MM_p[j++].src=a[i];
      }
  }
}
```

Figure 18: False positives in `common.js` from `JustMusic.FM`.

5.2 False Positives

We should point out that a conservative analysis such as GATEKEEPER is inherently imprecise. Two main sources of false positives in our formulation are prototype handling and arrays. Only two widgets out of over 6,000 analyzed files in the JavaScript$_{GK}$ subset lead to false positives in our experiments. Almost all false positive reports come from the Sidebar widget, `JustMusic.FM`, file `common.js`. Because of our handling of arrays, the analysis conservatively concludes that certain heap-allocated objects can reach many others by following *any* element of array `a`, as shown in Figure 18. In fact, this example is contains a number of features that are difficult to analyze statically: array aliasing, the use of `arguments` array, as well as array element loads and stores, so it is not entirely surprising that their combination leads to imprecision.

It is common for a single imprecision within static analysis to create numerous "cascading" false positive reports. This is the case here as well. Luckily, it is possible to group cascading reports together in order to avoid overwhelming the user with false positives caused by a single imprecision. This imprecision in turn affects $FrozenViolation$ and $LocationAssign$ queries leading to many very similar reports. A total of 113 false positives are reported, but luckily they affect only two widgets.

5.3 Analysis Running Times

Our implementation uses a publicly available declarative analysis engine provided by bddbddb [32]. This is a

		Live Widgets					Vista Sidebar					Google Widgets				
Query	**Section**	**Viol.**	**Affected**	**%**	**FP**	**Affected**	**Viol.**	**Affected**	**%**	**FP**	**Affected**	**Viol.**	**Affected**	**%**	**FP**	**Affected**
$AlertCalls(i : I)$	4.1	54	29	1.1	0	0	161	84	2.9	0	0	57	35	3.6	0	0
$FrozenViolation(v : V)$	4.2	3	3	0.1	0	0	143	52	1.5	94	1	1	1	0.1	0	0
$DocumentWrite(i : I)$	4.3	5	1	0.0	0	0	175	75	1.7	0	0	158	88	8.1	0	0
$LocationAssign(v : V)$	4.4	3	3	0.1	2	1	157	109	3.8	15	1	9	9	0.7	0	0
$LocationChange(i : I)$	4.4	3	3	0.1	0	0	21	20	0.7	1	1	3	3	0.3	0	0
$WindowOpen(i : I)$	4.4	50	22	0.9	0	0	182	87	3.0	1	1	19	14	1.5	0	0
$XMLHttpRequest(i : I)$	4.5	1	1	0.0	0	0	—	—	—	—	—	—	—	—	—	—
$GlobalStore(v : V)$	4.5	136	45	1.7	0	0	—	—	—	—	—	—	—	—	—	—
$ActiveXExecute(i : I)$	4.5	—	—	—	—	—	0	0	0	0	0	—	—	—	—	—

Figure 16: Experimental result summary for nine policies described in Section 4. Because some policies are host-specific, we only run them on a subset of widgets. "—" indicates experiments that are not applicable.

	Live	**Sidebar**	**Google**
Number of instrumented files	2,000	1,179	194
Instrumentation points per file	1.74	8.86	5.63
Estimated overhead	40%	65%	73%

Figure 19: Instrumentation statistics.

highly optimized BDD-based solver for Datalog queries used for static analysis in the past. Because repeatedly starting bddbddb is inefficient we perform both the points-to analysis *and* run our Datalog queries corresponding to the policies in Section 4 as part of one run for each widget.

Our analysis is quite scalable in practice, as shown in Figure 17. This histogram shows the distribution of analysis time, in seconds. These results were obtained on a Pentium Core 2 duo 3 GHz machine with 4 GB of memory, running Microsoft Vista SP1. Note that the analysis time includes the JavaScript parsing time, the normalization time, the points-to analysis time, and the time to run all nine policies. For the vast majority of widgets, the analysis time is under 4 seconds, as shown by the cumulative percentage curve in the figure. The bddbddb-based approach has been shown to scale to much larger programs — up to 500,000 lines of code — in the past [32], so we are confident that we should be able to scale to larger codebases in GATEKEEPER as well.

5.4 Runtime Instrumentation

Programs outside of the $\text{JavaScript}_{\text{SAFE}}$ language subset but within the $\text{JavaScript}_{\text{GK}}$ language subset require instrumentation. Figure 19 summarizes data on the number of instrumentation points required, both as an absolute number and in proportion of the number of widgets that required instrumentation.

We plan to fully assess our runtime overhead as part of future work. However, we do not anticipate it to be prohibitively high. The number of instrumentation points per instrumented widget ranges roughly in proportion to the size and complexity of the widget. However, it is generally difficult to perform large-scale overhead measurements for a number of highly interactive widgets.

Instead we have devised an experiment to approximate the overheads. Note that we can discern the average density of checks from the numbers in Figure 19: for instance, for Live.com, the number of instrumentation points per file is 1.74, with an average file being 105 lines, as shown in Figure 15. This yields about 2% of all lines being instrumentated, on average.

To mimic this runtime check density, we generate a test script shown in Figure 20 with 100 fields stores, where the first two stores require runtime checking and the other 98 are statically known. For Sidebar and Google widgets, we construct similar test scripts with a different density of checks. As shown below, we use index `innerHTML` for one out of two rewritten cases for Live. We use it for 2 out of 3 cases for Sidebase, and 2 out of 4 cases for Google. This represents a pretty high frequency of `innerHTML` assignments.

We wrap this code in a loop that we run 1,000 times to be able to measure the overheads reliably and then take the median over several runs to account for noise. The baseline is the same test with no index or right-hand side checks. We observe overheads ranging between 40–73% across the different instrumentation densities, as shown in Figure 19. It appears that calls to `toStaticHTML` result in a pretty substantial runtime penalty. This is likely because the relatively heavy-weight HTML parser of the browser needs to be invoked on every HTML snippet.

Note that this experiment provides an approximate measure of overhead that real programs are likely to experience. However, these numbers are encouraging, as they are significantly smaller overheads on the order of 6–40x that tools like Caja may induce [28].

```
console.log(new Date().getTime());
var v1 = new Array();
var v2 = "<div onclick='alert(38);'>" +
    "<h2>Hello<script>alert(38)</script></div>";
for(var iter = 0; iter < 1000; iter++){
    // first store: check
    var i = 'innerHTML';
    if (i==="onclick" ||  i==="onkeypress" || ...) {
        alert("Disguised eval at <file>:<line>");
    } else
    if(i==="innerHTML" && __IsUnsafe(v2)){
        alert("Unsafe innerHTML at <file>:<line>");
    } else {
        v1[i] = v2;
    }

    // second store: check
    i = 'onclick';
    if (i==="onclick" ||  i==="onkeypress" || ...) {
        alert("Disguised eval at <file>:<line>");
    } else
    if(i==="innerHTML" && __IsUnsafe(v2)){
        alert("Unsafe innerHTML at <file>:<line>");
    } else {
        v1[i] = v2;
    }

    // all other stores are unchecked
    v1[i] = 2;
    v1[i] = 3;
    ...
    v1[i] = 100;
}
console.log(new Date().getTime());
```

Figure 20: Measuring the overhead of GATEKEEPER checking.

6 Related Work

Much of the work related to this paper focuses on limiting various attack vectors that exist in JavaScript. They do this through the use of type systems, language restrictions, and modifications to the browser or the runtime. We describe these strategies in turn below.

6.1 Static Safety Checks

JavaScript is a highly dynamic language which makes it difficult to reason about programs written in it. However, with certain expressiveness restrictions, desirable security properties can be achieved. ADSafe and Facebook both implement a form of static checking to ensure a form of safety in JavaScript code. ADSafe [13] disallows dynamic content, such as `eval`, and performs static checking to ensure the JavaScript in question is safe. Facebook takes an approach similar to ours in rewriting statically unresolved field stores, however, it appears that, unlike GATEKEEPER, they do not try to do local static analysis of field names. Facebook uses a JavaScript language variant called FBJS [15], that is like JavaScript in many ways, but DOM access is restricted and all variable names are prefixed with a unique identifier to prevent name clashes with other FBJS programs on the same page.

In many ways, however, designing a safe language subset is a tricky business. Until recently, is was difficult to write anything but most simple applications in AdSafe because of its static restrictions, at least in our personal experience. More recently, AdSafe was updated with APIs to lift some of initial restrictions and allow DOM access, etc., as well as several illustrative sample widgets. Overall, these changes to allow compelling widgets to be written are an encouraging sign. While quite expressive, FBJS has been the subject of several well-publicised attacks that circumvent the isolation of the global object offered through Facebook sandbox rewriting [2]. This demonstrates that while easy to implement, reasoning about what static language restrictions accomplish is tricky.

GATEKEEPER largely sidesteps the problem of proper language subset design, opting for whole program analysis instead. We do no try to prove that JavaScript$_{\text{SAFE}}$ programs cannot pollute the global namespace for *all* programs, for example. Instead, we take the entire program and a representation of its environment and use our static analysis machinery to check if this may happen for the input program in question. The use of static and points-to analysis for finding and vulnerabilities and ensuring security properties has been previously explored for other languages such as C [6] and Java [23].

An interesting recent development in JavaScript language standards committees is the strict mode (`use strict`) for JavaScript [14], page 223, which is being proposed around the time of this writing. Strict mode accomplishes many of the goals that JavaScript$_{\text{SAFE}}$ is designed to accomplish: `eval` is largely prohibited, bad coding practices such as assigning to the `arguments` array are prevented, `with` is no longer allowed, etc. Since the strict mode supports customization capabilities, going forward we hope to be able to express JavaScript$_{\text{SAFE}}$ and JavaScript$_{\text{GK}}$ restrictions in a standards-compliant way, so that future off-the-shelf JavaScript interpreters would be able to enforce them.

6.2 Rewriting and Instrumentation

A practical alternative to static language restrictions is instrumentation. Caja [29] is one such attempt at limiting capabilities of JavaScript programs and enforcing this through the use of runtime checks. WebSandbox is another project with similar goals that also attempts to enforce reliability and resource restrictions in addition to security properties [25].

Yu et al. traverse the JavaScript document and rewrite based on a security policy [35]. Unlike Caja and WebSandbox, they prove the correctness of their rewriting

with operational semantics for a subset of JavaScript called CoreScript. BrowserShield [30] similarly uses dynamic and recursive rewriting to ensure that JavaScript and HTML are safe, for a chosen version of safety, and all content generated by the JavaScript and HTML is also safe. Instrumentation can be used for more than just enforcing security policies. AjaxScope [20] rewrites JavaScript to insert instrumentation that sends runtime information, such as error reporting and memory leak detection, back to the content provider.

Compared to these techniques, GATEKEEPER has two main advantages. First, as a mostly static analysis, GATEKEEPER places little runtime overhead burden on the user. While we are not aware of a comprehensive overhead evaluation that has been published, it appears that the runtime overhead of Caja and WebSandbox may be high, depending on the level of rewriting. For instance, a Caja authors' report suggest that the overhead of various subsets that are part of Caja are 6–40x [28]. Second, as evidenced by the Facebook exploits mentioned above [2], it is challenging to reason about whether source-level rewriting provides complete isolation. We feel that sound static analysis may provide a more systematic way to reason about what code can do, especially in the long run, as it pertains to issues of security, reliability, and performance. While the soundness of the native environment and exhaustiveness of our runtime checks might be weak points of our approach, we feel that we can address these challenges as part of future work.

6.3 Runtime and Browser Support

Current browser infrastructure and the HTML standard require a page to fully trust foreign JavaScript if they want the foreign JavaScript to interact with their site. The alternative is to place foreign JavaScript in an isolated environment, which disallows any interaction with the hosting page. This leads to web sites trusting untrustworthy JavaScript code in order to provide a richer web site. One solution to get around this all-or-nothing trust problem is to modify browsers and the HTML standard to include a richer security model that allows untrusted JavaScript controlled access to the hosting page.

MashupOS [18] proposes a new browser that is modeled after an OS and modifies the HTML standard to provide new tags that make use of new browser functionality. They provide rich isolation between execution environments, including resource sharing and communication across instances. In a more lightweight modification to the browser and HTML, Felt et al. [16] add a new HTML tag that labels a `div` element as untrusted and limits the actions that any JavaScript inside of it can take. This would allow content providers to create a sand box in which to place untrusted JavaScript. Integrating GATEKEEPER techniques into the browser itself, without relying on server-side analysis, and making them fast enough for daily use, is part of future work.

6.4 Typing and Analysis of JavaScript

A more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [8] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety.

Other work has been done to devise a static type system that describes the JavaScript language [4, 5, 31]. These works focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subests of JavaScript. As far as we can tell, none of these approaches have been applied to realistic bodies of code. GATEKEEPER uses a pointer analysis to reason about the JavaScript program in contrast to the type systems and analyses of these works. We feel that the ability to reason about pointers and the program call graph allows us to express more interesting security policies than we would be able otherwise.

A contemporaneous project by Chugh et al. focuses on staged analysis of JavaScript and finding information flow violations in client-side code [10]. Chugh et al. focus on information flow properties such as reading document cookies and changing the locations, not unlike the location policy described in Section 4.4. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis.

7 Conclusions

This paper presents GATEKEEPER, a mostly static sound policy enforcement tool for JavaScript programs. GATEKEEPER is built on top of what to our knowledge is the first pointer analysis developed for JavaScript. To show the practicality of our approach, we describe nine representative security and reliability policies for JavaScript widgets. Statically checking these policies results in 1,341 verified warnings in 684 widgets, with 113 false positives affecting only two widgets.

We feel that static analysis of JavaScript is a key building block for enabling an environment in which code from different parties can safely co-exist and interact. The ability to analyze a programming language using automatic tools is a valuable one for long-term language success.

It is therefore our hope that our experience with analyzable JavaScript language subsets will inform the design of language restrictions build into future versions of the JavaScript language, as illustrated by the JavaScript `use strict` mode.

While in this paper our focus is on policy enforcement, the techniques outlines here are generally useful for any task that involves reasoning about code such as code optimization, rewriting, program understanding tools, bug finding tools, etc. Moreover, we hope that GATEKEEPER paves the way for centrally-hosted software repositories such as the iPhone application store, Windows Marketplace, or Android Market to ensure the security and quality of software contributed by third parties.

Acknowledgments

We are grateful to Trishul Chilimbi, David Evans, Karthik Pattabiraman, Nikhil Swamy, and the anonymous reviewers for their feedback on this paper. We appreciate John Whaley's help with bddbddb.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] Ajaxian. Facebook JavaScript and security. `http://ajaxian.com/archives/facebook-javascript-and-security`, Aug. 2007.

[3] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, University of Copenhagen, 1994.

[4] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD 04, volume WOOD of ENTCS. Elsevier, 2004. http://www.binarylord.com/ work/js0wood.pdf*, 2004.

[5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *In Proceedings of the European Conference on Object-Oriented Programming*, pages 429–452, July 2005.

[6] D. Avots, M. Dalton, B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the International Conference on Software Engineering*, pages 332–341, May 2005.

[7] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *European Conference on Computer Systems*, pages 73–85, 2006.

[8] R. Cartwright and M. Fagan. Soft typing. *ACM SIGPLAN Notices*, 39(4):412–428, 2004.

[9] B. Chess, Y. T. O'Neil, and J. West. JavaScript hijacking. `www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, Mar. 2007.

[10] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[11] D. Crockford. Globals are evil. `http://yuiblog.com/blog/2006/06/01/global-domination/`, June 2006.

[12] D. Crockford. *JavaScript: the good parts*. 2008.

[13] D. Crockford. AdSafe: Making JavaScript safe for advertising. `http://www.adsafe.org`, 2009.

[14] ECMA. Ecma-262: Ecma/tc39/2009/025, 5th edition, final draft. `http://www.ecma-international.org/publications/files/drafts/tc39-2009-025.pdf`, Apr. 2009.

[15] Facebook, Inc. Fbjs. `http://wiki.developers.facebook.com/index.php/FBJS`, 2007.

[16] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Proceedings of the Workshop on Social Network Systems*, pages 25–30, 2008.

[17] Finjan Inc. Web security trends report. `http://www.finjan.com/GetObject.aspx?ObjId=506`.

