Improving Web Site Security with Data Flow Management

by

Alexander Siumann Yip

S.B., Computer Science and Engineering (2001)
M.Eng., Electrical Engineering and Computer Science (2002)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

© Massachusetts Institute of Technology 2009. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 21, 2009

Certified by . . . .
Robert T. Morris
Associate Professor
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Improving Web Site Security with Data Flow Management

by

Alexander Siumann Yip

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This dissertation describes two systems, Resin and BFlow, whose goal is to help Web developers build more secure Web sites. Resin and BFlow use data flow management to help reduce the security risks of using buggy or malicious code. Resin provides programmers with language-level mechanisms to track and manage the flow of data within the server. These mechanisms make it easy for programmers to catch server-side data flow bugs that result in security vulnerabilities, and prevent these bugs from being exploited. BFlow is a system that adds information flow control, a restrictive form of data flow management, both to the Web browser and to the interface between a browser and a server. BFlow makes it possible for a Web site to combine confidential data with untrusted JavaScript in its Web pages, without risking leaks of that data.

This work makes a number of contributions. Resin introduces the idea of a data flow assertion and demonstrates how to build them using three language-level mechanisms, policy objects, data tracking, and filter objects. We built prototype implementations of Resin in both the PHP and Python runtimes. We adapt seven real off-the-shelf applications and implement 11 different security policies in Resin which thwart at least 27 real security vulnerabilities. BFlow introduces an information flow control model that fits the JavaScript communication mechanisms, and a system that maps that model to JavaScript's existing isolation system. Together, these techniques allow untrusted JavaScript to read, compute with, and display confidential data without the risk of leaking that data, yet requires only minor changes to existing software. We built a prototype of the BFlow system and three different applications including a social networking application, a novel shared-data Web platform, and BFlogger, a third-party JavaScript platform similar to that of Blogger.com. We ported several untrusted JavaScript extensions from Blogger.com to BFlogger, and show that the extensions cannot leak data as they can in Blogger.com.

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Assistant Professor

Published Materials

Portions of Chapter 2 will appear in the publication [87]: Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, October 2009.

Portions of Chapter 3 appeared in the publication [86]: Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 233–246, Nuremberg, Germany, March 2009.

Acknowledgments

Many people contributed to the completion of this dissertation, including colleagues in PDOS and the systems community at MIT's CSAIL, as well as outside of the lab, at home and on campus. Although I cannot list them all, I will attempt to acknowledge these people here.

My advisers, both Robert Morris and Nickolai Zeldovich, were instrumental to this work. They taught me how to do research, think critically, be a graduate student, and teach effectively. Frans Kaashoek, Eddie Kohler, and Barbara Liskov also provided invaluable advice and guidance along the way.

My coauthors also contributed to this dissertation. Xi Wang made substantial contributions to the design and evaluation of RESIN, including new data flow assertions and performance enhancements, as well as the text in Chapter 2. Neha Narula made major contributions to the design and evaluation of BFLOW, in addition to the text in Chapter 3. Maxwell Krohn also contributed to the design of BFLOW and the earlier work in WikiCode and W5. Micah Brodsky, Petros Efstathopoulos, Steve VanDeBogart, and Michael Walfish also contributed to BFLOW through their contributions to WikiCode and W5.

Simply spending time in PDOS had an impact on me and this work. Sharing an office with Thomer M. Gil, Chris Lesniewski-Laas, Jinyang Li, Athicha Muthitacharoen, Jacob Strauss, and Jayashree Subramanian has been both entertaining and enlightening. Eating lunch with the likes of Silas Boyd-Wickizer, Benjie Chen, Russ Cox, Frank Dabek, Alex Pesterev, Jeremy Stribling, and company had a similar effect.

Lastly, this work would have never been completed without the consistent support and encouragement from my friends and family throughout the graduate school process. My parents, Laura Yip, Michelle Yip, Seanna Kim, and all my friends share credit for this work.

Contents

1 Introduction **17**

1.1 Resin 17
1.2 BFlow 18
1.3 Contributions 19
1.3.1 Resin 19
1.3.2 BFlow 19
1.4 Organization 20

2 Resin **21**

2.1 Introduction 21
2.2 Goals and Examples 23
2.2.1 Threat Model 26
2.3 Design 27
2.3.1 Design Overview 27
2.3.2 Filter Objects 29
2.3.3 Policy Objects 32
2.3.4 Data Tracking 33
2.4 Implementation 35
2.5 Applying Resin 36
2.5.1 Access Control Checks 36
2.5.2 Server-Side Script Injection 38
2.5.3 SQL Injection and Cross-Site Scripting 39
2.5.4 Other Attack Vectors 40
2.5.5 Application Integration 40
2.6 Security Evaluation 41
2.6.1 Programmer Effort 41
2.6.2 Preventing Vulnerabilities 43
2.6.3 Generality 44
2.7 Performance Evaluation 45
2.7.1 Application Performance 46
2.7.2 Microbenchmarks 46
2.8 Deployment 48
2.9 Limitations and Future Work 48
2.9.1 Data Flow Assertion Model 48
2.9.2 Language Runtimes 49

2.9.3 Applications 50
2.10 Related Work 50
2.10.1 Policy Specification 50
2.10.2 Data Tracking 52
2.11 Summary 53

3 BFlow **55**
3.1 Introduction 55
3.2 Background: JavaScript 57
3.3 Challenges 58
3.3.1 Threat Model and Security 58
3.3.2 Flexibility and Adoption 60
3.4 Design 60
3.4.1 Information Flow Control 61
3.4.2 Protection Zones 63
3.4.3 Controlling Intra-browser Communication 64
3.4.4 Controlling Browser-Server Communication 66
3.5 Visible Model 67
3.5.1 Developer Visible Model 68
3.5.2 Users Visible Model 70
3.6 Implementation 70
3.6.1 Client Implementation 70
3.6.2 User Authentication 71
3.6.3 Server Implementation 72
3.6.4 Server Storage 72
3.7 Applications 73
3.7.1 BF-Blogger 73
3.7.2 BF-Socialnet 74
3.7.3 W5 75
3.8 Evaluation 78
3.8.1 Security 78
3.8.2 Adoption 79
3.9 Deployment 81
3.10 Limitations and Future Work 81
3.10.1 Information Flow Control 81
3.10.2 User Interface and Understanding Labels 82
3.10.3 Applications 82
3.10.4 Out of Scope Attacks 83
3.10.5 Design Variations 83
3.11 Related Work 84
3.11.1 Discretionary Access Control 84
3.11.2 Mandatory Access Control 85
3.12 Summary 86

4 Integrating Resin and BFlow **87**

5 Conclusion **89**
5.1 Resin 89
5.2 BFlow 89
5.3 Summary 90

List of Figures

2-1 Overview of the HotCRP password data flow assertion. 27
2-2 Simplified PHP code for defining the HotCRP password policy class and annotating the password data. This policy only allows a password to be disclosed to the user's own email address or to the program chair. 28
2-3 Python code for the default filter for sockets. 31
2-4 Saving persistent policies to a SQL database for HotCRP passwords. Uses symbols from Figure 2-1. 34
2-5 Python code for a data flow assertion that checks read access control in MoinMoin. The *process_client* and *update_body* functions are simplified versions of MoinMoin equivalents. 37
2-6 Simplified PHP code for a data flow assertion that catches server-side script injection. In the actual implementation, *filter_read* verifies that each character in *$buf* has the *CodeApproval* policy. 38

3-1 Malicious JavaScript reads confidential data (a) via the DOM and (b) by exploiting vulnerable JavaScript. 59
3-2 After reading confidential data, the malicious JavaScript leaks confidential data to an adversary via the (a) adversary's server (b) Web site's public data. 59
3-3 BFlow overview. Untrusted protection zones are shaded. 61
3-4 Web page frame hierarchy with zones and labels. Each box is a frame. 65
3-5 W5 overview showing three applications. 76

List of Tables

2.1 Top CVE security vulnerabilities of 2008 [71]. 23
2.2 Top Web site vulnerabilities of 2007 [82]. 24
2.3 The RESIN API. A::B(args) denotes method B of an object of type A. Not shown is the API used by the programmer to specify and access filter objects for different data flow boundaries. 30
2.4 Results from using RESIN assertions to prevent previously-known and newly discovered vulnerabilities in several Web applications. 42
2.5 The average time taken to execute different operations in an unmodified PHP interpreter, a RESIN PHP interpreter without any policy, and a RESIN PHP interpreter with an empty policy. 47

3.1 Default IFC communication rules and declassification exceptions; zones S and R are untrusted. The prototype implements these rules for communication through `postMessageBF`, the FID channel and HTTP requests, but it is more restrictive than these rules for shared DOM variables and cookie communication across zones. 62
3.2 Lines of code (LOC) changed to port existing widgets to BF-Blogger and whether they see confidential data. 80

Chapter 1

Introduction

This dissertation addresses two security issues that affect Web sites today. The first issue is that Web sites are often vulnerable to attack because the Web site software has bugs that result in faulty data flow. A faulty data flow occurs when a programmer has an implicit invariant of how data should flow within the application, but then accidentally uses the data in a way that violates that invariant. For example, a programmer might want to keep a user's password confidential but accidentally send the password to another user, or the application might take untrusted user input and accidentally interpret it as code. RESIN [87] is a programming tool that helps programmers avoid these and other kinds of data flow bugs.