[18] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[19] `javascript-reference.info`. JavaScript obfuscators review. `http://javascript-reference.info/javascript-obfuscators-review.htm`, 2008.

[20] E. Kıcıman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[21] M. S. Lam, J. Whaley, B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Symposium on Principles of Database Systems*, June 2005.

[22] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. Technical Report MSR-TR-2009-43, Microsoft Research, Feb. 2009.

[23] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2005.

[24] Microsoft Corporation. Static driver verifier. `http://www.microsoft.com/whdc/devtools/tools/SDV.mspx`, 2005.

[25] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`, 2008.

[26] Microsoft Live Labs. Quality of service (QoS) protections. `http://websandbox.livelabs.com/documentation/use_qos.aspx`, 2008.

[27] Microsoft Security Bulletin. Vulnerabilities in Windows gadgets could allow remote code execution (938123). `http://www.microsoft.com/technet/security/Bulletin/MS07-048.mspx`, 2007.

[28] M. S. Miller. Is it possible to mix ExtJS and google-caja to enhance security. `http://extjs.com/forum/showthread.php?p=268731#post268731`, Jan. 2009.

[29] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-2007.pdf`, 2007.

[30] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

[31] P. Thiemann. Towards a type system for analyzing JavaScript programs. 2005.

[32] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2005.

[33] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 131–144, June 2004.

[34] Windows Live. Windows live gadget developer checklist. `http://dev.live.com/gadgets/sdk/docs/checklist.htm`, 2008.

[35] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of Conference on Principles of Programming Languages*, Jan. 2007.

NOZZLE: A Defense Against Heap-spraying Code Injection Attacks

Paruj Ratanaworabhan
Cornell University
paruj@csl.cornell.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Benjamin Zorn
Microsoft Research
zorn@microsoft.com

Abstract

Heap spraying is a security attack that increases the exploitability of memory corruption errors in type-unsafe applications. In a heap-spraying attack, an attacker coerces an application to allocate many objects containing malicious code in the heap, increasing the success rate of an exploit that jumps to a location within the heap. Because heap layout randomization necessitates new forms of attack, spraying has been used in many recent security exploits. Spraying is especially effective in web browsers, where the attacker can easily allocate the malicious objects using JavaScript embedded in a web page. In this paper, we describe NOZZLE, a runtime heap-spraying detector. NOZZLE examines individual objects in the heap, interpreting them as code and performing a static analysis on that code to detect malicious intent. To reduce false positives, we aggregate measurements across all heap objects and define a global heap health metric.

We measure the effectiveness of NOZZLE by demonstrating that it successfully detects 12 published and 2,000 synthetically generated heap-spraying exploits. We also show that even with a detection threshold set six times lower than is required to detect published malicious attacks, NOZZLE reports no false positives when run over 150 popular Internet sites. Using sampling and concurrent scanning to reduce overhead, we show that the performance overhead of NOZZLE is less than 7% on average. While NOZZLE currently targets heap-based spraying attacks, its techniques can be applied to any attack that attempts to fill the address space with malicious code objects (e.g., stack spraying [42]).

1 Introduction

In recent years, security improvements have made it increasingly difficult for attackers to compromise systems. Successful prevention measures in runtime environments and operating systems include stack protection [10], improved heap allocation layouts [7, 20], address space layout randomization [8, 36], and data execution prevention [21]. As a result, attacks that focus on exploiting memory corruptions in the heap are now popular [28].

Heap spraying, first described in 2004 by SkyLined [38], is an attack that allocates many objects containing the attacker's exploit code in an application's heap. Heap spraying is a vehicle for many high profile attacks, including a much publicized exploit in Internet Explorer in December 2008 [23] and a 2009 exploit of Adobe Reader using JavaScript embedded in malicious PDF documents [26].

Heap spraying requires that an attacker use another security exploit to trigger an attack, but the act of spraying greatly simplifies the attack and increases its likelihood of success because the exact addresses of objects in the heap do not need to be known. To perform heap spraying, attackers have to be able to allocate objects whose contents they control in an application's heap. The most common method used by attackers to achieve this goal is to target an application, such as a web browser, which executes an interpreter as part of its operation. By providing a web page with embedded JavaScript, an attacker can induce the interpreter to allocate their objects, allowing the spraying to occur. While this form of spraying attack is the most common, and the one we specifically consider in this paper, the techniques we describe apply to all forms of heap spraying. A number of variants of spraying attacks have recently been proposed including sprays involving compiled bytecode, ANI cursors [22], and thread stacks [42].

In this paper, we describe NOZZLE, a detector of heap spraying attacks that monitors heap activity and reports spraying attempts as they occur. To detect heap spraying attacks, NOZZLE has two complementary components. First, NOZZLE scans individual objects looking for signs of malicious intent. Malicious code commonly includes a landing pad of instructions (a so-called NOP sled) whose execution will lead to dangerous shellcode. NOZZLE focuses on detecting a sled through an analysis of its control flow. We show that prior work on sled detection [4, 16, 31, 43] has a high false positive rate when applied to objects in heap-spraying attacks (partly due to

the opcode density of the x86 instruction set). NOZZLE interprets individual objects as code and performs a static analysis, going beyond prior sled detection work by reasoning about code reachability. We define an attack surface metric that approximately answers the question: "If I were to jump randomly into this object (or heap), what is the likelihood that I would end up executing shellcode?"

In addition to local object detection, NOZZLE aggregates information about malicious objects across the entire heap, taking advantage of the fact that heap spraying requires large-scale changes to the contents of the heap. We develop a general notion of global "heap health" based on the measured attack surface of the applicatoin heap contents, and use this metric to reduce NOZZLE's false positive rates.

Because NOZZLE only examines object contents and requires no changes to the object or heap structure, it can easily be integrated into both native and garbage-collected heaps. In this paper, we implement NOZZLE by intercepting calls to the memory manager in the Mozilla Firefox browser (version 2.0.0.16). Because browsers are the most popular target of heap spray attacks, it is crucial for a successful spray detector to both provide high successful detection rates and low false positive rates. While the focus of this paper is on low-overhead online detection of heap spraying, NOZZLE can be easily used for offline scanning to find malicious sites in the wild [45]. For offline scanning, we can combine our spraying detector with other checkers such as those that match signatures against the exploit code, etc.

1.1 Contributions

This paper makes the following contributions:

- We propose the first effective technique for detecting heap-spraying attacks through runtime interpretation and static analysis. We introduce the concept of attack surface area for both individual objects and the entire heap. Because directing program control to shellcode is a fundamental property of NOP sleds, the attacker cannot hide that intent from our analysis.
- We show that existing published sled detection techniques [4, 16, 31, 43] have high false positive rates when applied to heap objects. We describe new techniques that dramatically lower the false positive rate in this context.
- We measure Firefox interacting with popular web sites and published heap-spraying attacks, we show that NOZZLE successfully detects 100% of 12 published and 2,000 synthetically generated heap-spraying exploits. We also show that even with a detection threshold set six times lower than is required to detect known malicious attacks, NOZZLE reports no false positives when tested on 150 popular Alexa.com sites.
- We measure the overhead of NOZZLE, showing that without sampling, examining every heap object slows execution 2–14 times. Using sampling and concurrent scanning, we show that the performance overhead of NOZZLE is less than 7% on average.
- We provide the results of applying NOZZLE to Adobe Reader to prevent a recent heap spraying exploit embedded in PDF documents. NOZZLE succeeds at stopping this attack without any modifications, with a runtime overhead of 8%.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on heap spraying attacks. Section 3 provides an overview of NOZZLE and Section 4 goes into the technical details of our implementation. Section 5 summarizes our experimental results. While NOZZLE is the first published heap spraying detection technique, our approach has several limitations, which we describe fully in Section 6. Finally, Section 7 describes related work and Section 8 concludes.

2 Background

Heap spraying has much in common with existing stack and heap-based code injection attacks. In particular, the attacker attempts to inject code somewhere in the address space of the target program, and through a memory corruption exploit, coerce the program to jump to that code. Because the success of stack-based exploits has been reduced by the introduction of numerous security measures, heap-based attacks are now common. Injecting and exploiting code in the heap is more difficult for an attacker than placing code on the stack because the addresses of heap objects are less predictable than those of stack objects. Techniques such as address space layout randomization [8, 36] further reduce the predictability of objects on the heap. Attackers have adopted several strategies for overcoming this uncertainty [41], with heap spraying the most successful approach.

Figure 1 illustrates a common method of implementing a heap-spraying attack. Heap spraying requires a memory corruption exploit, as in our example, where an attacker has corrupted a vtable method pointer to point to an incorrect address of their choosing. At the same time, we assume that the attacker has been able, through entirely legal methods, to allocate objects with contents of their choosing on the heap. Heap spraying relies on populating the heap with a large number of objects containing the attacker's code, assigning the vtable exploit to jump to an

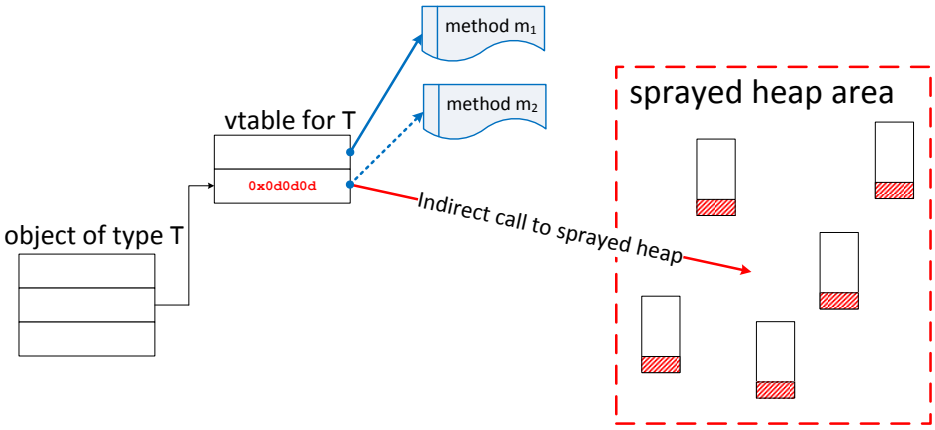

Figure 1: Schematic of a heap spraying attack.

```
 1. <SCRIPT language="text/javascript">
 2.   shellcode = unescape("%u4343%u4343%...");
 3.   oneblock = unescape("%u0D0D%u0D0D");
 4.
 5.   var fullblock = oneblock;
 6.   while (fullblock.length<0x40000) {
 7.       fullblock += fullblock;
 8.   }
 9.
10. sprayContainer = new Array();
11. for (i=0; i<1000; i++) {
12.     sprayContainer[i] = fullblock + shellcode;
13. }
14. </SCRIPT>
```

Figure 2: A typical JavaScript heap spray.

arbitrary address in the heap, and relying on luck that the jump will land inside one of their objects. To increase the likelihood that the attack will succeed, attackers usually structure their objects to contain an initial NOP sled (indicated in white) followed by the code that implements the exploit (commonly referred to as shellcode, indicated with shading). Any jump that lands in the NOP sled will eventually transfer control to the shellcode. Increasing the size of the NOP sled and the number of sprayed objects increases the probability that the attack will be successful.

Heap spraying requires that the attacker control the contents of the heap in the process they are attacking. There are numerous ways to accomplish this goal, including providing data (such as a document or image) that when read into memory creates objects with the desired properties. An easier approach is to take advantage of scripting languages to allocate these objects directly. Browsers are particularly vulnerable to heap spraying because JavaScript embedded in a web page authored by the attacker greatly simplifies such attacks.

The example shown in Figure 2 is modelled after a previously published heap-spraying exploit [44]. While we are only showing the JavaScript portion of the page, this payload would be typically embedded within an HTML page on the web. Once a victim visits the page, the JavaScript payload is automatically executed. Lines 2 allocates the shellcode into a string, while lines 3–8 of the JavaScript code are responsible for setting up the spraying NOP sled. Lines 10–13 create JavaScript objects each of which is the result of combining the sled with the shellcode. It is quite typical for published exploits to contain a long sled (256 KB in this case). Similarly, to increase the effectiveness of the attack, a large number of JavaScript objects are allocated on the heap, 1,000 in this case. Figure 10 in Section 5 provides more information on previously published exploits.

3 Overview

While type-safe languages such as Java, C#, and JavaScript reduce the opportunity for malicious attacks, heap-spraying attacks demonstrate that even a type-safe program can be manipulated to an attacker's advantage. Unfortunately, traditional signature-based pattern matching approaches used in the intrusion detection literature are not very effective when applied to detecting heap-spraying attacks. This is because in a language as flexible as JavaScript it is easy to hide the attack code by either using encodings or making it polymorphic; in fact, most JavaScript worms observed in the wild use some form of encoding to disguise themselves [19, 34]. As a result, effective detection techniques typically are not syntactic. They are performed at runtime and employ some level of semantic analysis or runtime interpretation. Hardware support has even been provided to address this problem, with widely used architectures supporting a "no-execute bit", which prevents a process from executing code on specific pages in its address space [21]. We dis-

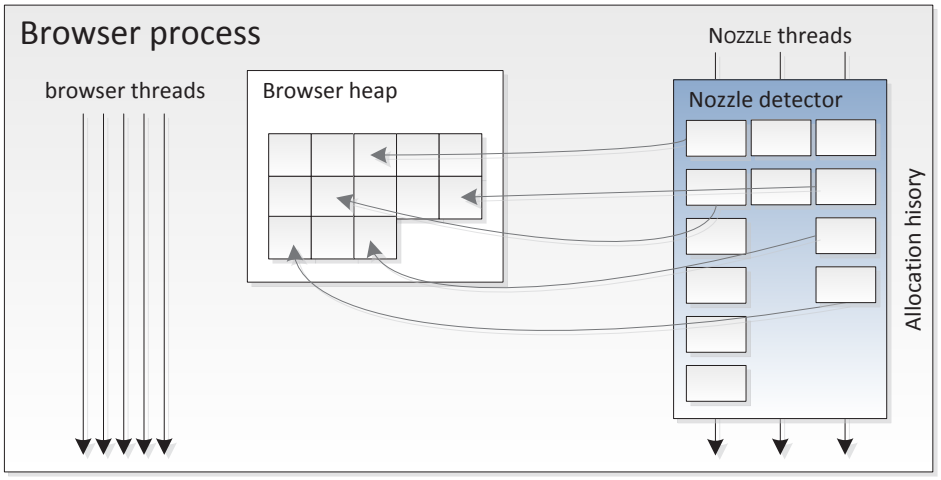

Figure 3: NOZZLE system architecture.

cuss how NOZZLE complements existing hardware solutions in Section 7. In this paper, we consider systems that use the x86 instruction set architecture (ISA) running the Windows operating system, a ubiquitous platform that is a popular target for attackers.

3.1 Lightweight Interpretation

Unlike previous security attacks, a successful heap-spraying attack has the property that the attack influences the contents of a large fraction of the heap. We propose a two-level approach to detecting such attacks: scanning objects locally while at the same time maintaining heap health metrics globally.

At the individual object level, NOZZLE performs lightweight interpretation of heap-allocated objects, treating them as though they were code. This allows us to recognize potentially unsafe code by interpreting it within a safe environment, looking for malicious intent.

The NOZZLE lightweight emulator scans heap objects to identify valid x86 code sequences, disassembling the code and building a control flow graph [35]. Our analysis focuses on detecting the NOP sled, which is somewhat of a misnomer. The sled can be composed of arbitrary instructions (not just NOPs) as long as the effect they have on registers, memory, and the rest of the machine state do not terminate execution or interfere with the actions of the shellcode. Because the code in the sled is intended to be the target of a misdirected jump, and thus has to be executable, the attacker cannot hide the sled with encryption or any means that would prevent the code from executing. In our analysis, we exploit the fundamental nature of the sled, which is to direct control flow specifically to the shellcode, and use this property as a means of detecting it. Furthermore, our method does not require detecting or assume there exists a definite partition between the shellcode and the NOP sled.

Because the attack jump target cannot be precisely controlled, the emulator follows control flow to identify basic blocks that are likely to be reached through jumps from multiple offsets into the object. Our local detection process has elements in common with published methods for sled detection in network packet processing [4, 16, 31, 43]. Unfortunately, the density of the x86 instruction set makes the contents of many objects look like executable code, and as a result, published methods lead to high false positive rates, as demonstrated in Section 5.1.

We have developed a novel approach to mitigate this problem using global heap health metrics, which effectively distinguishes benign allocation behavior from malicious attacks. Fortunately, an inherent property of heap-spraying attacks is that such attacks affect the heap globally. Consequently, NOZZLE exploits this property to drastically reduce the false positive rate.

3.2 Threat Model

We assume that the attacker has access to memory vulnerabilities for commonly used browsers and also can lure users to a web site whose content they control. This provides a delivery mechanism for heap spraying exploits. We assume that the attacker does not have further access to the victim's machine and the machine is otherwise uncompromised. However, the attacker does *not* control the precise location of any heap object.

We also assume that the attacker knows about the NOZZLE techniques and will try to avoid detection. They may have access to the browser code and possess detailed knowledge of system-specific memory layout properties

such as object alignment. There are specific potential weaknesses that NOZZLE has due to the nature of its runtime, statistical approach. These include time-of-check to time-of-use vulnerabilities, the ability of the attacker to target their attack under NOZZLE's thresholds, and the approach of inserting junk bytes at the start of objects to avoid detection. We consider these vulnerabilities carefully in Section 6, after we have presented our solution in detail.

4 Design and Implementation

In this section, we formalize the problem of heap spray detection, provide improved algorithms for detecting suspicious heap objects, and describe the implementation of NOZZLE.

4.1 Formalization

This section formalizes our detection scheme informally described in Section 3.1, culminating in the notion of a *normalized attack surface*, a heap-global metric that reflects the overall heap exploitability and is used by NOZZLE to flag potential attacks.

Definition 1. *A sequence of bytes is* legitimate*, if it can be decoded as a sequence of valid x86 instructions. In a variable length ISA this implies that the processor must be able to decode every instruction of the sequence. Specifically, for each instruction, the byte sequence consists of a valid opcode and the correct number of arguments for that instruction.*

Unfortunately, the x86 instruction set is quite dense, and as a result, much of the heap data can be interpreted as legitimate x86 instructions. In our experiments, about 80% of objects allocated by Mozilla Firefox contain byte sequences that can be interpreted as x86 instructions.

Definition 2. *A* valid instruction sequence *is a legitimate instruction sequence that does* not *include instructions in the following categories:*

- *I/O or system calls (*`in`*,* `outs`*, etc)*
- *interrupts (*`int`*)*
- *privileged instructions (*`hlt`*,* `ltr`*)*
- *jumps outside of the current object address range.*

These instructions either divert control flow out of the object's implied control flow graph or generate exceptions and terminate (privileged instructions). If they appear in a path of the NOP sled, they prevent control flow from reaching the shellcode via that path. When these instructions appear in the shellcode, they do not hamper the control flow in the NOP sled leading to that shellcode in any way.

Semi-lattice	L	bitvectors of length N
Top	$\top$	$\bar{1}$
Initial value	$init(B_i)$	$\bar{0}$
Transfer function	$TF(B_i)$	$0 \ldots 010 \ldots 0$ (ith bit set)
Meet operator	$\wedge(x, y)$	$x \vee y$ (bitwise or)
Direction		*forward*

Figure 4: Dataflow problem parametrization for computing the surface area (see Aho et al.).

Previous work on NOP sled detection focuses on examining possible attacks for properties like valid instruction sequences [4, 43]. We use this definition as a basic object filter, with results presented in Section 5.1. Using this approach as the sole technique for detecting attacks leads to an unacceptable number of false positives, and more selective techniques are necessary.

To improve our selectivity, NOZZLE attempts to discover objects in which control flow through the object (the NOP sled) frequently reaches the same basic block(s) (the shellcode, indicated in Figure 1), the assumption being that an attacker wants to arrange it so that a random jump into the object will reach the shellcode with the greatest probability.

Our algorithm constructs a control flow graph (CFG) by interpreting the data in an object at offset Δ as an instruction stream. For now, we consider this offset to be zero and discuss the implications of malicious code injected at a different starting offset in Section 6. As part of the construction process, we mark the basic blocks in the CFG as valid and invalid instruction sequences, and we modify the definition of a basic block so that it terminates when an invalid instruction is encountered. A block so terminated is considered an invalid instruction sequence. For every basic block within the CFG we compute the *surface area*, a proxy for the likelihood of control flow passing through the basic block, should the attacker jump to a random memory address within the object.

Algorithm 1. Surface area computation.

Inputs: *Control flow graph C consisting of*

- *Basic blocks $B_1, \ldots, B_N$*
- *Basic block weights, $\bar{W}$, a single-column vector of size N where element W_i indicates the size of block B_i in bytes*
- *A validity bitvector $\bar{V}$, a single-row bitvector whose ith element is set to one only when block B_i contains a valid instruction sequence and set to zero otherwise.*
- *$MASK_1, \ldots, MASK_N$, where $MASK_i$ is a single-row bitvector of size N where all the bits are one except at the i^{th} position where the bit is zero.*

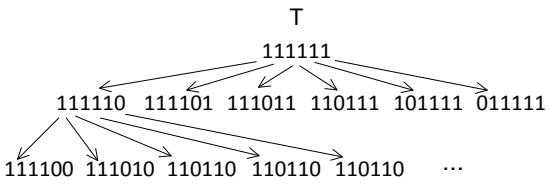

Figure 5: Semi-lattice used in Example 1.

Outputs: *Surface area for each basic block* $SA(B_i), B_i \in C$.

Solution: We define a parameterized dataflow problem using the terminology in Aho et al. [2], as shown in Figure 4. We also relax the definition of a conventional basic block; whenever an invalid instruction is encountered, the block prematurely terminates. The goal of the dataflow analysis is to compute the reachability between basic blocks in the control graph inferred from the contents of the object. Specifically, we want to determine whether control flow could possibly pass through a given basic block if control starts at each of the other $N-1$ blocks. Intuitively, if control reaches a basic block from many of the other blocks in the object (demonstrating a "funnel" effect), then that object exhibits behavior consistent with having a NOP sled and is suspicious.

Dataflow analysis details: The dataflow solution computes $out(B_i)$ for every basic block $B_i \in C$. $out(B_i)$ is a bitvector of length N, with one bit for each basic block in the control flow graph. The meaning of the bits in $out(B_i)$ are as follows: the bit at position j, where $j \neq i$ indicates whether a possible control path exists starting at block j and ending at block i. The bit at position i in B_i is always one. For example, in Figure 6, a path exists between block 1 and 2 (a fallthrough), and so the first bit of $out(B_2)$ is set to 1. Likewise, there is no path from block 6 to block 1, so the sixth bit of $out(B_1)$ is zero.

The dataflow algorithm computes $out(B_i)$ for each B_i by initializing them, computing the contribution that each basic block makes to $out(B_i)$, and propagating intermediate results from each basic block to its successors (because this is a forward dataflow computation). When results from two predecessors need to be combined at a join point, the meet operator is used (in this case a simple bitwise or). The dataflow algorithm iterates the forward propagation until the results computed for each B_i do not change further. When no further changes occur, the final values of $out(B_i)$ have been computed. The iterative algorithm for this forward dataflow problem is guaranteed to terminate in no more than the number of steps equal to the product of the semi-lattice height and the number of basic blocks in the control flow graph [2].

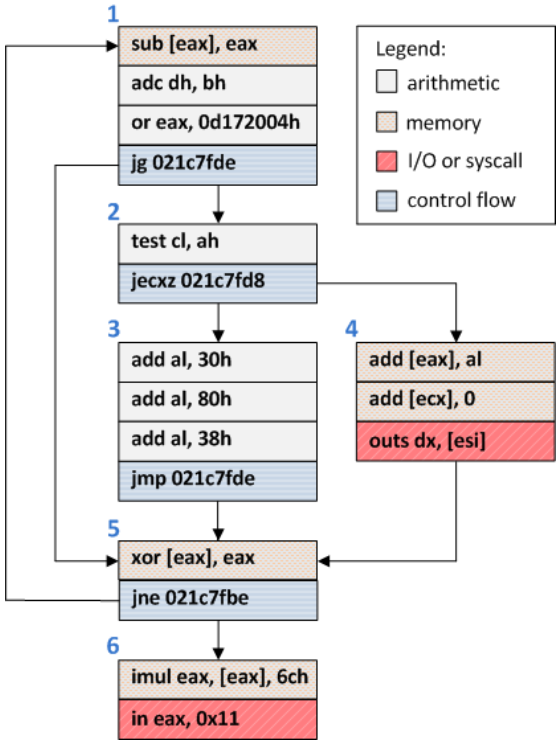

Figure 6: The control flow graph for Example 1.

Having calculated $out(B_i)$, we are now ready to compute the surface area of the basic block B_i. The surface area of a given block is a metric that indicates how likely the block will be reached given a random control flow landing on this object. The surface area of basic block B_i, $SA(B_i)$, is computed as follows:

$$SA(B_i) = (out(B_i) \wedge \bar{V} \wedge MASK_i) \cdot \bar{W}$$

where $out(B_i)$ is represented by a bitvector whose values are computed using the iterative dataflow algorithm above. $\bar{V}$, $\bar{W}$, and $MASK_i$ are the algorithm's inputs. $\bar{V}$ is determined using the validity criteria mentioned above, while $\bar{W}$ is the size of each basic block in bytes. $MASK_i$ is used to mask out the contribution of B_i's weight to its own surface area. The intuition is that we discard the contribution from the block itself as well as other basic blocks that are not valid instruction sequences by logically bitwise ANDing $out(B_i)$, $\bar{V}$, and $MASK_i$. Because the shellcode block does not contribute to actual attack surface (since a jump inside the shellcode is not likely to result in a successful exploit), we do not include the weight of B_i as part of the attack surface. Finally, we perform vector multiplication to account for the weight each basic block contributes—or does not—to the surface area of B_i.

In summary, the surface area computation based on the dataflow framework we described accounts for the contribution each basic block, through its weight and validity,

has on every other blocks reachable by it. Our computation method can handle code with complex control flow involving arbitrary nested loops. It also allows for the discovery of malicious objects even if the object has no clear partition between the NOP sled and the shellcode itself.

Complexity analysis. The standard iterative algorithm for solving dataflow problems computes $out(B_i)$ values with an average complexity bound of $O(N)$. The only complication is that doing the lattice meet operation on bitvectors of length N is generally an $O(N)$ and *not a constant time* operation. Luckily, for the majority of CFGs that arise in practice — 99.08% in the case of Mozilla Firefox opened and interacted on `www.google.com` — the number of basic blocks is fewer than 64, which allows us to represent dataflow values as long integers on 64-bit hardware. For those rare CFGs that contain over 64 basic blocks, a generic bitvector implementation is needed.

Example 1 Consider the CFG in Figure 6. The semilattice for this CFG of size 6 is partially shown in Figure 5. Instructions in the CFG are color-coded by instruction type. In particular, system calls and I/O instructions interrupt the normal control flow. For simplicity, we show $\bar{W}_i$ as the number of instructions in each block, instead of the number of bytes. The values used and produced by the algorithm are summarized in Figure 7. The $out'(B_i)$ column shows the intermediate results for dataflow calculation after the first pass. The final solution is shown in the $out(B_i)$ column. □

Given the surface area of individual blocks, we compute the *attack surface area* of object `o` as:

$$SA(o) = max(SA(B_i), B_i \in C)$$

For the entire heap, we accumulate the attack surface of the individual objects.

Definition 3. *The* attack surface area *of heap H, $SA(H)$, containing objects $o_1, \ldots, o_n$ is defined as follows:*

$$\sum_{i=1,\ldots,n} SA(o_i)$$

Definition 4. *The* normalized attack surface area *of heap H, denoted as $NSA(H)$, is defined as: $SA(H)/|H|$.*

The normalized attack surface area metric reflects the overall heap "health" and also allows us to adjust the frequency with which NOZZLE runs, thereby reducing the runtime overhead, as explained below.

4.2 Nozzle Implementation

NOZZLE needs to periodically scan heap object content in a way that is analogous to a garbage collector mark phase. By instrumenting allocation and deallocation routines, we maintain a table of live objects that are later scanned asynchronously, on a different NOZZLE thread.

We adopt garbage collection terminology in our description because the techniques are similar. For example, we refer to the threads allocating and freeing objects as the mutator threads, while we call the NOZZLE threads scanning threads. While there are similarities, there are also key differences. For example, NOZZLE works on an unmanaged, type-unsafe heap. If we had garbage collector write barriers, it would improve our ability to address the TOCTTOU (time-of-check to time-of-use) issue discussed in Section 6.

4.2.1 Detouring Memory Management Routines

We use a binary rewriting infrastructure called Detours [14] to intercept functions calls that allocate and free memory. Within Mozilla Firefox these routines are `malloc`, `calloc`, `realloc`, and `free`, defined in `MOZCRT19.dll`. To compute the surface area, we maintain information about the heap including the total size of allocated objects.

NOZZLE maintains a hash table that maps the addresses of currently allocated objects to information including size, which is used to track the current size and contents of the heap. When objects are freed, we remove them from the hash table and update the size of the heap accordingly. Note that if NOZZLE were more closely integrated into the heap allocator itself, this hash table would be unnecessary.

NOZZLE maintains an ordered work queue that serves two purposes. First, it is used by the scanning thread as a source of objects that need to be scanned. Second, NOZZLE waits for objects to mature before they are scanned, and this queue serves that purpose. Nozzle only considers objects of size greater than 32 bytes to be put in the work queue as the size of any harmful shellcode is usually larger than this

To reduce the runtime overhead of NOZZLE, we randomly sample a subset of heap objects, with the goal of covering a fixed fraction of the total heap. Our current sampling technique is based on sampling by object, but as our results show, an improved technique would base sampling frequency on bytes allocated, as some of the published attacks allocate a relatively small number of large objects.

4.2.2 Concurrent Object Scanning

We can reduce the performance impact of object scanning, especially on multicore hardware, with the help of multiple scanning threads. As part of program detouring, we rewrite the `main` function to allocate a pool of N scanning threads to be used by NOZZLE, as shown in Figure 2.

B_i	$TF(B_i)$	$\bar{V}_i$	$\bar{W}_i$	$out'(B_i)$	$out(B_i)$	$out(B_i) \wedge \bar{V} \wedge MASK_i$	$SA(B_i)$
1	100000	1	4	100000	111110	011010	8
2	010000	1	2	110000	111110	101010	10
3	001000	1	4	111000	111110	110010	8
4	000100	0	3	110100	111110	111010	12
5	000010	1	2	111110	111110	111000	10
6	000001	0	2	111111	111111	111010	12

Figure 7: Dataflow values for Example 1.

This way, a mutator only blocks long enough when allocating and freeing objects to add or remove objects from a per-thread work queue.

The task of object scanning is subdivided among the scanning threads the following way: for an object at address a, thread number

$$(a >> p) \ \% \ N$$

is responsible for both maintaining information about that object and scanning it, where p is the number of bits required to encode the operating system page size (typically 12 on Windows). In other words, to preserve the spatial locality of heap access, we are distributing the task of scanning *individual pages* among the N threads. Instead of maintaining a global hash table, each thread maintains a local table keeping track of the sizes for the objects it handles.

Object scanning can be triggered by a variety of events. Our current implementation scans objects once, after a fixed delay of one object allocation (i.e., we scan the previously allocated object when we see the next object allocated). This choice works well for JavaScript, where string objects are immutable, and hence initialized immediately after they are allocated. Alternately, if there are extra cores available, scanning threads could pro-actively rescan objects without impacting browser performance and reducing TOCTTOU vulnerabilities (see Section 6).

4.3 Detection and Reporting

NOZZLE maintains the values $NSA(H)$ and $SA(H)$ for the currently allocated heap H. The criteria we use to conclude that there is an attack in progress combines an absolute and a relative threshold:

$$(NSA(H) > th_{norm}) \wedge (SA(H) > th_{abs})$$

When this condition is satisfied, we warn the user about a potential security attack in progress and allow them to kill the browser process. An alternative would be to take advantage of the error reporting infrastructure built into modern browsers to notify the browser vendor.