The second issue is that Web sites have begun to run third-party code with access to confidential user data, which can result in data leakage. BFLOW [86] is a system that allows Web sites to run third-party code with confidential data without the risk of leaking that data. This chapter introduces these two systems.

1.1 RESIN

Software developers often have a plan for correct data flow in their applications. For example, in order for a Web site to avoid SQL injection attacks, user input must flow through a sanitization function before the application can use the data in a SQL query. Today, programmers usually implement their data flow plans implicitly in their application code; for example to address SQL injection, programmers often try to call the sanitization function in all the correct places, on all the data flow paths. Unfortunately, there are often many such places, and it is easy to miss some, which results in vulnerabilities.

RESIN is a programming tool that allows programmers to implement an implicit data flow plan explicitly in the form of a high-level *data flow assertion* that applies to the entire application. RESIN verifies that the application abides by the explicit data flow plan throughout the application, even in places where the programmer might have accidentally violated the plan.

The main challenges facing RESIN are knowing when to verify a data flow assertion, providing a convenient way for programmers to express data flow assertions, and

designing mechanisms that make it possible for many different assertions to coexist in the same application without interfering with each other.

Resin addresses these challenges using three ideas: *policy objects*, *data tracking*, and *filter objects*. Programmers explicitly annotate data, such as strings, with policy objects, that help the assertion code understand the data and decide what kind of assertions apply to the data. The Resin runtime then tracks these policy objects as the data propagates through the application. When the data is about to leave the control of Resin, such as being sent over the network, Resin invokes filter objects to check the data flow assertions with assistance from the data's policy objects.

We implemented Resin in two different language runtimes, Python and PHP. We then evaluate Resin by implementing a wide range of data flow assertions in real Web applications. The results show that assertions are short, on the order of tens of lines of code, and require changes in only a few places in the application code. They also show that Resin policies are effective at preventing many vulnerabilities such as SQL injection, cross-site scripting, directory traversal, missing access control checks, and server-side script injection.

1.2 BFlow

In addition to server-side bugs causing vulnerabilities, Web site developers have begun facing vulnerabilities due to third-party scripts running in the browser. In particular, programmers have begun to incorporate JavaScript written by untrusted programmers into their Web sites to expand and improve functionality. However, an increasing number of Web sites manage users' confidential data, and when a Web site combines untrusted JavaScript with confidential user data, the site opens itself to attack. The untrusted JavaScript can leak that confidential data to adversaries by sending the data via other JavaScript running in the browser, or by sending the data in a request to a Web server.

Supporting untrusted JavaScript with confidential data is significantly different from the goals of Resin since a Web site that incorporates untrusted JavaScript will likely run malicious JavaScript code, whereas Resin only helps a programmer secure trusted code.

BFlow is a system that adds information flow control (IFC) [18] to Web browsers and the browser-server interface. BFlow tracks confidential data as it flows from the server to the browser, within the browser, and from the browser back to the server. Since BFlow knows whether untrusted JavaScript may have read confidential data, BFlow can restrict the JavaScript's communication channels so that the JavaScript cannot leak that data to someone who lacks permission to read it. BFlow makes it possible to run untrusted JavaScript in the browser, with access to confidential data, without the risk of leaking that data.

There are two main challenges in applying IFC to Web browser scripts. The first, is fitting IFC into the Web environment; in a Web system, who are the principals, who should configure the security policies, and how does data from different principals interact? Also, at what granularity should the system apply IFC, can the system pre-

serve the special data flow channels that the Web architecture assumes exist between browser scripts, and between a script and a Web server? The second main challenge is to support the large amount of existing software including browser-side scripts and the browsers themselves.

To address the first challenge, BFlow gives each Web site control over its own data. When a user inserts confidential data into a Web site, the Web site is responsible for that data, and site's programmers control who may receive the data according to the Web site's disclosure policy. If a Web site discloses data to another Web site, then the recipient site will have the ability to further disclose the data to any other site or user. To address the second challenge, BFlow overlays its IFC mechanisms onto existing JavaScript abstractions such as browser frames and server origins. For example, BFlow performs IFC at the granularity of *protection zones* which are sets of HTML frames. BFlow then uses the browser's existing isolation mechanisms to implement zone isolation to avoid modifying the browser to add browser support for BFlow.

We implemented a BFlow prototype as a reference monitor running in the browser and a number of minor changes to the browser-server interface. The prototype reference monitor is a browser extension for an off-the-shelf browser. This implementation requires no modifications to the base browser or the JavaScript interpreter. We also implemented three Web applications that demonstrate the range of functionality that untrusted JavaScript can have while running in BFlow. These applications include: a blog that supports existing third-party extensions; a social networking site that implements common application features in untrusted JavaScript; and a multi-application Web platform that permits sharing user data between applications, yet preserves the privacy of user data.

1.3 Contributions

This dissertation makes a number of contributions.

1.3.1 Resin

Resin's contributions are the idea of a data flow assertion, and a technique for implementing data flow assertions using filter objects, policy objects, and data tracking. Experiments with several real applications further show that data flow assertions are concise, effective at preventing many security vulnerabilities, and incrementally deployable in existing applications.

1.3.2 BFlow

BFlow's contributions are a set of information flow control rules that govern the JavaScript communication mechanisms, a mapping from BFlow's IFC rules to the browser's existing JavaScript isolation system, and an abstraction called a protection zone that eases the deployment of existing JavaScript into BFlow. Together, these

techniques allow untrusted JavaScript to read, compute with, and display confidential data without the risk of leaking that data. Experiments with porting existing third-party JavaScript to BFlow, and building new applications in BFlow show that it is possible for existing code to run in the BFlow environment with few changes, and that programmers can build applications in BFlow that might have been too insecure to build with existing techniques.

1.4 Organization

The remainder of this dissertation is organized as follows: Chapter 2 describes the Resin system, and Chapter 3 describes the BFlow system. Chapter 4 provides some thoughts about future research directions. Finally, Chapter 5 concludes.

Chapter 2

Resin

Resin is a new language runtime that helps prevent security vulnerabilities, by allowing programmers to specify data flow assertions. Resin provides *policy objects*, which programmers use to specify assertion code and metadata; *data tracking*, which allows programmers to associate assertions with application data, and to keep track of assertions as the data flow through the application; and *filter objects*, which programmers use to define data flow boundaries at which assertions are checked. Resin's runtime checks data flow assertions by propagating policy objects along with data, as that data moves through the application, and then invoking filter objects when data crosses a data flow boundary, such as when writing data to the network or a file.

Using Resin, Web application programmers can prevent a range of problems, from SQL injection and cross-site scripting, to inadvertent password disclosure and missing access control checks. Adding a Resin assertion to an application requires few changes to the existing application code, and an assertion can reuse existing code and data structures. For instance, 23 lines of code detect and prevent three previously-unknown missing access control vulnerabilities in phpBB, a popular Web forum application. Other assertions comprising tens of lines of code prevent a range of vulnerabilities in Python and PHP applications. A prototype of Resin incurs a 33% CPU overhead running the HotCRP conference management application.

2.1 Introduction

Software developers often have a plan for correct data flow within their applications. For example, a user u's password may flow out of a Web site only via an email to user u's email address. As another example, user inputs must always flow through a sanitizing function before flowing into a SQL query or HTML, to avoid SQL injection or cross-site scripting vulnerabilities. Unfortunately, today these plans are implemented implicitly: programmers try to insert code in *all* the appropriate places to ensure correct flow, but it is easy to miss some, which can lead to exploits. For example, one popular Web application, phpMyAdmin [65], requires sanitizing user input in 1,409 places. Not surprisingly, phpMyAdmin has suffered 60 vulnerabilities because some of these calls were forgotten [71].

This chapter presents Resin, a system that allows programmers to make their plan for correct data flow explicit using data flow assertions. Programmers can write a data flow assertion in one place to capture the application's high-level data flow invariant, and Resin checks the assertion in all relevant places, even places where the programmer might have otherwise forgotten to check.

Resin operates within a language runtime, such as the Python or PHP interpreter. Resin tracks application data as it flows through the application, and checks data flow assertions on every executed path. Resin uses runtime mechanisms because they can capture dynamic properties, like user-defined access control lists, while integration with the language allows programmers to reuse the application's existing code in an assertion. Resin is designed to help programmers gain confidence in the correctness of their application, and is not designed to handle malicious code.