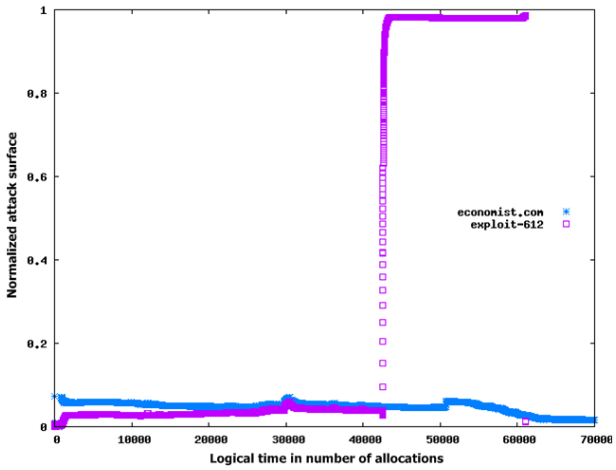

Figure 8: Global normalized attack surface for `economist.com` versus a published exploit (612).

These thresholds are defined based on a comparison of benign and malicious web pages (Section 5.1). The guiding principle behind the threshold determination is that for the attacker to succeed, the exploit needs to be effective with reasonable probability. For the absolute threshold, we choose five megabytes, which is roughly the size of the Firefox heap when opening a blank page. A real attack would need to fill the heap with at least as many malicious objects, assuming the attacker wanted the ratio of malicious to non-malicious objects to be greater than 50%.

5 Evaluation

The bulk of our evaluation focuses on applying NOZZLE to the Firefox web browser. Section 5.5 talks about using NOZZLE to protect Adobe Acrobat Reader.

We begin our evaluation by showing what a heap-spraying attack looks like as measured using our normalized attack surface metric. Figure 8 shows the attack surface area of the heap for two web sites: a benign site (`economist.com`), and a site with a published heap-spraying attack, similar to the one presented in Figure 2. Figure 8 illustrates how distinctive a heap-spraying attack

is when viewed through the normalized attack surface filter. The success of NOZZLE depends on its ability to distinguish between these two kinds of behavior. After seeing Figure 8, one might conclude that we can detect heap spraying activity based on how rapidly the heap grows. Unfortunately, benign web sites as `economist.com` can possess as high a heap growth rate as a rogue page performing heap spraying. Moreover, unhurried attackers may avoid such detection by moderating the heap growth rate of their spray. In this section, we present the false positive and false negative rate of NOZZLE, as well as its performance overhead, demonstrating that it can effectively distinguish benign from malicious sites.

For our evaluations, we collected 10 heavily-used benign web sites with a variety of content and levels of scripting, which we summarize in Figure 9. We use these 10 sites to measure the false positive rate and also the impact of NOZZLE on browser performance, discussed in Section 5.3. In our measurements, when visiting these sites, we interacted with the site as a normal user would, finding a location on a map, requesting driving directions, etc. Because such interaction is hard to script and reproduce, we also studied the false positive rate of NOZZLE using a total of 150 benign web sites, chosen from the most visited sites as ranked by Alexa [5][1]. For these sites, we simply loaded the first page of the site and measured the heap activity caused by that page alone.

To evaluates NOZZLE's ability to detect malicious attacks, we gathered 12 published heap-spraying exploits, summarized in Figure 10. We also created 2,000 synthetically generated exploits using the Metasploit framework [12]. Metasploit allows us to create many malicious code sequences with a wide variety of NOP sled and shellcode contents, so that we can evaluate the ability of our algorithms to detect such attacks. Metasploit is parameterizable, and as a result, we can create attacks that contain NOP sleds alone, or NOP sleds plus shellcode. In creating our Metasploit exploits, we set the ratio of NOP sled to shellcode at 9:1, which is quite a low ratio for a real attack but nevertheless presents no problems for NOZZLE detection.

5.1 False Positives

To evaluate the false positive rate, we first consider using NOZZLE as a global detector determining whether a heap is under attack, and then consider the false-positive rate of NOZZLE as a local detector that is attempting to detect individual malicious objects. In our evaluation, we compare NOZZLE and STRIDE [4], a recently published local detector.

[1] Our tech report lists the full set of sites used [32].

Site URL	Download (kilobytes)	JavaScript (kilobytes)	Load time (seconds)
`economist.com`	613	112	12.6
`cnn.com`	885	299	22.6
`yahoo.com`	268	145	6.6
`google.com`	25	0	0.9
`amazon.com`	500	22	14.8
`ebay.com`	362	52	5.5
`facebook.com`	77	22	4.9
`youtube.com`	820	160	16.5
`maps.google.com`	285	0	14.2
`maps.live.com`	3000	2000	13.6

Figure 9: Summary of 10 benign web sites we used as NOZZLE benchmarks.

Date	Browser	Description	milw0rm
11/2004	IE	IFRAME Tag BO	612
04/2005	IE	DHTML Objects Corruption	930
01/2005	IE	.ANI Remote Stack BO	753
07/2005	IE	`javaprxy.dll` COM Object	1079
03/2006	IE	`createTextRang` RE	1606
09/2006	IE	VML Remote BO	2408
03/2007	IE	ADODB Double Free	3577
09/2006	IE	WebViewFolderIcon `setSlice`	2448
09/2005	FF	0xAD Remote Heap BO	1224
12/2005	FF	`compareTo()` RE	1369
07/2006	FF	`Navigator` Object RE	2082
07/2008	Safari	Quicktime Content-Type BO	6013

Figure 10: Summary of information about 12 published heap-spraying exploits. BO stands for "buffer overruns" and RE stands for "remote execution."

5.1.1 Global False Positive Rate

Figure 11 shows the maximum normalized attack surface measured by NOZZLE for our 10 benchmark sites (top) as well as the top 150 sites reported by Alexa (bottom). From the figure, we see that the maximum normalized attack surface remains around 6% for most of the sites, with a single outlier from the 150 sites at 12%. In practice, the median attack surface is typically much lower than this, with the maximum often occurring early in the rendering of the page when the heap is relatively small. The `economist.com` line in Figure 8 illustrates this effect. By setting the spray detection threshold at 15% or above, we would observe no false positives in any of the sites measured.

5.1.2 Local False Positive Rate

In addition to being used as a heap-spray detector, NOZZLE can also be used locally as a malicious object detector. In this use, as with existing NOP and shellcode detectors such as STRIDE [4], a tool would report an object as potentially malicious if it contained data that could be interpreted as code, and had other suspicious properties. Previous work in this area focused on detection of malware in network packets and URIs, whose content is very different than heap objects. We evaluated NOZZLE

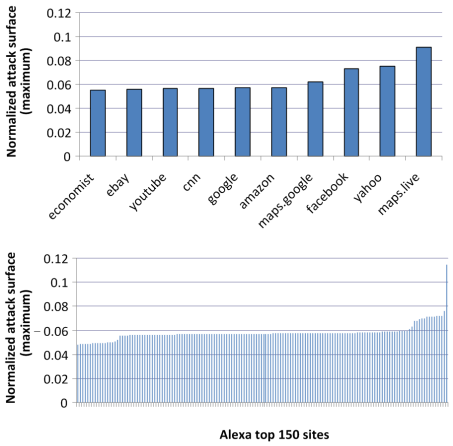

Figure 11: Global normalized attack surface for 10 benign benchmark web sites and 150 additional top Alexa sites, sorted by increasing surface. Each element of the X-axis represents a different web site.

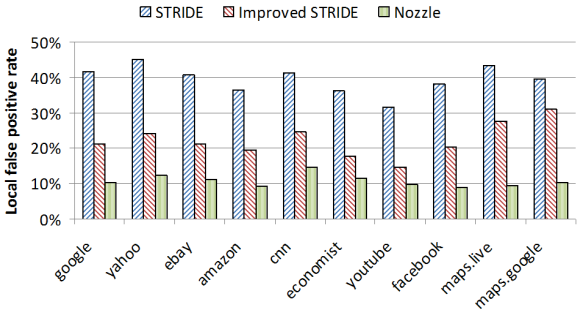

Figure 12: Local false positive rate for 10 benchmark web sites using NOZZLE and STRIDE. Improved STRIDE is a version of STRIDE that uses additional instruction-level filters, also used in NOZZLE, to reduce the false positive rate.

and STRIDE algorithm, to see how effective they are at classifying benign heap objects.

Figure 12 indicates the false positive rate of two variants of STRIDE and a simplified variant of NOZZLE. This simplified version of NOZZLE only scans a given heap object and attempts to disassemble and build a control flow graph from its contents. If it succeeds in doing this, it considers the object suspect. This version does not include any attack surface computation. The figure shows that, unlike previously reported work where the false positive rate for URIs was extremely low, the false positive rate for heap objects is quite high, sometimes above 40%. An improved variant of STRIDE that uses more information about the x86 instruction set (also used in NOZZLE) reduces this rate, but not below 10% in any case. We conclude that, unlike URIs or the content of network packets, heap objects often have contents that can be entirely interpreted as code on the x86 architecture. As a result, existing methods of sled detection do not directly apply to heap objects. We also show that even NOZZLE, without incorporating our surface area computation, would have an unacceptably high false positive rate.

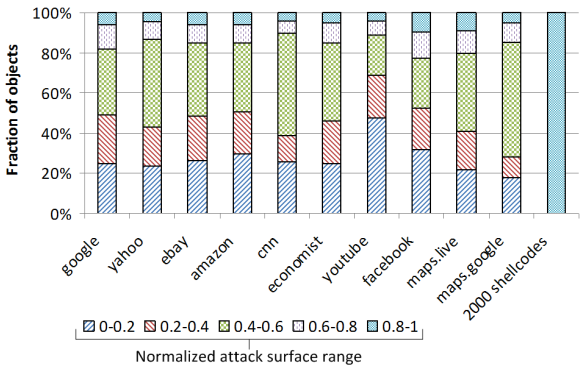

Figure 13: Distribution of filtered object surface area for each of 10 benchmark web sites (benign) plus 2,000 synthetic exploits (see Section 5.2). Objects measured are only those that were considered valid instruction sequences by NOZZLE (indicated as false positives in Figure 12.

To increase the precision of a local detector based on NOZZLE, we incorporate the surface area calculation described in Section 4. Figure 13 indicates the distribution of measured surface areas for the roughly 10% of objects in Figure 12 that our simplified version of NOZZLE was not able to filter. We see from the figure that many of those objects have a relatively small surface area, with less that 10% having surface areas from 80-100% of the size of the object (the top part of each bar). Thus, roughly 1% of objects allocated by our benchmark web sites qualify as suspicious by a local NOZZLE detector, compared to roughly 20% using methods reported in prior work. Even at 1%, the false positive rate of a local NOZZLE detector is too high to raise an alarm whenever a single instance of a suspicious object is observed, which motivated the development of our global heap health metric.

5.2 False Negatives

As with the false positive evaluation, we can consider NOZZLE both as a local detector (evaluating if NOZZLE is capable of classifying a known malicious object correctly), and as a global detector, evaluating whether it correctly detects web pages that attempt to spray many copies of malicious objects in the heap.

Figure 14 evaluates how effective NOZZLE is at avoid-

Configuration	min	mean	std
Local, NOP w/o shellcode	0.994	0.997	0.002
Local, NOP with shellcode	0.902	0.949	0.027

Figure 14: Local attack surface metrics for 2,000 generated samples from Metasploit with and without shellcode.

Configuration	min	mean	std
Global, published exploits	0.892	0.966	0.028
Global, Metasploit exploits	0.729	0.760	0.016

Figure 15: Global attack surface metrics for 12 published attacks and 2,000 Metasploit attacks integrated into web pages as heap sprays.

ing local false negatives using our Metasploit exploits. The figure indicates the mean and standard deviation of the object surface area over the collection of 2,000 exploits, both when shellcode is included with the NOP sled and when the NOP sled is measured alone. The figure shows that NOZZLE computes a very high attack surface in both cases, effectively detecting all the Metasploit exploits both with and without shellcode.

Figure 15 shows the attack surface statistics when using NOZZLE as a global detector when the real and synthetic exploits are embedded into a web page as a heap-spraying attack. For the Metasploit exploits which were not specifically generated to be heap-spraying attacks, we wrote our own JavaScript code to spray the objects in the heap. The figure shows that the published exploits are more aggressive than our synthetic exploits, resulting in a mean global attack surface of 97%. For our synthetic use of spraying, which was more conservative, we still measured a mean global attack surface of 76%. Note that if we set the NOP sled to shellcode at a ratio lower than 9:1, we will observe a correspondingly smaller value for the mean global attack surface. All attacks would be detected by NOZZLE with a relatively modest threshold setting of 50%. We note that these exploits have global attack surface metrics 6–8 times larger than the maximum measured attack surface of a benign web site.

5.3 Performance

To measure the performance overhead of NOZZLE, we cached a typical page for each of our 10 benchmark sites. We then instrument the start and the end of the page with the JavaScript `newDate().getTime()` routine and compute the delta between the two. This value gives us how long it takes to load a page in milliseconds. We collect our measurements running Firefox version 2.0.0.16 on a 2.4 GHz Intel Core2 E6600 CPU running Windows XP

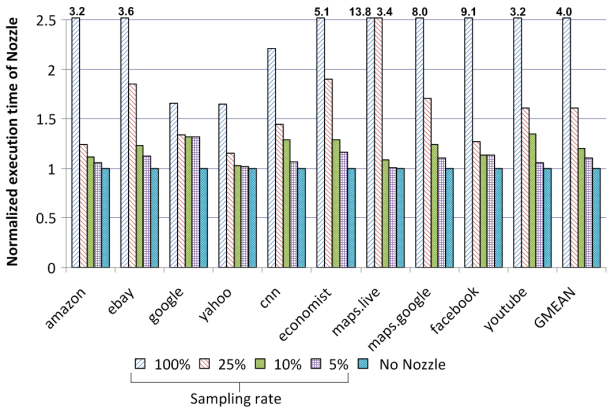

Figure 16: Relative execution overhead of using NOZZLE in rendering a typical page of 10 benchmark web sites as a function of sampling frequency.

Service Pack 3 with 2 gigabytes of main memory. To minimize the effect of timing due to cold start disk I/O, we first load a page and discard the timing measurement. After this first trial, we take three measurements and present the median of the three values. The experiments were performed on an otherwise quiescent machine and the variance between runs was not significant.

In the first measurement, we measured the overhead of NOZZLE without leveraging an additional core, i.e., running NOZZLE as a single thread and, hence, blocking Firefox every time a memory allocation occurs. The resulting overhead is shown in Figure 16, both with and without sampling. The overhead is prohibitively large when no sampling is applied. On average, the no sampling approach incurs about 4X slowdown with as much as 13X slowdown for `maps.live.com`. To reduce this overhead, we consider the sampling approach. For these results, we sample based on object counts; for example, sampling at 5% indicates that one in twenty objects is sampled. Because a heap-spraying attack has global impact on the heap, sampling is unlikely to significantly reduce our false positive and false negative rates, as we show in the next section. As we reduce the sampling frequency, the overhead becomes more manageable. We see an average slowdown of about 60%, 20% and 10% for sampling frequency of 25%, 10% and 5%, respectively, for the 10 selected sites.

For the second measurement, taking advantage of the second core of our dual core machine, we configured NOZZLE to use one additional thread for scanning, hence, unblocking Firefox when it performs memory allocation. Figure 17 shows the performance overhead of NOZZLE with parallel scanning. From the Figure, we see that with no sampling, the overhead of using NOZZLE ranges from 30% to almost a factor of six, with a geometric mean of

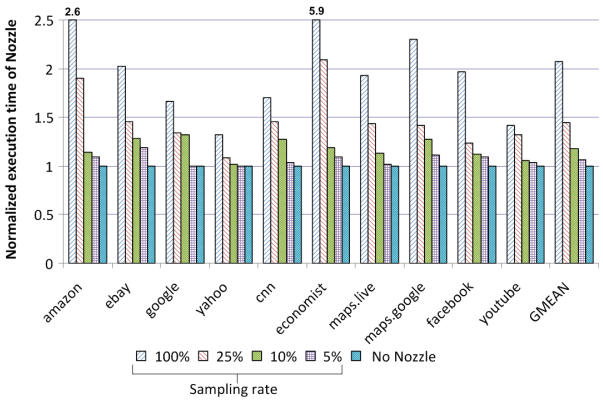

Figure 17: Overhead of using NOZZLE on a dual-core machine.

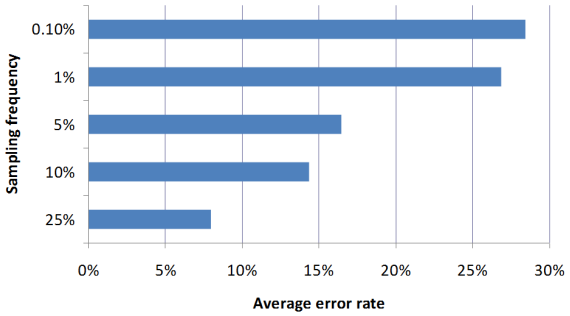

Figure 18: Average error rate due to sampling of the computed average surface area for 10 benign benchmark web sites.

two times slowdown. This is a significant improvement over the serial version. When we further reduce the sampling rate, we see further performance improvement as with the first measurement. Reducing the sampling rate to 25%, the mean overhead drops to 45%, while with a sampling rate of 5%, the performance overhead is only 6.4%.

5.4 Impact of Sampling on Detection

In this section, we show the impact of sampling on the amount of error in the computation of the attack surface metric for both benign and malicious inputs.

Figure 18 shows the error rate caused by different levels of sampling averaged across the 10 benign web sites. We compute the error rate $E = |Sampled - Unsampled|/Unsampled$. The figure shows that for sample rates of 0.1% or above the error rate is less than 30%. To make this concrete, for a benign website, instead of calculating the normalized attack surface correctly as 5%, with a 0.1% sampling rate, we would instead calculate the normalized attack surface as 6.5%, still far below any threshold we might use for signaling an attack. Noting that the malicious pages have attack surfaces that are 6–8 times larger than benign web pages, we conclude that sampling even at 5% is unlikely to result in significant numbers of false positives.

	Sampling Rate				
	100%	25%	10%	5%	1%
12 Published	0	0	0	0	50%
2,000 Metasploit	0	0	0	0	0

Figure 19: False negative rate for 12 real and 2,000 Metasploit attacks given different object sampling rates.

In Figure 19, we show the impact of sampling on the number of false negatives for our published and synthetic exploits. Because existing exploits involve generating the heap spray in a loop, the only way sampling will miss such an attack is to sample at such a low rate that the objects allocated in the loop escape notice. The figure illustrates that for published attacks sampling even at 5% results in no false negatives. At 1%, because several of the published exploits only create on the order of tens of copies of very large spray objects, sampling based on object count can miss these objects, and we observe a 50% (6/12) false negative rate. Sampling based on bytes allocated instead of objects allocated would reduce this false negative rate to zero.

5.5 Case Study: Adobe Reader Exploit

In February 2009, a remote code execution vulnerability was discovered in Adobe Acrobat and Adobe Reader [26]. The attack, which is still active on the Internet as of the time of this writing, exploited an integer overflow error and was facilitated by a JavaScript heap spray. Without making any modifications to NOZZLE, we used Detours to instrument the commercially-distributed binary of Adobe Reader 9.1.0 (`acrord32.exe`) with NOZZLE. The instrumentation allowed us to monitor the memory allocations being performed by the embedded JavaScript engine and detect possible spraying attacks. To test whether NOZZLE would detect this new attack, we embedded the heap spraying part of the published attack [6], disabling the JavaScript that caused the integer overflow exploit.

NOZZLE correctly detected this heap spraying attack, determining that the attack surface of the heap was greater than 94% by the time the heap spray was finished. No modifications were made either to the NOZZLE implementation or the surface area calculation to enable NOZZLE to detect this attack, which gives us confidence that NOZZLE is capable of protecting a wide range of software, going well beyond just web browsers.

To facilitate overhead measurements, we created a large document by concatenating six copies of the ECMA 262 standard — a 188-page PDF document [11] — with itself. The resulting document was 1,128 pages long and took 4,207 kilobytes of disk space. We added scripting code to the document to force Adobe Reader to "scroll" through this large document, rendering every page sequentially. We believe this workload to be representative of typical Adobe Reader usage, where the user pages through the document, one page at a time.

We measured the overhead of NOZZLE running in Adobe Reader on an Intel Core 2 2.4 GHz computer with 4 GB of memory running Windows Vista SP1. We measured elapsed time for Adobe Reader with and without NOZZLE on a lightly loaded computer and averaged five measurements with little observed variation. Without NOZZLE, Adobe Reader took an average of 18.7 seconds to render all the pages, and had a private working set of 18,772 kilobytes as measured with the Windows Task Manager. With a sampling rate set to 10% and multiprocessor scanning disabled, Adobe Reader with NOZZLE took an average of 20.2 seconds to render the pages, an average CPU overhead of 8%. The working set of Adobe Reader with NOZZLE average 31,849 kilobytes, an average memory overhead of 69%. While the memory overhead is high, as mentioned, we anticipate that this overhead could easily be reduced by integrating NOZZLE more closely with the underlying memory manager.

6 Limitations of NOZZLE

This section discusses assumptions and limitations of the current version of NOZZLE. In summary, assuming that the attackers are fully aware of the NOZZLE internals, there are a number of ways to evade its detection.

- As NOZZLE scans objects at specific times, an attacker could determine that an object has been scanned and arrange to put malicious content into the object only *after* it has been scanned (a TOCTTOU vulnerability).
- As NOZZLE currently starts scanning each object at offset zero, attackers can avoid detection by writing the first few bytes of the malicious object with a series of uninterpretable opcodes.
- Since NOZZLE relies on the use of a threshold for detection, attackers can populate the heap with fewer malicious objects to stay just under the detection threshold.
- Attackers can find ways to inject the heap with sprays that do not require large NOP sleds. For example, sprays with jump targets that are at fixed offsets in every sprayed page of memory are possible [39].
- Attackers can confuse NOZZLE's surface area measurement by designing attacks that embed multiple shellcodes within the same object or contain cross-object jumps.

Below we discuss these issues, their implications, and possible ways to address them.

6.1 Time-of-check to Time-of-use

Because NOZZLE examines object contents only at specific times, this leads to a potential time-of-check to time-of-use (TOCTTOU) vulnerability. An attacker aware that NOZZLE was being used could allocate a benign object, wait until NOZZLE scans it, and then rapidly change the object into a malicious one before executing the attack.

With JavaScript-based attacks, since JavaScript `Strings` are immutable, this is generally only possible using JavaScript `Arrays`. Further, because NOZZLE may not know when objects are completely initialized, it may scan them prematurely, creating another TOCTTOU window. To address this issue, NOZZLE scans objects once they mature on the assumption that most objects are written once when initialized, soon after they are allocated. In the future, we intend to investigate other ways to reduce this vulnerability, including periodically rescanning objects. Rescans could be triggered when NOZZLE observes a significant number of heap stores, perhaps by reading hardware performance counters.

Moreover, in the case of a garbage-collected language such as JavaScript or Java, NOZZLE can be integrated directly with the garbage collector. In this case, changes observed via GC *write barriers* [29] may be used to trigger NOZZLE scanning.

6.2 Interpretation Start Offset

In our discussion so far, we have interpreted the contents of objects as instructions starting at offset zero in the object, which allows NOZZLE to detect the current generation of heap-spraying exploits. However, if attackers are aware that NOZZLE is being used, they could arrange to fool NOZZLE by inserting junk bytes at the start of objects. There are several reasons that such techniques will not be as successful as one might think. To counter the most simplistic such attack, if there are invalid or illegal instructions at the beginning of the object, NOZZLE skips bytes until it finds the first valid instruction.

Note that while it may seem that the attacker has much flexibility to engineer the offset of the start of the malicious code, the attacker is constrained in several important ways. First, we know that it is likely that the attacker is trying to maximize the probability that the attack will succeed. Second, recall that the attacker has to corrupt a control transfer but does not know the specific address in an

object where the jump will land. If the jump lands on an invalid or illegal instruction, then the attack will fail. As a result, the attacker may seek to make a control transfer to every address in the malicious object result in an exploit. If this is the case, then NOZZLE will correctly detect the malicious code. Finally, if the attacker knows that NOZZLE will start interpreting the data as instructions starting at a particular offset, then the attacker might intentionally construct the sled in such a way that the induced instructions starting at one offset look benign, while the induced instructions starting at a different offset are malicious. For example, the simplist form of this kind of attack would have uniform 4-byte benign instructions starting at byte offset 0 and uniform malicious 4-byte instructions starting at byte offset 2. Note also that these overlapped sequences cannot share any instruction boundaries because if they did, then processing instructions starting at the benign offset would eventually discover malicious instructions at the point where the two sequences merged.

While the current version of NOZZLE does not address this specific simple case, NOZZLE could be modified to handle it by generating multiple control flow graphs at multiple starting offsets. Furthermore, because x86 instructions are typically short, most induced instruction sequences starting at different offsets do not have many possible interpretations before they share a common instruction boundary and merge. While it is theoretically possible for a determined attacker to create a non-regular, non-overlapping sequence of benign and malicious instructions, it is not obvious that the malicious sequence could not be discovered by performing object scans at a small number of offsets into the object. We leave an analysis of such techniques for future work.

6.3 Threshold Setting

The success of heap spraying is directly proportional to the density of dangerous objects in the program heap, which is approximated by NOZZLE's normalized attack surface metric. Increasing the number of sprayed malicious objects increases the attacker's likelihood of success, but at the same time, more sprayed objects will increase the likelihood that NOZZLE will detect the attack. What is worse for the attacker, failing attacks often result in program crashes. In the browser context, these are recorded on the user's machine and sent to browser vendors using a crash agent such as Mozilla Crash reporting [24] for per-site bucketing and analysis.

What is interesting about attacks against browsers is that from a purely financial standpoint, the attacker has a strong incentive to produce exploits with a high likelihood of success. Indeed, assuming the attacker is the one discovering the vulnerability such as a dangling pointer or buffer overrun enabling the heap-spraying attack, they can sell their finding directly to the browser maker. For instance, the Mozilla Foundation, the makers of Firefox, offers a cash reward of $500 for every exploitable vulnerability [25]. This represents a conservative estimate of the financial value of such an exploit, given that Mozilla is a non-profit and commercial browser makes are likely to pay more [15]. A key realization is that in many cases heap spraying is used for direct financial gain. A typical way to monetize a heap-spraying attack is to take over a number of unsuspecting users' computers and have them join a botnet. A large-scale botnet can be sold on the black market to be used for spamming or DDOS attacks.

According to some reports, to cost of a large-scale botnet is about $.10 per machine [40, 18]. So, to break even, the attacker has to take over at least 5,000 computers. For a success rate α, in addition to 5,000 successfully compromised machines, there are $5,000 \times (1-\alpha)/\alpha$ failed attacks. Even a success rate as high as 90%, the attack campaign will produce failures for 555 users. Assuming these result in crashes reported by the crash agent, this many reports from a single web site may attract attention of the browser maker. For a success rate of 50%, the browser make is likely to receive 5,000 crash reports, which should lead to rapid detection of the exploit!

As discussed in Section 5, in practice we use the relative threshold of 50% with Nozzle. We do not believe that a much lower success rate is financially viable from the standpoint of the attacker.

6.4 Targeted Jumps into Pages

One approach to circumventing NOZZLE detection is for the attacker to eliminate the large NOP sled that heap sprays typically use. This may be accomplished by allocating page-size chunks of memory (or multiples thereof) and placing the shellcode at fixed offsets on every page [39]. While our spraying detection technique currently will not discover such attacks, it is possible that the presence of possible shellcode at fixed offsets on a large number of user-allocated pages can be detected by extending NOZZLE, which we will consider in future work.

6.5 Confusing Control Flow Patterns

NOZZLE attempts to find basic blocks that act as sinks for random jumps into objects. One approach that will confuse NOZZLE is to include a large number of copies of shellcode in an object such that no one of them has a high surface area. Such an approach would still require that a high percentage of random jumps into objects result in non-terminating control flow, which might also be used as a trigger for our detector.

Even more problematic is an attack where the attacker includes inter-object jumps, under the assumption that,

probabilistically, there will be a high density of malicious objects and hence jumps outside of the current object will still land in another malicious object. NOZZLE currently assumes that jumps outside of the current object will result in termination. We anticipate that our control flow analysis can be augmented to detect groupings of objects with possible inter-object control flow, but we leave this problem for future work.

6.6 Summary

In summary, there are a number of ways that clever attackers can defeat NOZZLE's current analysis techniques. Nevertheless, we consider NOZZLE an important first step to detecting heap spraying attacks and we believe that improvements to our techniques are possible and will be implemented, just as attackers will implement some of the possible exploits described above.

The argument for using NOZZLE, despite the fact that hackers will find ways to confound it, is the same reason that virus scanners are installed almost ubiquitously on computer systems today: it will detect and prevent many known attacks, and as new forms of attacks develop, there are ways to improve its defenses as well. Ultimately, NOZZLE, just like existing virus detectors, is just one layer of a defense in depth.

7 Related Work

This section discusses exploit detection and memory attack prevention.

7.1 Exploit Detection

As discussed, a code injection exploit consists of at least two parts: the NOP sled and shellcode. Detection techniques target either or both of these parts. Signature-based techniques, such as Snort [33], use pattern matching, including possibly regular expressions, to identify attacks that match known attacks in their database. A disadvantage of this approach is that it will fail to detect attacks that are not already in the database. Furthermore, polymorphic malware potentially vary its shellcode on every infection attempt, reducing the effectiveness of pattern-based techniques. Statistical techniques, such as Polygraph [27], address this problem by using improbable properties of the shellcode to identify an attack.

The work most closely related to NOZZLE is Abstract Payload Execution (APE), by Toth and Kruegel [43], and STRIDE, by Akritidis et al. [4, 30], both of which present methods for NOP sled detection in network traffic. APE examines sequences of bytes using a technique they call *abstract execution* where the bytes are considered valid and correct if they represent processor instructions with legal memory operands. APE identifies sleds by computing the execution length of valid and correct sequences, distinguishing attacks by identifying sufficiently long sequences.

The authors of STRIDE observe that by employing jumps, NOP sleds can be effective even with relatively short valid and correct sequences. To address this problem, they consider all possible subsequences of length n, and detect an attack only when all such subsequences are considered valid. They demonstrate that STRIDE handles attacks that APE does not, showing also that tested over a large corpus or URIs, their method has an extremely low false positive rate. One weakness of this approach, acknowledged by the authors is that "...a worm writer could blind STRIDE by adding invalid instruction sequences at suitable locations..." ([30], p. 105).

NOZZLE also identifies NOP sleds, but it does so in ways that go beyond previous work. First, prior work has not considered the specific problem of sled detection in heap objects or the general problem of heap spraying, which we do. Our results show that applying previous techniques to heap object results in high false positive rates. Second, because we target heap spraying specifically, our technique leverages properties of the entire heap and not just individual objects. Finally, we introduce the concept of surface area as a method for prioritizing the potential threat of an object, thus addressing the STRIDE weakness mentioned above.

7.2 Memory Attack Prevention

While NOZZLE focuses on detecting heap spraying based on object and heap properties, other techniques take different approaches. Recall that heap spraying requires an additional memory corruption exploit, and one method of preventing a heap-spraying attack is to ignore the spray altogether and prevent or detect the initial corruption error. Techniques such as control flow integrity [1], write integrity testing [3], data flow integrity [9], and program shepherding [17] take this approach. Detecting all such possible exploits is difficult and, while these techniques are promising, their overhead has currently prevented their widespread use.

Some existing operating systems also support mechanisms, such as Data Execution Prevention (DEP) [21], which prevent a process from executing code on specific pages in its address space. Implemented in either software or hardware (via the no-execute or "NX" bit), execution protection can be applied to vulnerable parts of an address space, including the stack and heap. With DEP turned on, code injections in the heap cannot execute.

While DEP will prevent many attacks, we believe that NOZZLE is complementary to DEP for the following rea-

sons. First, security benefits from defense-in-depth. For example, attacks that first turn off DEP have been published, thereby circumventing its protection [37]. Second, compatibility issues can prevent DEP from being used. Despite the presence of NX hardware and DEP in modern operating systems, existing commercial browsers, such as Internet Explorer 7, still ship with DEP disabled by default [13]. Third, runtime systems that perform just-in-time (JIT) compilation may allocate JITed code in the heap, requiring the heap to be executable. Finally, code injection spraying attacks have recently been reported in areas other than the heap where DEP cannot be used. Sotirov and Dowd describe spraying attacks that introduce exploit code into a process address space via embedded .NET User Controls [42]. The attack, which is disguised as one or more .NET managed code fragments, is loaded in the process text segment, preventing the use of DEP. In future work, we intend to show that NOZZLE can be effective in detecting such attacks as well.