A key challenge facing Resin is knowing when to verify a data flow assertion. Consider the assertion that a user's password can flow only to the user herself. There are many different ways that an adversary might violate this assertion, and extract someone's password from the system. The adversary might trick the application into emailing the password; the adversary might use a SQL injection attack to query the passwords from the database; or the adversary might fetch the password file from the server using a directory traversal attack. Resin needs to cover every one of these paths to prevent password disclosure.

A second challenge is to design a generic mechanism that makes it easy to express data flow assertions, including common assertions like cross-site scripting avoidance, as well as application-specific assertions. For example, HotCRP [44], a conference management application, has its own data flow rules relating to password disclosure and reviewer conflicts of interest, among others. Can a single assertion API allow for succinct assertions for cross-site scripting avoidance as well as HotCRP's unique data flow rules?

The final challenge is to make data flow assertions coexist with each other and with the application code. A single application may have many different data flow assertions, and it must be easy to add an additional assertion if a new data flow rule arises, without having to change existing assertions. Moreover, applications are often written by many different programmers. One programmer may work on one part of the application and lack understanding of the application's overall data flow plan. Resin should be able to enforce data flow assertions without all the programmers being aware of the assertions.

Resin addresses these challenges using three ideas: policy objects, data tracking, and filter objects. Programmers explicitly annotate data, such as strings, with policy objects, which encapsulate the assertion functionality that is specific to that data. Programmers write policy objects in the same language that the rest of the application is written in, and can reuse existing code and data structures, which simplifies writing application-specific assertions. The Resin runtime then tracks these policy objects as the data propagates through the application. When the data is about to leave the control of Resin, such as being sent over the network, Resin invokes filter objects to check the data flow assertions with assistance from the data's policy objects.

Vulnerability	Count	Percentage
SQL injection	1176	20.4%
Cross-site scripting	805	14.0%
Denial of service	661	11.5%
Buffer overflow	550	9.5%
Directory traversal	379	6.6%
Server-side script injection	287	5.0%
Missing access checks	263	4.6%
Other vulnerabilities	1647	28.6%
Total	5768	100%

Table 2.1: Top CVE security vulnerabilities of 2008 [71].

We evaluate Resin in the context of application security by showing how these three mechanisms can prevent a wide range of vulnerabilities in real Web applications, while requiring programmers to write only tens of lines of code. One application, the MoinMoin wiki [56], required only 8 lines of code to catch the same access control bugs that required 2,000 lines in Flume [46], although Flume provides stronger guarantees. HotCRP can use Resin to uphold its data flow rules, by adding data flow assertions that control who may read a paper's reviews, and to whom HotCRP can email a password reminder. Data flow assertions also help prevent a range of other previously-unknown vulnerabilities in Python and PHP Web applications. A prototype Resin runtime for PHP has acceptable performance overhead, amounting to 33% for HotCRP.

The contributions of this work are the idea of a data flow assertion, and a technique for implementing data flow assertions using filter objects, policy objects, and data tracking. Experiments with several real applications further show that data flow assertions are concise, effective at preventing many security vulnerabilities, and incrementally deployable in existing applications.

The rest of the chapter is organized as follows. The next section discusses the specific goals and motivation for Resin. Section 2.3 presents the design of the Resin runtime, and Section 2.4 describes our implementation. Section 2.5 illustrates how Resin prevents a range of security vulnerabilities. Sections 2.6 and 2.7 present our evaluation of Resin's ease of use, effectiveness, and performance. We discuss Resin's limitations in Section 2.9. Section 2.10 covers related work, and Section 2.11 summarizes.

2.2 Goals and Examples

Resin's main goal is to help programmers avoid security vulnerabilities by treating exploits as data flow violations, and then using data flow assertions to detect these violations. This section explains how faulty data flows cause vulnerabilities, and how data flow assertions can prevent those vulnerabilities.

Vulnerability	Vulnerable sites among those surveyed
Cross-site scripting	31.5%
Information leakage	23.3%
Predictable resource location	10.2%
SQL injection	7.9%
Insufficient access control	1.5%
HTTP response splitting	0.8%

Table 2.2: Top Web site vulnerabilities of 2007 [82].

SQL Injection and Cross-Site Scripting

SQL injection and cross-site scripting vulnerabilities are common and can affect almost any Web application. Together, they account for over a third of all reported security vulnerabilities in 2008, as seen in Table 2.1. These vulnerabilities result from user input data flowing into a SQL query string or HTML without first flowing through their respective sanitization functions. To avoid these vulnerabilities today, programmers insert calls to the correct sanitization function on every single path on which user input can flow to SQL or HTML. In practice this is difficult to accomplish because there are many data flow paths to keep track of, and some of them are non-intuitive. For example, in one cross-site scripting vulnerability, phpBB queried a malicious *whois* server, and then incorporated the response into HTML without first sanitizing the response. A survey of Web applications [82] summarized in Table 2.2 illustrates how common these bugs are with cross-site scripting affecting more than 31% of applications, and SQL injection affecting almost 8%.

If there were a tool that could enforce a data flow assertion on an entire application, a programmer could write an assertion to catch these bugs and prevent an adversary from exploiting them. For example, an assertion to prevent SQL injection exploits would verify that:

Data Flow Assertion 1 *Any user input data must flow through a sanitization function before it flows into a SQL query.*

Resin aims to be such a tool.

Directory Traversal

Directory traversal is another common vulnerability that accounts for 6.6% of the vulnerabilities in Table 2.1. In a directory traversal attack, a vulnerable application allows the user to enter a file name, but neglects to limit the directories available to the user. To exploit this vulnerability, an adversary typically inserts the ".." string as part of the file name which allows the adversary to gain unauthorized access to read, or write files in the server's file system. These exploits can be viewed as faulty data flows. If the adversary reads a file without the proper authorization, the file's data is incorrectly flowing to the adversary. If the adversary writes to a file without the proper authorization, the adversary is causing an invalid flow into the file. Data flow

assertions can address directory traversal vulnerabilities by enforcing data flow rules on the use of files. For example, a programmer could encode the following directory traversal assertion to protect against invalid writes:

Data Flow Assertion 2 *No data may flow into directory d unless the authenticated user has write permission for d.*

Server-Side Script Injection

Server-side script injection accounts for 5% of the vulnerabilities reported in Table 2.1. To exploit these vulnerabilities, an adversary uploads code to the server and then fools the application into running that code. For instance, many PHP applications load script code for different visual themes at runtime, by having the user specify the file name for their desired theme. An adversary can exploit this by uploading a file with the desired code onto the server (many applications allow uploading images or attachments), and then supplying the name of that file as the theme to load.

Even if the application is careful to not include user-supplied file names, a more subtle problem can occur. If an adversary uploads a file with a `.php` extension, the Web server may allow the adversary to directly execute that file's contents by simply issuing an HTTP request for that file. Avoiding such problems requires coordination between many parts of the application, and even the Web server, to understand which file extensions are "dangerous". This attack can be viewed as a faulty data flow and could be addressed by the following data flow assertion:

Data Flow Assertion 3 *The interpreter may not interpret any user-supplied code.*

Access Control

Insufficient access control can also be viewed as a data flow violation. These vulnerabilities allow an adversary to read data without proper authorization and make up 4.6% of the vulnerabilities reported in 2008. For example, a missing access control check in MoinMoin wiki allowed a user to read any wiki page, even if the page's access control list (ACL) did not permit the user to read that page [79]. Like the previous vulnerabilities, this data leak can be viewed as a data flow violation; the wiki page is flowing to a user who lacks permission to receive the page. This vulnerability could be addressed with the data flow assertion:

Data Flow Assertion 4 *Wiki page p may flow out of the system only to a user on p's ACL.*

Insufficient access control is particularly challenging to address because access control rules are often unique to the application. For example, MoinMoin's ACL rules differ from HotCRP's access control rules, which ensure that only paper authors and program committee (PC) members may read paper reviews, and that PC members may not view a paper's authors if the author list is anonymous. Ideally, a data flow assertion could take advantage of the code and data structures that an application already uses to implement its access control checks.

Password Disclosure

Another example of a specific access control vulnerability is a password disclosure vulnerability that was discovered in HotCRP; we use this bug as a running example for the rest of this chapter. This bug was a result of two separate features, as follows.

First, a HotCRP user can ask HotCRP to send a password reminder email to the user's email address, in case the user forgets the password. HotCRP makes sure to send the email only to the email address of the account holder as stored in the server. The second feature is an *email preview* mode, in which the site administrator configures HotCRP to display email messages in the browser, rather than send them via email. In this vulnerability, an adversary asks HotCRP to send a password reminder for another HotCRP user (the victim) while HotCRP is in *email preview* mode. HotCRP will display the content of the password reminder email in the adversary's browser, instead of sending the password to that victim's email address, thus revealing the victim's password to the adversary.

A data flow assertion could have prevented this vulnerability because the assertion would have caught the invalid password flow despite the unexpected combination of the password reminder and *email preview* mode. The assertion in this case would have been:

Data Flow Assertion 5 *User u's password may leave the system only via email to u's email address, or to the program chair.*

2.2.1 Threat Model

As we have shown, many vulnerabilities in today's applications can be thought of as programming errors that allow faulty data flows. Adversaries exploit these faulty data flows to bypass the application's security plan. Resin aims to prevent adversaries from exploiting these faulty data flows by allowing programmers to explicitly specify data flow assertions, which are then checked at runtime in all places in the application.

We expect that programmers would specify data flow assertions to prevent well-known vulnerabilities shown in Table 2.1, as well as existing application-specific rules, such as HotCRP's rules for password disclosure or reviewer conflicts of interest. As programmers write new code, they can use data flow assertions to make sure their data is properly handled in code written by other developers, without having to look at the entire code base. Finally, as new problems are discovered, either by attackers or by programmers auditing the code, data flow assertions can be used to fix an entire class of vulnerabilities, rather than just a specific instance of the bug.

Resin treats the entire language runtime, and application code, as part of the trusted computing base. Resin assumes the application code is not malicious, and does not prevent an adversary from compromising the underlying language runtime or the OS. In general, a buffer overflow attack can compromise a language runtime, but buffer overflows are less of an issue for Resin because code written in languages like PHP and Python is not susceptible to buffer overflows.

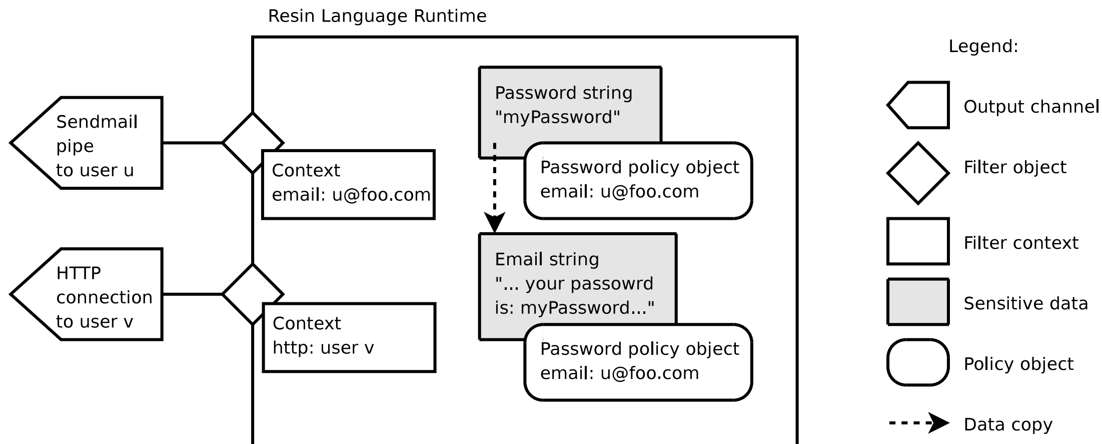

Figure 2-1: Overview of the HotCRP password data flow assertion.

2.3 Design

Many of the vulnerabilities described in Section 2.2 can be addressed with data flow assertions, but the design of such an assertion system requires solutions to a number of challenges. First, the system must enforce assertions on the many communication channels available to the application. Second, the system must provide a convenient API in which programmers can express many different types of data flow assertions. Finally, the system must handle several assertions in a single application gracefully; it should be easy to add new assertions, and doing so should not disrupt existing assertions. This section describes how Resin addresses these design challenges, beginning with an example of how a data flow assertion prevents the HotCRP password disclosure vulnerability described in Section 2.2.

2.3.1 Design Overview

To illustrate the high-level design of Resin and what a programmer must do to implement a data flow assertion, this section describes how a programmer would implement Data Flow Assertion 5, the HotCRP password assertion, using Resin. This example does not use all of Resin's features, but it does show Resin's main concepts.

Conceptually, the programmer needs to restrict the flow of passwords. However, passwords are handled by a number of modules in HotCRP, including the authentication code and code that formats and sends email messages. Thus, the programmer must confine passwords by defining a data flow boundary that surrounds the entire application. Then the programmer allows a password to exit that boundary only if that password is flowing to the owner via email, or to the program chair. Finally, the programmer marks the passwords as sensitive so that the boundary can identify which data contains password information, and writes a small amount of assertion checking code.

```
class PasswordPolicy extends Policy {
  private $email;
  function __construct($email) {
    $this->email = $email;
  }
  function export_check($context) {
    if ($context['type'] == 'email' &&
        $context['email'] == $this->email) return;
    global $Me;
    if ($context['type'] == 'http' &&
        $Me->privChair) return;
    throw new Exception('unauthorized disclosure');
  }
}

policy_add($password, new PasswordPolicy('u@foo.com'));
```

Figure 2-2: Simplified PHP code for defining the HotCRP password policy class and annotating the password data. This policy only allows a password to be disclosed to the user's own email address or to the program chair.

Resin provides three mechanisms that help the programmer implement such an assertion (see Figure 2-1):

- Programmers use *filter objects* to define data flow boundaries. A filter object interposes on an input/output channel or a function call interface.
- Programmers explicitly annotate sensitive data with *policy objects*. A policy object can contain code and metadata for checking assertions.
- Programmers rely on Resin's runtime to perform *data tracking* to propagate policy objects along with sensitive data when the application copies that data within the system.

Resin by default defines a data flow boundary around the language runtime using filter objects that cover all I/O channels, including pipes and sockets. By default, Resin also annotates some of these default filter objects with *context* metadata that describes the specific filter object. For example, Resin annotates each filter object connected to an outgoing email channel with the email's recipient address. The default set of filters and contexts defining the boundary are appropriate for the HotCRP password assertion, so the programmer need not define them manually.

In order for Resin to track the passwords, the programmer must annotate each password with a policy object, which is a language-level object that contains fields and methods. In this assertion, a user's password will have a policy object that contains a copy of the user's email address so that the assertion can determine which email address may receive the password data. When the user first sets their password, the programmer copies the user's email address from the current session information into the password's policy object.

The programmer also writes the code that checks the assertion, in a method called *export_check* within the password policy object's class definition. Figure 2-2 shows the code the programmer must write to implement this data flow assertion, including the policy object's class definition and the code that annotates a password with a

policy object. The policy object also shows how an assertion can benefit from the application's data structures; this assertion uses an existing flag, *$Me->privChair*, to determine whether the current user is the program chair.

Once a password has the appropriate policy object, RESIN's data tracking propagates that policy object along with the password data; when the application copies or moves the data within the system, the policy goes along with the password data. For example, after HotCRP composes the email content using the password data, the email content will also have the password policy annotation (as shown in Figure 2-1).

RESIN enforces the assertion by making each filter object call *export_check* on the policy object of any data that flows through the filter. The filter object passes its context as an argument to *export_check* to provide details about the specific I/O channel (e.g., the email's recipient).

This assertion catches HotCRP's faulty data flow before it can leak a password. When HotCRP tries to send the password data over an HTTP connection, the connection's filter object invokes the *export_check* method on the password's policy object. The *export_check* code observes that HotCRP is incorrectly trying to send the password over an HTTP connection, and throws an exception which prevents HotCRP from sending the password to the adversary. This solution works for all disclosure paths through the code because RESIN's default boundary controls all output channels; HotCRP cannot reveal the password without traversing a filter object.

This example is just one way to implement the password data flow assertion, and there may be other ways. For example, the programmer could implement the assertion checking code in the filter objects rather than the password's policy object. However, modifying filter objects is less attractive because the programmer would need to modify every filter object that a password can traverse. Putting the assertion code in the policy object allows the programmer to write the assertion code in one place.

2.3.2 Filter Objects

A filter object, represented by a diamond in Figure 2-1, is a generic interposition mechanism that application programmers use to create data flow boundaries around their applications. An application can associate a filter object with a function call interface, or an I/O channel such as a file handle, socket, or pipe.

RESIN aims to support data flow assertions that are specific to an application, so RESIN allows a programmer to implement a filter object as a language-level object in the same language as the rest of the application. This allows the programmer to reuse the application's code and data structures, and allows for better integration with applications.

When an application sends data across a channel guarded by a filter object, RESIN invokes a method in that filter object with the data as an argument. If the interposition point is an I/O channel, RESIN will invoke either *filter_read* or *filter_write*; for function calls, RESIN will invoke *filter_func* (see Table 2.3). *Filter_read* and *filter_write* can check or alter the in-transit data. *Filter_func* can check or alter the function's arguments and return value.