8 Conclusions

We have presented NOZZLE, a runtime system for detecting and preventing heap-spraying security attacks. Heap spraying has the property that the actions taken by the attacker in the spraying part of the attack are legal and type safe, allowing such attacks to be initiated in JavaScript, Java, or C#. Attacks using heap spraying are on the rise because security mitigations have reduced the effectiveness of previous stack and heap-based approaches.

NOZZLE is the first system specifically targeted at detecting and preventing heap-spraying attacks. NOZZLE uses lightweight runtime interpretation to identify specific suspicious objects in the heap and maintains a global heap health metric to achieve low false positive and false negative rates, as measured using 12 published heap spraying attacks, 2,000 synthetic malicious exploits, and 150 highly-visited benign web sites. We show that with sampling, the performance overhead of NOZZLE can be reduced to 7%, while maintaining low false positive and false negative rates. Similar overheads are observed when NOZZLE is applied to Adobe Acrobat Reader, a recent target of heap spraying attacks. The fact that NOZZLE was able to thwart a real published exploit when applied to the Adobe Reader binary, without requiring any modifications to our instrumentation techniques, demonstrates the generality of our approach.

While we have focused our experimental evaluation on heap-spraying attacks exclusively, we believe that the our techniques are more general. In particular, in future work, we intend to investigate using our approach to detect a variety of exploits that use code masquarading as data, such as images, compiled bytecode, etc. [42].

In the future, we intend to further improve the selectivity of the NOZZLE local detector, demonstrate NOZZLE's effectiveness for attacks beyond heap spraying, and further tune NOZZLE's performance. Because heap-spraying attacks can be initiated in type-safe languages, we would like to evaluate the cost and effectiveness of incorporating NOZZLE in a garbage-collected runtime. We are also interested in extending NOZZLE from detecting heap-spraying attacks to tolerating them as well.

Acknowledgements

We thank Emery Berger, Martin Burtscher, Silviu Calinoiu, Trishul Chilimbi, Tal Garfinkel, Ted Hart, Karthik Pattabiraman, Patrick Stratton, and Berend-Jan "SkyLined" Wever for their valuable feedback during the development of this work. We also thank our anonymous reviewers for their comments.

References

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[4] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, pages 375–392. Springer, 2005.

[5] Alexa Inc. Global top sites. `http://www.alexa.com/site/ds/top_sites`, 2008.

[6] Arr1val. Exploit made by Arr1val proved in Adobe 9.1 and 8.1.4 on Linux. `http://downloads.securityfocus.com/vulnerabilities/exploits/34736.txt`, Feb. 2009.

[7] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[8] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120. USENIX, Aug. 2003.

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.

[11] ECMA. ECMAScript language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 1999.

[12] J. C. Foster. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007.

[13] M. Howard. Update on Internet Explorer 7, DEP, and Adobe software. `http://blogs.msdn.com/michael_howard/archive/2006/12/12/update-on-internet-explorer-7-dep-and-adobe-software.aspx`, 2006.

[14] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *In Proceedings of the USENIX Windows NT Symposium*, pages 135–143, 1999.

[15] iDefense Labs. Annual vulnerability challenge. `http://labs.idefense.com/vcp/challenge.php`, 2007.

[16] I.-K. Kim, K. Kang, Y. Choi, D. Kim, J. Oh, and K. Han. A practical approach for detecting executable codes in network traffic. In S. Ata and C. S. Hong, editors, *Proceedings of Managing Next Generation Networks and Services*, volume 4773 of *Lecture Notes in Computer Science*, pages 354–363. Springer, 2007.

[17] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, pages 191–206, 2002.

[18] J. Leyden. Phatbot arrest throws open trade in zombie PCs. `http://www.theregister.co.uk/2004/05/12/phatbot_zombie_trade`, May 2004.

[19] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the USENIX Annual Technical Conference*, pages 335–348, June 2008.

[20] A. Marinescu. Windows Vista heap management enhancements. In *BlackHat US*, 2006.

[21] Microsoft Corporation. Data execution prevention. `http://technet.microsoft.com/en-us/library/cc738483.aspx`, 2003.

[22] Microsoft Corporation. Microsoft Security Bulletin MS07-017. `http://www.microsoft.com/technet/security/Bulletin/MS07-017.mspx`, Apr. 2007.

[23] Microsoft Corporation. Microsoft Security Advisory (961051). `http://www.microsoft.com/technet/security/advisory/961051.mspx`, Dec. 2008.

[24] Mozilla Developer Center. Crash reporting page. `https://developer.mozilla.org/En/Crash_reporting`, 2008.

[25] Mozilla Security Group. Mozilla security bug bounty program. `http://www.mozilla.org/security/bug-bounty.html`, 2004.

[26] Multi-State Information Sharing and Analysis Center. Vulnerability in Adobe Reader and Adobe Acrobat could allow remote code execution. `http://www.msisac.org/advisories/2009/2009-008.cfm`, Feb. 2009.

[27] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[28] J. D. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[29] P. P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the International Symposium on Memory Management*, pages 20–25, 1998.

[30] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, pages 87–106, 2007.

[31] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.

[32] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, Nov. 2008.

[33] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX conference on System administration*, pages 229–238, 1999.

[34] Samy. The Samy worm. `http://namb.la/popular/`, Oct. 2005.

[35] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002.

[36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the Conference on Computer and Communications Security*, pages 298–307, 2004.

[37] Skape and Skywing. Bypassing windows hardware-enforced DEP. *Uninformed Journal*, 2(4), Sept. 2005.

[38] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. `http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory/_iframe.html.php`, 2004.

[39] SkyLined. Personal communication, 2009.

[40] Sophos Inc. Stopping zombies, botnets and other email- and web-borne threats. `http://blogs.piercelaw.edu/tradesecretsblog/SophosZombies072507.pdf`, 12 2006.

[41] A. Sotirov. Heap feng shui in JavaScript. In *Proceedings of Blackhat Europe*, 2007.

[42] A. Sotirov and M. Dowd. Bypassing browser memory pro-

tections. In *Proceedings of BlackHat*, 2008.

[43] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[44] R. van den Heetkamp. Heap spraying. `http://www.0x000000.com/index.php?i=412&bin=110011100`, Aug. 2007.

[45] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2006)*, Feb. 2006.

Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense

Adam Barth
UC Berkeley
abarth@eecs.berkeley.edu

Joel Weinberger
UC Berkeley
jww@cs.berkeley.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

Abstract

We identify a class of Web browser implementation vulnerabilities, *cross-origin JavaScript capability leaks*, which occur when the browser leaks a JavaScript pointer from one security origin to another. We devise an algorithm for detecting these vulnerabilities by monitoring the "points-to" relation of the JavaScript heap. Our algorithm finds a number of new vulnerabilities in the open-source WebKit browser engine used by Safari. We propose an approach to mitigate this class of vulnerabilities by adding access control checks to browser JavaScript engines. These access control checks are backwards-compatible because they do not alter semantics of the Web platform. Through an application of the inline cache, we implement these checks with an overhead of 1–2% on industry-standard benchmarks.

1 Introduction

In this paper, we identify a class of Web browser implementation vulnerabilities, which we refer to as *cross-origin JavaScript capabilities leaks*, and develop systematic techniques for detecting, exploiting, and defending against these vulnerabilities. An attacker who exploits a cross-origin JavaScript capability leak can inject a malicious script into an honest Web site's security origin. These attacks are more severe than cross-site scripting (XSS) attacks because they affect all Web sites, including those free of XSS vulnerabilities. Once an attacker can run script in an arbitrary security origin, the attacker can, for example, issue transactions on the user's bank account, regardless of any SSL encryption, cross-site scripting filter, or Web application firewall.

We observe that these cross-origin JavaScript capability leaks are caused by an architectural flaw shared by most modern Web browsers: the Document Object Model (DOM) and the JavaScript engine enforce the same-origin policy using two different security models. The DOM uses an access control model, whereas the JavaScript engine uses object-capabilities.

- **Access Control.** The DOM enforces the same-origin policy using a reference monitor that prevents one Web site from accessing resources allocated to another Web site. For example, whenever a script attempts to access the cookie database, the DOM checks whether the script's security origin has sufficient privileges to access the cookies.
- **Object-Capabilities.** The JavaScript engine enforces the same-origin policy using an object-capability discipline that prevents one Web site from obtaining JavaScript pointers to sensitive objects that belong to a foreign security origin. Without JavaScript pointers to sensitive objects in foreign security origins, malicious scripts are unable to interfere with those objects.

Most modern Web browsers, including Internet Explorer, Firefox, Safari, Google Chrome, and Opera, use this design. However, the design's mismatch in enforcement paradigms leads to vulnerabilities whenever the browser leaks a JavaScript pointer from one security origin to another. Once a malicious script gets a JavaScript pointer to an honest JavaScript object, the attacker can leverage the object-capability security model of the JavaScript engine to escalate its DOM privileges. With escalated DOM privileges, the attacker can completely compromise the honest security origin by injecting a malicious script into the honest security origin.

To study this class of vulnerabilities, we devise an algorithm for detecting individual cross-origin JavaScript capability leaks. Using this algorithm, we uncover new instances of cross-origin JavaScript capability leaks in the WebKit browser engine used by Safari. We then illustrate how an attack can abuse these leaked JavaScript pointers by constructing proof-of-concept exploits. We propose defending against cross-origin JavaScript capability leaks by harmonizing the security models used by the DOM and the JavaScript engine.

- **Leak Detection.** We design an algorithm for automatically detecting cross-origin JavaScript capability leaks by monitoring the "points-to" relation among JavaScript objects in the heap. From this relation, we define the security origin of each JavaScript object by tracing its "prototype chain." We then search the graph for edges that connect objects in one security origin with objects in another security origin. These *suspicious edges* likely represent cross-origin JavaScript capability leaks.

- **Vulnerabilities and Exploitation.** We implement our leak detection algorithm and find two new high-severity cross-origin JavaScript capability leaks in WebKit. Although these vulnerabilities are implementation errors in WebKit, the presence of the bugs illustrates the fragility of the general architecture. (Other browsers have historically had similar vulnerabilities [17, 18, 19].) We detail these vulnerabilities and construct proof-of-concept exploits to demonstrate how an attacker can leverage a leaked JavaScript pointer to inject a malicious script into an honest security origin.
- **Defense.** We propose that browser vendors proactively defend against cross-origin JavaScript capability leaks by implementing access control checks throughout the JavaScript engine instead of reactively plugging each leak. Adding access control checks to the JavaScript engine addresses the root cause of these vulnerabilities (the mismatch between the security models used by the DOM and by the JavaScript engine) and provides defense-in-depth in the sense that both an object-capability and an access control failure are required to create an exploitable vulnerability. This defense is perfectly backwards-compatible because these access checks do not alter the semantics of the Web platform. Our implementation of these access control checks in WebKit incurs an overhead of only 1–2% on industry-standard benchmarks.

Contributions. We make the following contributions:

- We identify a class of Web browser implementation vulnerabilities: cross-origin JavaScript capability leaks. These vulnerabilities arise when the browser leaks a JavaScript pointer from one security origin to another security origin.
- We introduce an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the "points-to" relation of the JavaScript heap. Our algorithm uses a graph-based definition of the security origin of a JavaScript object.
- We reveal cross-origin JavaScript capability leaks and demonstrate techniques for exploiting these vulnerabilities. These exploits rely on the mismatch between the DOM's access control security model and the JavaScript engine's object-capability security model.
- We propose that browsers defend against cross-origin JavaScript capability leaks by implementing access control checks in the JavaScript engine. This defense is perfectly backwards-compatible and achieves a low overhead of 1–2%.

Organization. This paper is organized as follows. Section 2 identifies cross-origin JavaScript capability leaks as a class of vulnerabilities. Section 3 presents our algorithm for detecting cross-origin JavaScript capability leaks. Section 4 details the individual vulnerabilities we uncover with our algorithm and outlines techniques for exploiting these vulnerabilities. Section 5 proposes defending against cross-origin JavaScript capability leaks by adding access control checks to the JavaScript engine. Section 6 relates our work to the literature. Section 7 concludes.

2 JavaScript Capability Leaks

In this section, we describe our interpretation of JavaScript pointers as object-capabilities and identify cross-origin JavaScript capability leaks as a class of implementation vulnerabilities in browsers. We then sketch how these vulnerabilities are exploited and the consequences of a successful exploit.

2.1 Object-Capabilities

In modern Web browsers, the JavaScript engine enforces the browser's same-origin policy using an object-capability discipline: a script can obtain pointers only to JavaScript objects created by documents in its security origin. A script can obtain JavaScript pointers to JavaScript objects either by accessing properties of JavaScript object to which the script already has a JavaScript pointer or by conjuring certain built-in objects such as the global object and `Object.prototype` [14]. As in other object-capability systems, the ability to influence an object is tied to the ability to designate the object. In browsers, a script can manipulate a JavaScript object only if the script has a pointer to the object. Without a pointer to an object in a foreign security origin, a malicious script cannot influence honest JavaScript objects and cannot interfere with honest security origins.

One exception to this object-capability discipline is the JavaScript global object. According to the HTML 5 specification [10], the global object (also known as the `window` object) is visible to foreign security origins. There are a number of APIs for obtaining pointers to global objects from foreign security origins. For example, the `contentWindow` property of an `<iframe>` element is the global object of the document contained in the frame. Unlike most JavaScript objects, the global object is also a DOM object (called `window`) and is equipped with a reference monitor that prevents scripts in foreign security origins from getting or setting arbitrary properties of the object. This reference monitor does not forbid all accesses because some are desirable. For example, the `postMessage` method [10] is exposed across origins to facilitate mashups [1]. These exposed properties complicate the enforcement of the same-origin policy, which can lead to vulnerabilities.

2.2 Capability Leaks

Browsers occasionally contain bugs that leak JavaScript pointers from one security origin to another. These vulnerabilities are easy for developers to introduce into browsers because the DOM contains pointers to JavaScript objects in multiple security origins and developers can easily select the wrong pointer to disclose to a script. We identify these vulnerabilities as a class, which we call *cross-origin JavaScript capabilities leaks*, because they follow a common pattern. Identifying this class lets us analyze the concepts common to these vulnerabilities in all browsers.

The JavaScript language makes pointer leaks particularly devastating for security because JavaScript objects inherit many of their properties from a *prototype* object. When a script accesses a property of an object, the JavaScript engine uses the following algorithm to look up the property:

- If the object has the property, return its value.
- Otherwise, look up the property on the object's prototype (designated by the current object's `__proto__` property).

These prototype objects, in turn, inherit many of their properties from their prototypes in a chain that leads back to the `Object.prototype` object, whose `__proto__` property is `null`. All the objects associated with a given document have a prototype chain that leads back to that document's `Object.prototype` object. Given a JavaScript pointer to an object, a script can traverse this prototype chain by accessing the object's `__proto__` property. In particular, if an attacker obtains a pointer to an honest object, the attacker can obtain a pointer to the honest document's `Object.prototype` object and can influence the behavior of all the other JavaScript objects associated with the honest document.

2.3 Laundries

Once the attacker has obtained a pointer to the `Object.prototype` of an honest document, the attacker has several avenues for compromising the honest security origin. One approach is to abuse powerful functions reachable from `Object.prototype`, which we refer to as *laundries* because they let the attacker "wash away" his or her agency (analogous to laundering money). These functions often call one or more DOM APIs, letting the attacker call these APIs indirectly. Because these functions are defined by the honest document, the DOM's reference monitor allows the access [10]. However, if the attacker calls these functions with unexpected arguments, the functions might become confused deputies [9] and inadvertently perform the attacker's misdeeds.