Function	Caller	Semantics
filter::filter_read(*data*, *offset*)	Runtime	Invoked when data comes in through a data flow boundary, and can assign initial policies for *data*; e.g., by de-serializing from persistent storage.
filter::filter_write(*data*, *offset*)	Runtime	Invoked when data is exported through a data flow boundary; typically invokes assertion checks or serializes policy objects to persistent storage.
filter::filter_func(*args*)	Runtime	Checks and/or proxies a function call.
policy::export_check(*context*)	Filter object	Checks if data flow assertion allows exporting data, and throws exception if not; *context* provides information about the data flow boundary.
policy::merge(*policy_object_set*)	Runtime	Returns set of policies (typically zero or one) that should apply to merging of data tagged with this policy and data tagged with *policy_object_set*.
policy_add(*data*, *policy*)	Programmer	Adds *policy* to *data*'s policy set.
policy_remove(*data*, *policy*)	Programmer	Removes *policy* from *data*'s policy set.
policy_get(*data*)	Programmer	Returns set of policies associated with *data*.

Table 2.3: The Resin API. A::B(args) denotes method B of an object of type A. Not shown is the API used by the programmer to specify and access filter objects for different data flow boundaries.

```
class DefaultFilter(Filter):
  def __init__(self): self.context = {}
  def filter_write(self, buf):
    for p in policy_get(buf):
      if hasattr(p, 'export_check'):
        p.export_check(self.context)
    return buf
```

Figure 2-3: Python code for the default filter for sockets.

For example, in an HTTP splitting attack, the adversary inserts an extra CR-LF-CR-LF delimiter into the HTTP output to confuse browsers into thinking there are two HTTP responses. To thwart this type of attack, the application programmer could write a filter object that scans for unexpected CR-LF-CR-LF character sequences, and then attach this filter to the HTTP output channel. As a second example, a function that encrypts data is a natural data flow boundary. A programmer may choose to attach a filter object to the encryption function that removes policy objects for confidentiality assertions such as the *PasswordPolicy* from Section 2.3.1.

Default Filter Objects

RESIN pre-defines default filter objects on all I/O channels into and out of the runtime, including sockets, pipes, files, HTTP output, email, SQL, and code import. Since these default filter objects are at the edge of the runtime, data can flow freely within the application and the default filters will only check assertions before making program output visible to the outside world. This boundary should be suitable for many assertions because it surrounds the entire application. The default boundary also helps programmers avoid accidentally overlooking an I/O channel, which would result in an incomplete boundary that would not cover all possible flows.

The default filter objects check the in-transit data for policies, as shown in Figure 2-3. If a filter finds a policy that has an *export_check* method, the filter invokes the policy's *export_check* method. As described in Section 2.3.1, *export_check* typically checks the assertion and throws an exception if the flow would violate the assertion.

Since the policy's *export_check* method may need additional information about the filter's specific I/O channel or function call to check the assertion, RESIN attaches context information, in the form of a hash table, to some of the default filters as described in Section 2.3.1. RESIN also allows the application to add its own key-value pairs to the context hash table of default filter objects.

The context key-value pairs are likely specific to the I/O channel or function call that the filter guards, and the default filter passes the context hash table as an argument to *export_check*. In the HotCRP example, the context for a sendmail pipe contains the recipient of the email (as shown in Figure 2-1).

Importing Code

RESIN treats the interpreter's execution of script code as another data flow channel, with its own filter object. This allows programmers to interpose on all code flow-

ing into the interpreter, and ensure that such code came from an approved source. This can prevent server-side script injection attacks, where an adversary tricks the interpreter into executing adversary-provided script code.

Write Access Control

In addition to runtime boundaries, RESIN also permits an application to place filter objects on persistent files to control write access, because data tracking alone cannot prevent modifications. In particular, RESIN allows programmers to specify access control checks for files and directories in a persistent filter object that's stored in the extended attributes of a specific file or directory. The runtime automatically invokes this persistent filter object when data flows into or out of that file, or modifies that directory (such as creating, deleting, or renaming files). This programmer-specified filter object can check whether the current user is authorized to access that file or directory. These persistent filter objects associated with a specific file or directory are separate from the filter objects associated by default with every file's I/O channel.

2.3.3 Policy Objects

Like a filter object, a policy object is a language-level object, and can reuse the application's existing code and data structures. A policy object can contain fields and methods that work in concert with filter objects; policy objects are represented by the rounded rectangles in Figure 2-1.

To work with default filter objects, a policy object should have an *export_check* method as shown in Table 2.3. As mentioned earlier, default filter objects invoke *export_check* when data with a policy passes through a filter, so *export_check* is where programmers implement an assertion check for use with default filters. If the assertion fails, *export_check* should throw an exception so that RESIN can prevent the faulty data flow.

The main distinction between policy objects and filter objects is that a policy object is specific to data, and a filter object is specific to a channel. A policy object would contain data specific metadata and code; for example, the HotCRP password policy contains the email address of the password's account holder. A filter object would contain channel specific metadata; for example, the email filter object contains the recipient's email address.

Even though RESIN allows programmers to write many different filter and policy objects, the interface between all filters and policies remains largely the same, if the application uses *export_check*. This limits the complexity of adding or changing filters and policies, because each policy object need not know about all possible filter objects, and each filter object need not know about all possible policy objects (although this does not preclude the programmer from implementing special cases for certain policy-filter pairs).

2.3.4 Data Tracking

Resin keeps track of policy objects associated with data. The programmer attaches a policy object to a datum—a primitive data element such as an integer or a character in a string (although it is common to assign the same policy to all characters in a string). The Resin runtime then propagates that policy object along with the data, as the application code moves or copies the data.

To attach a policy object to data, the programmer uses the *policy_add* function listed in Table 2.3. Since an application may have multiple data flow assertions, a single datum may have multiple policy objects, all contained in the datum's *policy set*.

Resin propagates policies in a fine grained manner. For example, if an application concatenates the string "foo" (with policy p_1), and "bar" (with policy p_2), then in the resulting string "foobar", the first three characters will have only policy p_1 and the last three characters will have only p_2. If the programmer then takes the first three characters of the combined string, the resulting substring "foo" will only have policy p_1. Tracking data at the character level minimizes interference between different data flow assertions, whose data may be combined in the same string, and minimizes unintended policy propagation, when marshaling and un-marshaling data structures. For example, in the HotCRP password reminder email message, only the characters comprising the user's password have the password policy object. The rest of the string is not associated with the password policy object, and can potentially be manipulated and sent over the network without worrying about the password policy (assuming there are no other policies).

Resin tracks explicit data flows such as variable assignment; most of the bugs we encountered, including all the bugs described in Sections 2.2 and 2.6, use explicit data flows. However, some data flows are implicit. One example is a control flow channel, such as when a value that has a policy object influences a conditional branch, which then changes the program's output. Another example of an implicit flow is through data structure layout; an application can store data in an array using a particular order. Resin does not track this order information, and a programmer cannot attach a policy object to the array's order. These implicit data flows are sometimes surprising and difficult to understand, and Resin does not track them. If the programmer wants to specify data flow assertions about such data, the programmer must first make this data explicit, and only then attach a policy to it.

Persistent Policies

Resin only tracks data flow inside the language runtime, and checks assertions at the runtime boundary, because it cannot control what happens to the data after it leaves the runtime. However, many applications store data persistently in file systems and databases. For example, HotCRP stores user passwords in a SQL database. It can be inconvenient and error-prone for the programmer to manually save metadata about the password's policy object when saving it to the database, and then reconstruct the policy object when reading the password later.

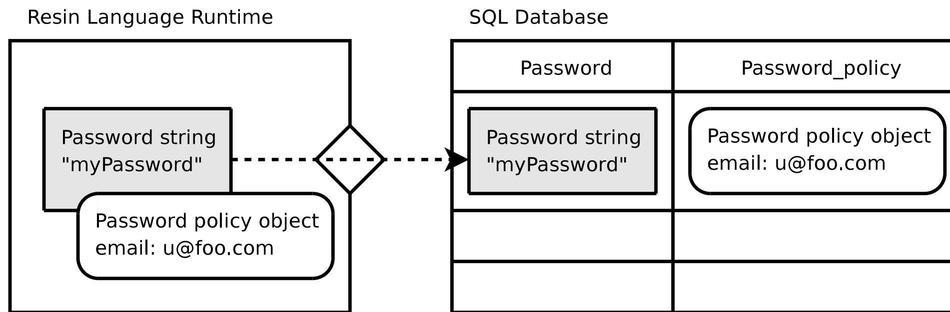

Figure 2-4: Saving persistent policies to a SQL database for HotCRP passwords. Uses symbols from Figure 2-1.

To help programmers of such applications, Resin transparently tracks data flows to and from persistent storage. Resin's default filter objects serialize policy objects to persistent files and database storage when data is written out, and de-serializes the policy objects when data is read back into the runtime.

For data going to a file, the file's default filter object serializes the data's policy objects into the file's extended attributes. Whenever the application reads data from the file, the filter reads the serialized policy from the file's extended attributes, and associates it with the newly-read data. Resin tracks policies for file data at byte-level granularity, as it does for strings.

Resin also serializes policies for data stored in a SQL database, as shown in Figure 2-4. Resin accomplishes this by attaching a default filter object to the function used to issue SQL queries, and using that filter to rewrite queries and results. For a *CREATE TABLE* query, the filter adds an additional policy column to store the serialized policy for each data column. For a query that writes to the database, the filter augments the query to store the serialized policy of each cell's value into the corresponding policy column. Last, for a query that fetches data, the filter augments the query to fetch the corresponding policy column, and associates each de-serialized policy object with the corresponding data cell in the resulting set of rows.

Storing policies persistently also helps other programs, such as the Web server, to check invariants on file data. For example, if an application accidentally stores passwords in a world-readable file, and an adversary tries to fetch that file via HTTP, a Resin-aware Web server will invoke the file's policy objects before transmitting the file, fail the *export_check*, and prevent password disclosure.

Resin only serializes the class name and data fields of a policy object. This allows programmers to change the code for a policy class to evolve its behavior over time. For example, a programmer could change the *export_check* method of HotCRP's password policy object to disallow disclosure to the program chair without changing the existing persistent policy objects. However, if the application needs to change the fields of a persistent policy, the programmer will need to update the persistent policies, much like database schema migration.

Merging Policies

Resin uses character-level tracking to avoid having to merge policies when individual data elements are propagated verbatim, such as through concatenation or taking a substring. Unfortunately, merging is inevitable in some cases, such as when string characters with different policies are converted to integer values and added up to compute a checksum. In many situations, such low-level data transformation corresponds to a boundary, such as encryption or hashing, and would be a good fit for an application-specific filter object. However, relying on the programmer to insert filter objects in all such places would be error-prone, and Resin provides a safety net by merging policy objects in the absence of any explicit actions by the programmer.

By default, Resin takes the union of policy objects of source operands, and attaches them to the resulting datum. The *union* strategy is suitable for some data flow assertions. For example, an assertion that tracks user-supplied inputs by marking them with a *UserData* policy would like to label the result as *UserData* if any source operand was labeled as such. In contrast, the *intersection* strategy may be applicable to other policies. An assertion that tracks data authenticity by marking data with an *AuthenticData* policy would like to label the result as *AuthenticData* only if all source operands were labeled that way.

Because different policies may have different notions of a safe merge strategy, Resin allows a policy object to override the *merge* method shown in Table 2.3. When application code merges two data elements, Resin invokes the *merge* method on each policy of each source operand, passing in the entire policy set of the other operand as the argument. The *merge* method returns a set of policy objects that it wants associated with the new datum, or throws an exception if this policy should not be merged. The *merge* method can consult the set of policies associated with the other operand to implement either the *union* or *intersection* strategies. The Resin runtime then labels the resulting datum with the union of all policies returned by all *merge* methods.

2.4 Implementation

We have implemented two Resin prototypes, one in the PHP runtime, and the other in Python. At a high-level, Resin requires the addition of a pointer, that points to a set of policy objects, to the runtime's internal representation of a datum. For example, in PHP, the additional pointer resides in the *zval* object for strings and numbers. For strings, each policy object contains a character range for which the policy applies. When copying or moving data from one primitive object to another, the language runtime copies the policy set from the source to the destination, and modifies the character ranges if necessary. When merging individual data elements, the runtime invokes the policies' merge functions.

The PHP prototype involved 5,944 lines of code. The largest module is the SQL parsing and translation mechanism at about 2,600 lines. The core data structures and related functions make up about 1,100 lines. Most of the remaining 2,200 lines are

spent propagating and merging policy objects. Adding propagation to the core PHP language required changes to its virtual machine opcode handlers, such as variable assignment, addition, and string concatenation. In addition, PHP implements many of its library functions, such as *substr* and *printf*, in C, which are outside of PHP's virtual machine and require additional propagation code.

To allow the Web server to check persistent policies for file data, as described in Section 2.3.4, we modified the *mod_php* Apache module to de-serialize and invoke policy objects for all static files it serves. Doing so required modifying 49 lines of code in *mod_php*.

The Python prototype only involved 681 lines of code; this is fewer than the PHP prototype for two reasons. First, our Python prototype does not implement all the Resin features; it lacks character-level data tracking, persistent policy storage in SQL databases, and Apache static file support. Second, Python uses fewer C libraries, so it required little propagation code beyond the opcode handlers.

2.5 Applying Resin

Resin's main goal is to allow programmers to avoid security vulnerabilities by specifying data flow assertions. Section 2.3.1 already showed how a programmer can implement a data flow assertion that prevents password disclosure in HotCRP. This section shows how a programmer would implement data flow assertions for a number of other vulnerabilities and applications using Resin.

The following examples use the syntax described in Table 2.3. Additionally, these examples use `sock.__filter` to access a socket's filter object, and in the Python code, *policy_add* and *policy_remove* return a new string with the same contents but a different policy set, because Python strings are immutable.

2.5.1 Access Control Checks

As mentioned in Section 2.2, Resin aims to address missing access control checks. To illustrate how a programmer would use Resin to verify access control checks, this section provides an example implementation of Data Flow Assertion 4, the assertion that verifies MoinMoin wiki's read ACL scheme (see Section 2.2).

The MoinMoin ACL assertion prevents a wiki page from flowing to a user that's not on the page's ACL. One way for a programmer to implement this assertion in Resin is to:

1. annotate HTTP output channels with context that identifies the user on the other end of the channel;
2. define a *PagePolicy* object that contains an ACL;
3. implement an *export_check* method in *PagePolicy* that matches the output channel against the *PagePolicy*'s ACL;
4. attach a *PagePolicy* to the data in each wiki page.

```
def process_client(client_sock):
  req = parse_request(client_sock)
  client_sock.__filter.context['user'] = req.user
  ... process req ...

class PagePolicy(Policy):
  def __init__(self, acl): self.acl = acl
  def export_check(self, context):
    if not self.acl.may(context['user'], 'read'):
      raise Exception("insufficient access")

class Page:
  def update_body(self, text):
    text = policy_add(text, PagePolicy(self.getACL()))
    ... write text to page's file ...
```

Figure 2-5: Python code for a data flow assertion that checks read access control in MoinMoin. The *process_client* and *update_body* functions are simplified versions of MoinMoin equivalents.

Figure 2-5 shows all the code necessary for this implementation. The *process_client* function annotates each HTTP connection's context with the current user, after parsing the user's request and credentials. *PagePolicy* contains a copy of the ACL, and implements *export_check*. The *update_body* method creates a *PagePolicy* object and attaches it to the page's data before saving the page to the file system. One reason why the *PagePolicy* is short is that it reuses existing application code to perform the access control check.

This example assertion illustrates the use of persistent policies. The *update_body* function associates a *PagePolicy* with the contents of a page immediately before writing the page to a file. As the page data flows to the file, the default filter object serializes the *PagePolicy* object, including the access control list, to the file system. When MoinMoin reads this file later, the default filter will de-serialize the *PagePolicy* and attach it to the page data in the runtime, so that RESIN will automatically enforce the same access control policy.

In this implementation, the *update_body* function provides a single place where MoinMoin saves the page to the file system, and a thus a single place to attach the *PagePolicy*. If, however, MoinMoin had multiple code paths that stored pages in the file system, the programmer could assign the policy to the page contents earlier, perhaps directly to the CGI input variables.

In addition to read access checks, the programmer can also define a data flow assertion that verifies write access checks. MoinMoin's write ACLs imply the assertion: *data may flow into wiki page p only if the user is on p's write ACL*. MoinMoin stores a wiki page as a directory that contains each version of the page as a separate file. The programmer can implement this assertion by creating a filter class that verifies the write ACL against the current user, and then attaching filter instances to the files and directory that represent a wiki page. The filters restrict the modification of existing versions, and also the creation of new versions based on the page's ACL.

```
class CodeApproval extends Policy {
  function export_check($context) {}
}

function make_file_executable($f) {
  $code = file_get_contents($f);
  policy_add($code, new CodeApproval());
  file_put_contents($f, $code);
}

class InterpreterFilter extends Filter {
  function filter_read($buf) {
    foreach (policy_get($buf) as $p)
      if ($p instanceof CodeApproval)
        return $buf;
    throw new Exception('not executable');
  }
}
```

Figure 2-6: Simplified PHP code for a data flow assertion that catches server-side script injection. In the actual implementation, *filter_read* verifies that each character in *$buf* has the *CodeApproval* policy.

2.5.2 Server-Side Script Injection

Another class of vulnerabilities that Resin aims to address is server-side script injection, as described in Section 2.2, which can be addressed with Data Flow Assertion 3. One way for the programmer to implement this assertion is to:

1. define an empty *CodeApproval* policy object;[1]
2. annotate application code and libraries with *CodeApproval* policy objects;
3. change the interpreter's default input filter (see Section 2.3.2) to require a *CodeApproval* policy on all imported code.