Most Web sites contains innumerable laundries. We illustrate how an attacker can abuse a laundry by examining a representative laundry from the Prototype JavaScript library [22]: `invoke`. The `invoke` method is used to call a method, specified by name, on each object contained in an array. The attacker can use this function to trick the honest page into calling a universal DOM method, such as `setTimeout`. Suppose the attacker has a JavaScript pointer to an array named `honest_array` from an honest document that uses the Prototype library (for how this might occur, see Section 4.3) and that `honest_window` is the honest document's global object. The attacker can inject a malicious script into the honest security origin as follows:

```
honest_array.push(honest_window);
honest_array.invoke("setTimeout",
    "... malicious script ...", 0);
```

The attacker first adds the `honest_window` object to the array and then asks the honest principal to call the `setTimeout` method of the `honest_window`. When the JavaScript engine attempts to call the `setTimeout` DOM API, the DOM permits the call because the honest `invoke` method (acting as a confused deputy) issued the call. The DOM then runs the malicious script supplied by the attacker in the honest security origin.

2.4 Consequences

Once the attacker is able to run a malicious script in the honest security origin, all the browser's cross-origin security protections evaporate. The situation is as if every Web site contained a cross-site scripting vulnerability: the attacker can steal the user's authentication cookie or password, learn confidential information present on the Web site (e.g., read email messages on a webmail site), and issue transactions on behalf of the user (e.g., transfer money out of the user's bank account). Because these cross-origin JavaScript capability leaks are browser vulnerabilities, there is little a Web site can do to defend itself against these attacks.

3 JavaScript Capability Leak Detection

In this section, we describe the design and implementation of an algorithm for detecting cross-origin JavaScript capability leaks. Although the algorithm has a modest overhead, our instrumented browser performs comparably to Safari 3.1, letting us analyze complex Web applications.

3.1 Design

Assigning Security Origins. To detect cross-origin JavaScript capability leaks, we monitor the *heap graph*, the "points-to" relation between JavaScript objects in the

JavaScript heap (see Section 3.2 for details about the "points-to" relation). We annotate each JavaScript object in the heap graph with a security origin indicating which security origin "owns" the object. We compute the security origin of each object directly from the "is-prototype-of" relation in the heap graph using the following algorithm:

1. Let `obj` be the JavaScript object in question.
2. If `obj` was created with a non-`null` prototype, assign `obj` the same origin as its prototype.
3. Otherwise, `obj` must be the object prototype for some document d. In that case, assign `obj` the security origin of d (i.e., the scheme, host, and port of that d's URL).

This algorithm is unambiguous because, when created, each JavaScript object has a unique prototype, identified by its `__proto__` property. Although an object's `__proto__` can change over time, we fix the security origin of an object at creation-time.

Minimal Capabilities. This algorithm for assigning security origins to objects is well-suited to analyzing leaks of JavaScript pointers for two reasons. First, the algorithm is defined largely without reference to the DOM, letting us catch bugs in the DOM. Second, the algorithm reflects an object-capability perspective in that each JavaScript object is a strictly more powerful object-capability than the `Object.prototype` object that terminates its prototype chain. An attacker with a JavaScript pointer to the object can follow the object's prototype chain by repeatedly dereferencing the object's `__proto__` property and eventually obtain a JavaScript pointer to the `Object.prototype` object. In these terms, we view the `Object.prototype` object as the "minimal object-capability" of an origin.

Suspicious Edges. After annotating the heap graph with the security origin of each object, we detect a leaked JavaScript pointer as an edge from an object in one security origin to an object in another security origin. These *suspicious edges* represent failures of the JavaScript engine to segregate JavaScript objects into distinct security origins. Not all of these suspicious edges are actually security vulnerabilities because the HTML specification requires some JavaScript objects, such as the global object, be visible to foreign security origins. To prevent exploits, browsers equip these objects with a reference monitor that prevents foreign security origins from getting or setting arbitrary properties of the object. In addition to the global object, a handful of other JavaScript objects required to be visible to foreign security origins. These objects are annotated in WebKit's Interface Description Language (IDL) with the attribute `DoNotCheckDomainSecurity`.

3.2 The "Points-To" Relation

In our heap graph, we include two kinds of points in the "points-to" relation: explicit pointers that are stored as properties of JavaScript objects and implicit pointers that are stored internally by the JavaScript engine.

Explicit Pointers. A script can alter the properties of an object using the `get`, `set`, and `delete` operations.

- `get` looks up the value of an object property.
- `set` alters the value of an object property.
- `delete` removes a property from an object.

To monitor the "points-to" relation between JavaScript objects in the JavaScript heap, we instrument the `set` operation. Whenever the JavaScript engine invokes the `set` operation to store a JavaScript object in a property of another JavaScript object, we add an edge between the two objects in our representation of the heap graph. If the `set` operation overwrites an existing property, we remove the obsolete edge from the graph. To improve performance, we ignore JavaScript values because JavaScript values cannot hold JavaScript pointers and therefore are leaves in the heap graph. We remove JavaScript objects from the heap graph when the objects are deleted by the JavaScript garbage collector.

Implicit Pointers. The above instrumentation does not give us a complete picture of the "points-to" relation in the JavaScript heap because the operational semantics of the JavaScript language [14] rely on a number of implicit JavaScript pointers, which are not represented explicitly as properties of a JavaScript object. For example, consider the following script:

```
var x = ...
function f() {
  var y = ...
  function g() {
    var z = ...
    function h() { ... }
  }
}
```

Function `h` can obtain the JavaScript pointers stored in variables `x`, `y`, and `z` even though there are no JavaScript pointers between `h` and these objects. The function `h` can obtain these JavaScript pointers because the algorithm for resolving variable names makes use of an implicit "next" pointer that connects `h`'s scope object to the scope objects of `g`, `f`, and the global scope. Instead of being stored as properties of JavaScript objects, these implicit pointers are stored as member variables of native objects in the JavaScript engine. To improve the completeness of our heap graph, we include these implicit JavaScript pointers explicitly as edges between the JavaScript scope objects.

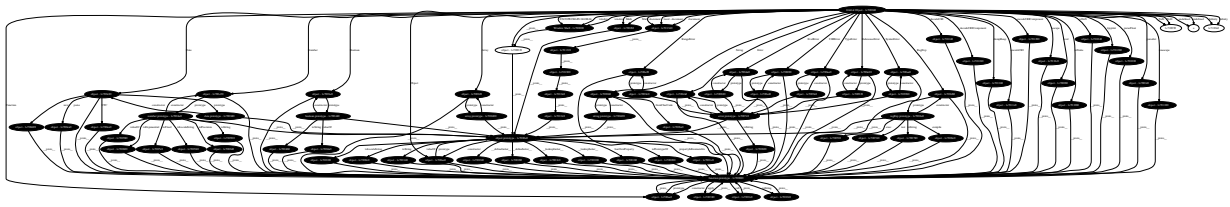

Figure 1: The heap graph of an empty document.

3.3 Implementation

We implemented our leak detection algorithm in a 1,393 line patch to WebKit's Nitro JavaScript engine. Our algorithm can construct heap graphs of complex Web applications, such as Gmail or the Apple Store. For example, one heap graph of a Gmail inbox contains 54,140 nodes and 130,995 edges. These graphs are often visually complex and difficult to interpret manually. Figure 1 illustrates the nature of these graphs by depicting the heap graph of an empty document. Although our instrumentation slows down the browser, the instrumented browser is still faster than Safari 3.1, demonstrating that our algorithm scales to complex Web applications.

4 Vulnerabilities and Exploitation

In this section, we use our leak detector to detect cross-origin JavaScript capability leaks in WebKit. After discovering two new vulnerabilities, we illuminate the vulnerabilities by constructing proof-of-concept exploits using three different techniques. In addition, we apply our understanding of JavaScript pointers to breaking the Subspace [11] mashup design.

4.1 Test Suite

To find example cross-origin JavaScript capability leaks, we run our instrumented browser through a test suite. Ideally, to reduce the number of false negatives, we would use a test suite with high coverage. Because our goal is to find example vulnerabilities, we use the WebKit project's regression test suite. This test suite exercises a variety of browser security features and tests for the non-existence of past security vulnerabilities. Using this test suite, our instrumentation found two new high-severity cross-origin JavaScript capability leaks. Instead of attempting to discover and patch all these leaks, we recommend a more comprehensive defense, detailed in Section 5.

WebKit's regression test suite uses a JavaScript object named `layoutTestController` to facilitate its tests. For example, each tests notifies the testing harness that the test is complete by calling the `notifyDone` method of the `layoutTestController`. We modified this `notifyDone` method to store the JavaScript heap graph in the file system after each test completes.

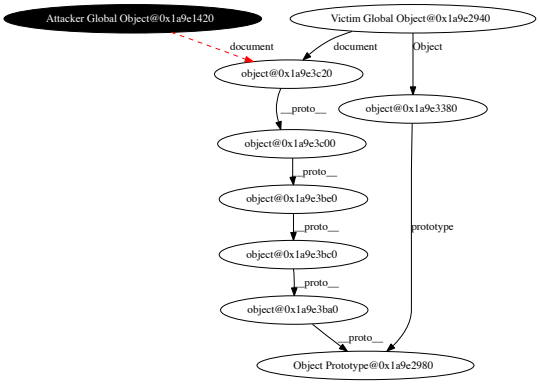

Figure 2: Selected nodes from a heap graph showing a cross-origin JavaScript capability leak of the document object, `object@0x1a9e3c20`, after a navigation.

The `layoutTestController` contains a number of objects that are shared between all security origins. Our instrumentation flags JavaScript pointers to these objects as suspicious, and, in fact, these pointers are exploitable in the test configuration of the browser. However, these pointers are not present in the release configuration of the browser because the `layoutTestController` itself is present only during testing. We white listed these objects as visible to multiple security origins.

4.2 Navigation and Document

Vulnerability. When the browser navigates a window from one Web page to another, the browser replaces the document originally displayed in the window with a new document retrieved from the network. Our instrumentation found that WebKit leaks a JavaScript pointer to the new document object every time a window navigates because the DOM updates the `document` property of the old global object to point to the new document occupying the frame. This leak is visible in the heap graph (see Figure 2) as a dashed line from `Attacker Global Object@0x1a9e1420` to the honest document object, `object@0x1a9e3c20`.

Exploit. Crafting an exploit for this vulnerability is subtle. An attacker cannot simply hold a JavaScript pointer to the old global object and access its `document` property because all JavaScript pointers to global objects are updated to the new global object when a frame is navigated navigation [10]. However, the properties of the old global object are still visible to functions defined by the old document via the scope chain as global variables. In particular, an attacker can exploit this vulnerability as follows:

1. Create an `<iframe>` to `http://attacker.com/iframe.html`, which defines the following function in a malicious document:

   ```
   function exploit() {
     var elmt = document.
         createElement("script");
     elmt.src =
         "http://attacker.com/atk.js";
     document.body.appendChild(elmt);
   }
   ```

   Notice that the `exploit` function refers to the document as a global variable, `document`, and not as a property of the global object, `window.document`.
2. In the parent frame, store a pointer to the `exploit` function by running the following JavaScript:

   ```
   window.f = frames[0].exploit;
   ```
3. Navigate the frame to `http://example.com/`.
4. Call the function: `window.f()`.

After the attacker navigates the child frame to `http://example.com/`, the DOM changes the `document` variable in the function `exploit` to point to the honest document object instead of the attacker's document object. The `exploit` function can inject arbitrary script into the honest document using a number of standard DOM APIs. Once the attacker has injected script into the honest document, the attacker can impersonate the honest security origin to the browser.

4.3 Lazy Location and History

Vulnerability. For performance, WebKit instantiates the `window.location` and `window.history` objects lazily the first time they are accessed. When instantiating these objects, the browser constructs their prototype chains. In some situations, WebKit constructs an incorrect prototype chain that connects these objects to the `Object.prototype` of a foreign security origin, creating a vulnerability if, for example, a document uses the following script to "frame bust" [12] in order to avoid clickjacking [7] attacks:

```
top.location.href =
    "http://example.com/";
```

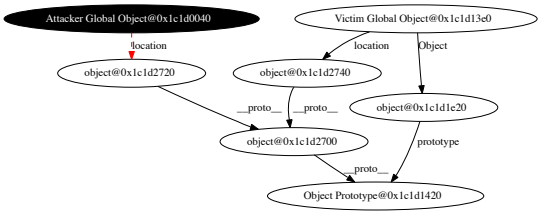

Figure 3: Selected nodes from a heap graph showing a cross-origin JavaScript capability leak of the location prototype, `object@0x1c1d2700`, to the attacker after the victim attempts to frame bust.

This line of JavaScript changes the location of the topmost frame, navigating that frame to a trusted Web site. The browser permits cross-origin access to a frame's location object to allow navigation [1]. If this script is the first script to access the location object of the top frame, then WebKit will mistakenly connect the top frame's newly constructed location object to the `Object.prototype` of the child frame (instead of to the `Object.prototype` of the top frame) because the child frame is currently in scope lexically.

Exploit. To exploit this cross-origin JavaScript capability leak, the attacker proceeds in two phases: (1) the attacker obtains a JavaScript pointer to the honest `Object.prototype`, and (2) the attacker abuses the honest `Object.prototype` to inject a malicious script into the honest security origin. To obtain a JavaScript pointer to the honest `Object.prototype`, the attacker create an `<iframe>` to an honest document that frame busts and runs the following script in response to the `beforeunload` event:

```
var location_prototype =
    window.location.__proto__;
var honest_object_prototype =
    location_prototype.__proto__;
```

Because the `beforeunload` event handler runs after the child frame has attempted to frame bust, the attacker's location object has been instantiated by the honest document and is mistakenly attached to the honest `Object.prototype` (see Figure 3). The attacker obtains a pointer to the honest `Object.prototype` by traversing this prototype chain.

Once the attacker has obtained a JavaScript pointer to the honest `Object.prototype`, there are a number of techniques the attacker can use to compromise the honest security origin completely. We describe two representative examples:

1. Many Web sites use JavaScript libraries to smooth over incompatibilities between browsers and reuse

common code. One of the more popular JavaScript libraries is the Prototype library [22], which is used by CNN, Apple, Yelp, Digg, Twitter, and many others. If the honest page uses the Prototype library, the attacker can inject arbitrary script into the honest page by abusing the powerful `invoke` function defined by the Prototype library. For example, the attacker can use the follow script:

```
var honest_function =
    honest_object_prototype.
    __defineGetter__;
var honest_array =
    honest_function.
    argumentNames();
honest_array.push(frames[0]);
honest_array.invoke("setTimeout",
    "... malicious script ...");
```

In the Prototype library, arrays contain a method named `invoke` that calls the method named by its first argument on each element of its array, passing the remaining arguments to the invoked method. To abuse this method, the attacker first obtains a pointer to an honest array object by calling the `argumentNames` method of an honest function reachable from the `honest_object_prototype` object. The attacker then pushes the global object of the child frame onto the array and calls the honest document's `setTimeout` method via `invoke`. The honest global object has a reference monitor that prevents the attacker from accessing `setTimeout` directly, but the reference monitor allows `invoke` to access `setTimeout` because `invoke` is defined by the honest document.

2. Even if the honest Web page does not use a complex JavaScript library, the attacker can often find a snippet of honest script to trick. For example, suppose the attacker installs a "setter" function for the `foo` property of the honest `Object.prototype` as follows:

```
function evil(x) {
  x.innerHTML =
      '<img src="about:blank"' +
      ' onerror="... script ...">';
});
honest_object_prototype.
    __defineSetter__('foo', evil);
```

Now, if the honest script stores a DOM node in a property of an object as follows:

```
var obj = new Object();
obj.foo = honest_dom_node;
```

> The JavaScript engine will call the attacker's setter function instead of storing `honest_dom_node` into the `foo` property of `obj`, causing the variable `x` to contain a JavaScript pointer to `honest_dom_node`. Once the attacker's function is called with a pointer to the honest DOM node, the attacker can inject malicious script into the honest document using the `innerHTML` API.

4.4 Capability Leaks in Subspace

The Subspace mashup design [11] lets a trusted integrator communicate with an untrusted gadget by passing a JavaScript pointer from the integrator to the gadget:

> A Subspace JavaScript object is created in the top frame and passed to the mediator frame... The mediator frame still has access to the Subspace object it obtained from the top frame, and passes this object to the untrusted frame. [11]

Unfortunately, the Subspace design relies on leaking a JavaScript pointer from a trusted security origin to an untrusted security origin, creating a cross-origin JavaScript capability leak. By leaking the communication object, Subspace also leaks a pointer to the trusted `Object.prototype` via the prototype chain of the communication object.