This data flow assertion instructs Resin to limit what code the interpreter may use. Figure 2-6 lists the code for implementing this assertion. When installing an application, the developer tags the application code and system libraries with a persistent *CodeApproval* policy object using *make_file_executable*. The *filter_read* method only allows code with a *CodeApproval* policy object to pass, ensuring that code from an adversary which would lack the *CodeApproval* policy, will not be executed, whether through *include* statements, *eval*, or direct HTTP requests.

The programmer must override the interpreter's filter in a global configuration file, to ensure the filter is set before any other code executes; PHP's *auto_prepend_file* option is one way to do this. If, instead, the application set the filter at the beginning of the application's own code, adversaries could bypass the check if they are able to upload and run their own `.php` files.

This example illustrates the need for programmer-specified filter objects in addition to programmer-specified context for default filters. The default filter calls

[1]The *CodeApproval* policy does not need to take the intersection of policies during merge because Resin's character-level data tracking avoids having to merge file data.

export_check on all the policies that pass through, but the default filter always permits data that has no policy. The filter in this script injection assertion requires that data have a *CodeApproval* policy, and reject data that does not.

2.5.3 SQL Injection and Cross-Site Scripting

As mentioned in Section 2.2, the two most popular attack vectors in Web applications today are SQL injection and cross-site scripting. This section presents two different strategies for using RESIN to address these vulnerabilities.

To implement the first strategy, the programmer:

1. defines two policy object classes: *UntrustedData* and *SQLSanitized*;
2. annotates untrusted input data with an *UntrustedData* policy;
3. changes the existing SQL sanitization function to attach a *SQLSanitized* object to the freshly sanitized data;
4. changes the SQL filter object to check the policy objects on each SQL query. If the query contains any characters that have the *UntrustedData* policy, but not the *SQLSanitized* policy, the filter will throw an exception and refuse to forward the query to the database.

Addressing cross-site scripting is similar, except that it uses *HTMLSanitized* rather than *SQLSanitized*. This strategy catches unsanitized data because the data will lack the correct *SQLSanitized* or *HTMLSanitized* policy object. The reason for appending *SQLSanitized* and *HTMLSanitized* instead of removing *UntrustedData* is to allow the assertion to distinguish between data that may be incorporated into SQL versus HTML since they use different sanitization functions. This strategy ensures that the programmer uses the correct sanitizer (e.g., the programmer did not accidentally use SQL quoting for a string used as part of an HTML document).

The second strategy for preventing SQL injection and cross-site scripting vulnerabilities is to use the same *UntrustedData* policy from the previous strategy, but rather than appending a policy like *SQLSanitized*, the SQL filter inspects the final query and throws an exception if any characters in the query's structure (keywords, white space, and identifiers) have the *UntrustedData* policy. The HTML filter performs a similar check for *UntrustedData* on JavaScript portions of the HTML to catch cross-site scripting errors, similar to a technique used in prior work [60].

A variation on the second strategy is to change the SQL filter's tokenizer to keep contiguous bytes with the *UntrustedData* policy in the same token, and to automatically sanitize the untrusted data in transit to the SQL database. This will prevent untrusted data from affecting the command structure of the query, and likewise for the HTML tokenizer. These two variations require the addition of either tokenizing or parsing to the filter objects, but they avoid relying on trusted quoting functions.

We have experimented with both of these strategies, and find that while the second approach requires more code for the parsers, many applications can reuse the same parsing code.

A SQL injection assertion is complementary to the other assertions we describe in this section. For instance, even if an application has a SQL injection vulnerability, and an adversary manages to execute the query `SELECT user, password FROM userdb`, the policy object for each password will still be de-serialized from the database, and will prevent password disclosure.

2.5.4 Other Attack Vectors

Finally, there are a number of other attack vectors that Resin can help defend against. For instance, to address the HTTP response splitting attack described in Section 2.3.2, a developer can use a filter to reject any CR-LF-CR-LF sequences in the HTTP header that came from user input.

As Web applications use more client-side code, they also use more JSON to transport data from the server to the client. Here, much like in SQL injection, an adversary may be able to craft an input string that changes the structure of the JSON's JavaScript data structure, or worse yet, include client-side code as part of the data structure. Web applications can use Resin's data tracking mechanisms to avoid these pitfalls as they would for SQL injection.

2.5.5 Application Integration

One potential concern when using Resin is that a data flow assertion can duplicate data flow checks and security checks that already exist in an application. As a concrete example, consider HotCRP, which maintains a list of authors for each paper. If a paper submission is anonymous, HotCRP must not reveal the submission's list of authors to the PC members. HotCRP already performs this check before adding the author list to the HTML output. Adding a Resin data flow assertion to verify read access to the author list will make HotCRP perform the access check a second time within the data flow assertion, duplicating the check that already exists.

If a programmer implements an application with Resin in mind, the programmer can use an exception to indicate that the user may not read certain data, thereby avoiding duplicate access checks. For example, we modified the HotCRP code that displays a paper submission to always try to display the submission's author list. If the submission is anonymous, the data flow assertion raises an exception; the display code catches that exception, and then displays the string "Anonymous" instead of the author list. This avoids duplicate checks because the page generation code does not explicitly perform the access control check. However, if the application sends HTML output to the browser during a *try* block and then encounters an exception later in the *try* block, the previously released HTML might be invalid because the *try* block did not run to completion.

Resin provides an *output buffering* mechanism to assist with this style of code. To use output buffering, the application starts a new *try* block before running HTML generation code that might throw an exception. At the start of the *try* block, the application notifies the outgoing HTML filter object to start buffering output. If the *try* block throws an exception, the corresponding *catch* block notifies the HTML filter

to discard the output buffer, and potentially send alternate output in its place (such as "Anonymous" in the example). However, if the *try* block runs to completion, the *try* block notifies the HTML filter to release the data in the output buffer.

Using exceptions, instead of explicit access checks, frees the programmer from needing to know exactly which checks to invoke in every single case, because RESIN invokes the checks. Instead, programmers need to only wrap code that might fail a check with an appropriate exception handler, and specify how to present an exception to the user.

2.6 Security Evaluation

The main criteria for evaluating RESIN is whether it is effective at helping a programmer prevent data flow vulnerabilities. To provide a quantitative measure of RESIN's effectiveness, we focus on three areas. First, we determine how much work a programmer must do to implement an existing implicit data flow plan as an explicit data flow assertion using RESIN. We then evaluate whether each data flow assertion actually prevents typical data flow bugs, both previously-known and previously-unknown bugs. Finally, we evaluate whether a single high-level assertion can be general enough to cover both common and uncommon data flows that might violate the assertion, by testing assertions against bugs that use surprising data paths.

2.6.1 Programmer Effort

To determine the level of effort required for a programmer to use RESIN, we took a number of existing, off-the-shelf applications and examined some of their implicit security-related data flow plans. We then implemented a RESIN data flow assertion for each of those implicit plans. Table 2.4 summarizes the results, showing the applications, the number of lines of code in the application, and the number of lines of code in each data flow assertion.

The results in Table 2.4 show that each data flow assertion requires a small amount of code, on the order of tens of lines of code. The assertion that checks read access to author lists in HotCRP requires the most changes, 32 lines. This is more code than other assertions because our implementation issues database queries and interprets the results to perform the access check, requiring extra code. However, many of the other assertions in Table 2.4 reuse existing code from the application's existing security plan, and are shorter.

Table 2.4 also shows that the effort required to implement a data flow assertion does not grow with the size of the application. This is because implementing an assertion only requires changes where sensitive data first enters the application, and/or where data exits the system, not on every path data takes through the application; RESIN's data tracking handles those data paths. For example, the cross-site scripting assertion for phpBB is only 22 lines of code even though phpBB is 172,000 lines of code.

Application	Lang.	App. LOC	Assertion LOC	Known vuln.	Discovered vuln.	Prevented vuln.	Vulnerability type
MIT EECS grad admissions	Python	18,500	9	0	3	3	SQL injection
MoinMoin	Python	89,600	8	2	0	2	Missing read access control checks
			15	0	0	0	Missing write access control checks
File Thingie file manager	PHP	3,200	19	0	1	1	Directory traversal, file access control
HotCRP	PHP	29,000	23	1	0	1	Password disclosure
			30	0	0	0	Missing access checks for papers
			32	0	0	0	Missing access checks for author list
myPHPscripts login library	PHP	425	6	1	0	1	Password disclosure
PHP Navigator	PHP	4,100	17	0	1	1	Directory traversal, file access control
phpBB	PHP	172,000	23	1	3	4	Missing access control checks
			22	4	0	4	Cross-site scripting
many [23, 40, 16, 61, 4]	PHP	–	12	5	0	5	Server-side script injection

Table 2.4: Results from using RESIN assertions to prevent previously-known and newly discovered vulnerabilities in several Web applications.