To verify this attack, we examined CrossSafe [25], a public implementation of Subspace. We ran a CrossSafe tutorial in our instrumented browser and examined the resulting heap graph. Our detector found a cross-origin JavaScript capability leak: the `channel` object is leaked from the integrator to the gadget. By repeatedly dereferencing the `__proto__` property, the untrusted gadget can obtain a JavaScript pointer to the trusted `Object.prototype` object. The untrusted gadget can then inject a malicious script into the trusted integrator using one of the techniques described in Section 4.3.

5 Defense

In this section, we propose a principled defense for cross-origin JavaScript capability leaks. Our defense addresses the root cause of these vulnerabilities and incurs a minor performance overhead.

5.1 Approach

Currently, browser vendors defend against cross-origin JavaScript capability leaks by patching each individual leak after the leak is discovered. We recommend another approach for defending against these vulnerabilities: add access control checks throughout the JavaScript engine. We recommend this principled approach over ad-hoc leak plugging for two reasons:

- This approach addresses the core design issue underlying cross-origin JavaScript capability leak vulnerabilities: the mismatch between the DOM's access control security model and the JavaScript engine's object-capability security model.
- This approach provides a second layer of defense: if the browser is leak-free, all the access control checks will be redundant and pass, but if the browser contains a leak, the access control checks prevent the attacker from exploiting the leak.

In a correctly implemented browser, Web content will be unable to determine whether the browser implements the access control checks we recommend. The additional access control checks enhance the mechanism used to enforce the same-origin policy but do not alter the policy itself, resulting in zero compatibility impact.

Another plausible approach to mitigating these vulnerabilities is to adopt an object-capability discipline throughout the DOM. This approach mitigates the severity of cross-origin JavaScript capability leaks by limiting the damage an attacker can wreak with the leaked capability. For example, if the browser leaks an honest history object to the attacker, the attacker would be able to manipulate the history object, but would not be able to alter the document object. Conceptually, either adding access control checks to the JavaScript engine or adopting an object-capability discipline throughout the DOM resolves the underlying architectural security issue, but we recommend adopting the access control paradigm for two main reasons:

- Adopting an object-capability discipline throughout the DOM requires "taming" [15] the DOM API. The current DOM API imbues every DOM node with the full authority of the node's security origin because the API exposes a number of "universal" methods, such as `innerHTML` that can be used to run arbitrary script. Other researchers have designed capability-based DOM APIs [4], but taming the DOM API requires a number of non-backwards compatible changes. A browser that makes these changes will be unpopular because the browser will be unable to display a large fraction of Web sites.
- The JavaScript language itself has a number of features that make enforcing an object-capability discipline challenging. For example, every JavaScript object has a prototype chain that eventually leads back to the `Object.prototype`, making it difficult to create a weaker object-capability than the `Object.prototype`. Unfortunately, the `Object.prototype` itself represents a powerful object-capability with the ability to interfere with the properties of every other object from the same document (e.g., the exploit in Section 4.3.)

Although we recommend that browsers adopt the access control paradigm for Web content, other projects, such as Caja [16] and ADsafe [3], take the opposite approach and elect to enforce an object-capability discipline on the DOM. These projects succeed with this approach because the preceding considerations do not apply: these projects target new code (freeing themselves from backwards compatibility constraints) that is written in a subset of JavaScript (freeing themselves from problematic language features). For further discussion, see Section 6.

5.2 Design

We propose adding access control checks to the JavaScript engine by inserting a reference monitor into each JavaScript object. The reference monitor interposes on each `get` and `set` operation (described in Section 3) and performs the following access control check:

1. Let the *active origin* be the origin of the document that defined the currently executing script.
2. Let the *target origin* be the origin that "owns" the JavaScript object being accessed, as computed by the algorithm in Section 3.1.
3. Allow the access if the browser considers the active origin and the target origin to be the same origin (i.e., if their scheme, hosts, and ports match).
4. Otherwise, deny access.

If the access is denied, the JavaScript engine returns the value `undefined` for `get` operations and simply ignores `set` operations. In addition to adding these access control checks, we record the security origin of each JavaScript object when the object is created. Our implementation does not currently insert access control checks for `delete` operations, but these checks could be added at a minor performance cost. Some JavaScript objects, such as the global object, are visible across origins. For these objects, our reference monitor defers to the reference monitors that already protect these objects.

5.3 Inline Cache

The main disadvantage of performing an access control check for every JavaScript property access is the runtime overhead of performing the checks. Sophisticated Web applications access JavaScript properties an enormous number of times per second, and browser vendors have heavily optimized these code paths. However, we observe that the proposed access checks are largely redundant and amenable to optimization because scripts virtually always access objects from the same origin.

Cutting-edge JavaScript engines, including Safari 4's Nitro JavaScript Engine, Google Chrome's V8 JavaScript engine, Firefox 3.5's TraceMonkey JavaScript engine, and Opera 11's Carakan JavaScript engine, optimize JavaScript property accesses using an

inline cache [24]. (Of the major browser vendors, only Microsoft has yet to announce plans to implement this optimization.) These JavaScript engines group together JavaScript objects with the same "structure" (i.e., whose properties are laid out the same order in memory). When a script accesses a property of an object, the engine caches the object's group and the memory offset of the property inline in the compiled script. The next time that compiled script accesses a property of an object, the inline cache checks whether the current object has the same structure as the original object. If the two objects have the same structure, a *cache hit*, the engine uses the memory offset stored in the cache to access the property. Otherwise, a *cache miss*, the engine accesses the property using the normal algorithm.

Notice that two objects share the same structure only if their prototypes share the same structure. Additionally, the Nitro JavaScript engine initializes each `Object.prototype` with a unique structure identifier, preventing two object from different security origins (as defined by our prototype-based algorithm) from being be grouped together as sharing the same structure. (Other JavaScript engines, such as V8, do contain structure groups that span security origins, but this design is not necessary for performance.) Whenever the inline cache has a hit, we observe the following:

- The current object is from the same security origin as the original object that created the cache entry because the two objects share the same structure.
- The script has the same security origin as when the cache entry was created because the cache is inlined into the script and the security origin of the script is fixed at compile time.

Taken together, these properties imply that the current access control check will return the same result as the original check because both of the origins involved in the check are unchanged. Therefore, we need not perform an access control check during a cache hit, greatly reducing the performance overhead of adding access control checks to the JavaScript engine.

5.4 Evaluation

To evaluate performance overhead of our defense, we added access control checks to Safari 4's Nitro JavaScript engine in a 394 line patch. We verified that our access control checks actually defeat the proof-of-concept exploits we construct in Section 4. To speed up the access control checks, we represented each security origin by a pointer, letting us allow the vast majority of accesses using a simple pointer comparison. In some rare cases, including to deny access, our implementation performs a more involved access check. The majority of performance overhead in our implementation is caused by computing the currently active origin from the lexical scope, which can be reduced with further engineering.

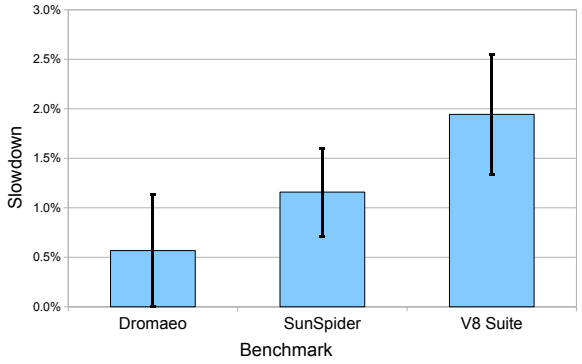

Figure 4: Overhead for access control checks as measure by industry-standard JavaScript benchmarks (average of 10 runs, 95% confidence).

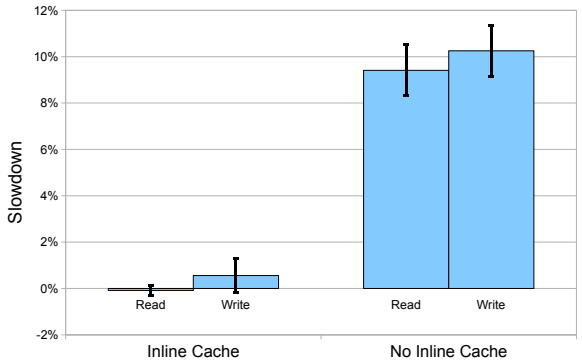

Figure 5: Overhead for reading and writing properties of JavaScript objects both with and without an inline cache as measured by microbenchmarks (average of 10 runs, 95% confidence).

Overall Performance. Our implementation incurs a small overhead on industry-standard JavaScript benchmarks (see Figure 4). On Mozilla's Dromaeo benchmark, we observed a 0.57% slowdown for access control versus an unmodified browser (average of 10 runs, ±0.58%, 95% confidence). On Apple's SunSpider benchmark, we observed a 1.16% slowdown (average of 10 runs, ±0.45%, 95% confidence). On Google's V8 benchmark, we observed a 1.94% slowdown (average of 10 runs, ±0.61, 95% confidence). We hypothesize that the variation in slowdown between these benchmarks is due to the differing balance between arithmetic operations and property accesses in the different benchmarks. Note that these overhead numbers are tiny in comparison with the 338% speedup of Safari 4 over Safari 3.1 [24].

Benefits of Inline Cache. We attribute much of the performance of our access checks to the inline cache, which lets our implementation skip redundant access control checks for repeated property accesses. To evaluate the performance benefits of the inline cache, we created two microbenchmarks, "read" and "write." In the read benchmark, we repeatedly performed a `get` operation on one property of a JavaScript object in a loop. In the write benchmark, we repeatedly performed a `set` operation on one property of a JavaScript object in a loop. We then measured the slowdown incurred by the access control checks both with the inline cache enabled and with the inline cache disabled (see Figure 5). With the inline cache enabled, we observed a -0.08% slowdown (average of 50 runs, $\pm 0.22\%$, 95% confidence) on the read benchmark and a 0.55% slowdown (average of 50 runs, $\pm 0.74\%$, 95% confidence) on the write benchmark. By contrast, with the inline cache disabled, we observed a 9.41% slowdown (average of 50 runs, $\pm 1.11\%$, 95% confidence) on the read benchmark and a 10.25% slowdown (average of 50 runs, $\pm 1.00\%$, 95% confidence) on the write benchmark.

From these observations we conclude that browser vendors can implement access control checks for every `get` and `set` operation with a performance overhead of less than 1–2%. To reap these security benefits with minimal overhead, the JavaScript engine should employ an inline cache to optimize repeated property accesses, and the inline cache should group structurally similar JavaScript objects only if those objects are from the same security origin.

6 Related Work

The operating system literature has a rich history of work on access control and object-capability systems [13, 21, 23, 8]. In this section, we focus on comparing our work to related work on access control and object-capability systems in Web browsers.

FBJS, Caja, and ADsafe. Facebook, Yahoo!, and Google have developed JavaScript subsets, called FBJS [5], ADsafe [3], and Caja [16], respectively, that enforce an object-capability discipline by removing problematic JavaScript features (such as prototypes) and DOM APIs (such as `innerHTML`). These projects take the opposite approach from this paper: they extend the JavaScript engine's object-capability security model to the DOM instead of extending the DOM's access control security model to the JavaScript engine. These projects choose this alternative design point for two reasons: (1) the projects target new social networking gadgets and advertisements that are free from compatibility constraints and (2) these projects are unable to alter legacy browsers because they must work in existing browsers. We face the opposite constraints: we cannot alter legacy content but we can change the browser. For these reasons, we recommend the opposite design point.

Opus Palladianum. The Opus Palladianum (OP) Web browser [6] isolates security origins into separate sandboxed components. This component-based browser architecture makes it easier to reason about cross-origin JavaScript capability leaks because these capability leaks must occur between browser components instead of within a single JavaScript heap. We can view the sandbox as a coarse-grained reference monitor. Unfortunately, the sandbox alone is too coarse-grained to implement standard browser features such as `postMessage`. To support these features, the OP browser must allow inter-component references, but without a public implementation, we are unable to evaluate whether these inter-component references give rise to cross-origin JavaScript capability leaks.

Script Accenting. Script accenting [2] is a technique for adding defense-in-depth to the browser's enforcement of the same-origin policy. To mitigate mistaken script execution, the browser encrypts script source code with a key specific to the security origin of the script. Whenever the browser attempts to run a script in a security origin, the browser first decrypts the script with the security origin's key. If decryption fails, likely because of a vulnerability, the browser refuses to execute the script. Script accenting similarly encrypts the names of JavaScript properties ostensibly preventing a script from manipulating properties of objects from another origin. Unfortunately, this approach is not expressive enough to represent the same-origin policy (e.g., this design does not support `document.domain`). In addition, script accenting requires XOR encryption to achieve sufficient performance, but XOR encryption lacks the integrity protection required to make the scheme secure.

Cross-Origin Wrappers. Firefox 3 uses cross-origin wrappers [20] to mitigate security vulnerabilities caused by cross-origin JavaScript capability leaks. Instead of exposing JavaScript objects directly to foreign security origins, Firefox exposes a "wrapper" object that mediates access to the wrapped object with a reference monitor. Implementing cross-origin wrappers correctly is significantly more complex than implementing access control correctly because the cross-origin wrappers must wrap and unwrap objects at the appropriate times in addition to implementing all the same access control checks. Our access control design can be viewed as a high-performance technique for reducing this complexity (and the attendant bugs) by adding the reference monitor to every object.

7 Conclusions

In this paper, we identify a class of vulnerabilities, *cross-origin JavaScript capability leaks*, that arise when the browser leaks a JavaScript pointer from one security origin to another. These vulnerabilities undermine the same-origin policy and prevent Web sites from securing themselves against Web attackers. We present an algorithm for detecting cross-origin JavaScript capability leaks by monitoring the "points-to" relation between JavaScript objects in the JavaScript heap. We implement our detection algorithm in WebKit and use it to find new cross-origin JavaScript capability leaks by running the WebKit regression test suite in our instrumented browser. Having discovered these leaked pointers, we turn our attention to exploiting these vulnerabilities. We construct exploits to illustrate the vulnerabilities and find that the root cause of the these vulnerabilities is the mismatch in security models between the DOM, which uses access control, and the JavaScript engine, which uses object-capabilities. Instead of patching each leak, we recommend that browser vendors repair the underlying architectural issue by implementing access control checks throughout the JavaScript engine. Although a straight-forward implementation that performed these checks for every access would have a prohibitive overhead, we demonstrate that a JavaScript engine optimization, the inline cache, reduces this overhead to 1–2%.

Acknowledgements. We thank Chris Karloff, Oliver Hunt, Collin Jackson, John C. Mitchell, Rachel Parke-Houben, and Sam Weinig for their helpful suggestions and feedback. This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

[1] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[2] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 2–11, New York, NY, USA, 2007. ACM.

[3] Douglas Crockford. ADsafe.

[4] Douglas Crockford. ADsafe DOM API.

[5] Facebook. Facebook Markup Language (FBML).

[6] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[7] Jeremiah Grossman. Clickjacking: Web pages can see and hear you, October 2008.

[8] Norm Hardy. The keykos architecture. *Operating Systems Review*, 1985.

[9] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[10] Ian Hickson et al. HTML 5 Working Draft.

[11] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th International World Wide Web Conference. (WWW)*, 2007.

[12] Peter-Paul Koch. Frame busting, 2004. `http://www.quirksmode.org/js/framebust.html`.

[13] Butler Lampson. Protection and access control in operating systems. *Operating Systems: Infotech State of the Art Report*, 14:309–326, 1972.

[14] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Proceedings of the 6th Asian Programming Language Symposium (APLAS)*, December 2008.

[15] Mark Miller. A theory of taming.

[16] Mark Miller. Caja, 2007.

[17] Mitre. CVE-2008-4058.

[18] Mitre. CVE-2008-4059.

[19] Mitre. CVE-2008-5512.

[20] Mozilla. XPConnect wrappers. `http://developer.mozilla.org/en/docs/XPConnect_wrappers`.

[21] Sape J. Mullender, Guido van Rossum, Andrew Tannenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[22] Prototype JavaScript framework. `http://www.prototypejs.org/`.

[23] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: a fast capability system. In *17th ACM Symposium on Operating System Principles*, New York, NY, USA, 1999. ACM.

[24] Maciej Stachowiak. Introducing SquirrelFish Extreme, 2008.

[25] Kris Zyp. CrossSafe.