As a point of comparison for programmer effort, consider the MoinMoin access control scheme that appeared in the Flume evaluation [46]. MoinMoin uses ACLs to limit who can read and write a wiki page. To implement this scheme under Flume, the programmer partitions MoinMoin into a number of components, each with different privileges, and then sets up the OS to enforce the access control system using information flow control. Adapting MoinMoin to use Flume requires modifying or writing about 2,000 lines of application code. In contrast, RESIN can check the same MoinMoin access control scheme using two assertions, an eight line assertion for reading, and a 15 line assertion for writing, as shown in Table 2.4. Most importantly, adding these checks with RESIN requires no structural or design changes to the application.

Although Flume provides assurance against malicious server code and RESIN does not, the RESIN assertions catch the same two vulnerabilities (see Section 2.6.2) that Flume catches, because they do not involve binary code injection. By focusing on a weaker threat model, RESIN's lightweight and easy-to-use mechanisms provide a compelling choice for programmers that want additional security assurance without much extra effort.

2.6.2 Preventing Vulnerabilities

To evaluate whether RESIN's data flow assertions are capable of preventing vulnerabilities, we checked some of the assertions in Table 2.4 against known vulnerabilities that the assertion should be able to prevent. The results are shown in Table 2.4, where the number of previously-known vulnerabilities is greater than zero.

The results in Table 2.4 show that each RESIN assertion does prevent the vulnerabilities it aims to prevent. For example, the phpBB access control assertion prevents a known missing access control check listed in the CVE [71], and the HotCRP password protection assertion shown in Section 2.3.1 prevents the password disclosure vulnerability described in Section 2.2. The assertion to prevent server-side script injection described in Section 2.5.2 prevents such vulnerabilities in five different applications [23, 40, 16, 61, 4].

Since we implemented these assertions with knowledge of the previously-known vulnerabilities, it is possible that the assertions are biased to thwart only those vulnerabilities. To address this bias, we tried to find new bugs, as an adversary would, that violate the assertions in Table 2.4. These results are shown in Table 2.4 where the number of newly discovered vulnerabilities is greater than zero.

These results show that RESIN assertions can prevent vulnerabilities, even if the programmer has no knowledge of the specific vulnerabilities when writing the assertion. For example, we implemented a generic data flow assertion to address SQL injection vulnerabilities in MIT's EECS graduate admissions system. Although the original programmers were careful to avoid most SQL injection vulnerabilities, the assertion revealed three previously-unknown SQL injection vulnerabilities in the admission committee's internal user interface.

As a second example, File Thingie and PHP Navigator are Web based file managers, and both support a feature that limits a user's write access to a particular home directory. We implemented this behavior as a write access filter as described in

Section 2.3.2. Again, both applications have code in place to check directory accesses, but after a careful examination, we discovered a directory traversal vulnerability that violates the write access scheme in each application. The data flow assertions catch both of these vulnerabilities.

As a final example, phpBB implements read access controls so that only certain users can read certain forum messages. We implemented an assertion to verify this access control scheme. In addition to preventing a previously-known access control vulnerability, the assertion also prevents three previously-unknown read access violations that we discovered. These results confirm that Resin's data flow assertions can thwart vulnerabilities, even if the programmer does not know they exist. Furthermore, these assertions likely eliminate even more vulnerabilities that we are not aware of.

The three vulnerabilities in phpBB are not in the core phpBB package, but in plugins written by third-party programmers. Large-scale projects like phpBB are a good example of the benefit of explicitly specifying data flow assertions with Resin. Consider a situation where a new programmer starts working on an existing application like HotCRP or phpBB. There are many implicit rules that programmers must follow in hundreds of places, such as who is responsible for sanitizing what data to prevent SQL injection and cross-site scripting, and who is supposed to call the access control function. If a programmer starts writing code before understanding all of these rules, the programmer can easily introduce vulnerabilities, and this turned out to be the case in the phpBB plugins we examined. Using Resin, one programmer can make a data flow rule explicit as an assertion and then Resin will check that assertion for all the other programmers.

These results also provide examples of a single data flow assertion thwarting more than one instance of an entire class of vulnerabilities. For example, the single read access assertion in phpBB thwarts four specific instances of read access vulnerabilities (see Table 2.4). As another example, a single server-side script injection assertion that works in all PHP applications catches five different previously-known vulnerabilities in the PHP applications we tested (see Table 2.4). This suggests that when a programmer inevitably finds a security vulnerability and writes a Resin assertion that addresses it, the assertion will prevent the broad class of problems that allow the vulnerability to occur in the first place, rather than only fixing the one specific instance of the problem.

2.6.3 Generality

To evaluate whether Resin data flow assertions are general enough to cover the many data flow paths available to an adversary, we checked whether the assertions we wrote detect a number of data flow bugs that use surprising data flow channels.

The results indicate that a high-level Resin assertion can detect and prevent vulnerabilities even if the vulnerability takes advantage of an unanticipated data flow path. For example, a common way for an adversary to exploit a cross-site scripting vulnerability is to enter malicious input through HTML form inputs. However, there was a cross-site scripting vulnerability in phpBB due to a more unusual data path.

In this vulnerability, phpBB requests data from a *whois* server and then uses the response without sanitizing it first; an adversary exploits this vulnerability by inserting malicious JavaScript code into a *whois* record and then requesting the *whois* record via phpBB. The RESIN assertion that protects against cross-site scripting in phpBB, listed in Table 2.4, prevents vulnerabilities at a high-level; the assertion treats *all* external input as untrusted and makes sure that the external input data flows through a sanitizer before phpBB may use the data in HTML. This assertion is able to prevent both the more common HTML form attack as well as the less common *whois* style attack because the assertion is general enough to cover many possible data flow paths.

A second example is in the read access controls for phpBB's forum messages. The common place to check for read access is before displaying the message to a user, but one of the read access vulnerabilities, listed in Table 2.4, results from a different data flow path. When a user replies to a message, phpBB includes a quotation of the original message in the reply message. In the vulnerable version, phpBB also allows a user to reply to a message even if the user lacks permission to read the message. To exploit this vulnerability, an adversary, lacking permission to read a message, replies to the message using its message ID, and then reads the content of the original message, quoted in the reply template. The RESIN assertion that checks the read access controls prevents this vulnerability because the assertion detects data flow from the original message to the adversary's browser, regardless of the path taken.

A final example comes from the two password disclosure vulnerabilities shown in Table 2.4. As described in Section 5, the HotCRP disclosure results from a logic bug in the email preview and the email reminder features. In contrast, the disclosure in the myPHPscripts login library [59] results from the library storing its users' passwords in a plain-text file in the same HTTP-accessible directory that contains the library's PHP files [62]. To exploit this, an adversary requests the password file with a Web browser. Despite preventing password disclosure through two different data flow paths, the assertions for password disclosure in HotCRP and myPHPscripts are very similar (the only difference is that HotCRP allows email reminders and myPHPscripts does not). This shows that a single RESIN data flow assertion can prevent attacks through a wide range of attack vectors and data paths.

2.7 Performance Evaluation

Although the main focus of RESIN is to improve application security, application developers may be hesitant to use these techniques if they impose a prohibitive performance overhead. In this section, we show that RESIN's performance is acceptable. We first measure the overhead of running HotCRP with and without the use of RESIN, and then break down the low-level costs that account for the overhead using microbenchmarks. The overall result is that a complex Web application like HotCRP incurs a 33% CPU overhead for generating a page, which is unlikely to be noticeable by end-users.

The following experiments were run on a single core of a 2.3GHz Xeon 5140 server with 4GB of memory running Linux 2.6.22. The unmodified PHP interpreter is version 5.2.5, the same version that the RESIN PHP interpreter is based on.

2.7.1 Application Performance

To evaluate the system-level overhead of RESIN, we compare a modified version of HotCRP running in the RESIN PHP interpreter against an unmodified version of HotCRP 2.26 running in an unmodified PHP interpreter. We measured the time to generate the Web page for a specific paper in HotCRP, including the paper's title, abstract, and author list (if not anonymized), as if a PC member requested it through a browser. The measured runtime includes the time taken to parse PHP code, recall the session state, make SQL queries, and invoke the relevant data flow assertions. In this example, RESIN invoked two assertions: one protected the paper title and abstract (and the PC member was allowed to see them), and the other protected the author list (and the PC member was not allowed to see it, due to anonymization). We used the output buffering technique from Section 2.5.5 to present a consistent interface even when the author list policy raised an exception. The resulting page consisted of 8.5KB of HTML.

The unmodified version of HotCRP generates the page in 66ms (15.2 requests per second) and the RESIN version uses 88ms (11.4 requests per second), averaged over 2000 trials. The performance of this benchmark is CPU limited. Despite our unoptimized RESIN prototype, its performance is likely to be adequate for many real world applications. For example, in the 30 minutes before the SOSP submission deadline in 2007, the HotCRP submission system logged only 390 user actions. Even if there were 10 page requests for each logged action (likely an overestimate), this would only average to 2.2 requests per second and a CPU utilization of 14.3% without RESIN, or 19.1% with RESIN on a single core. Adding a second CPU core doubles the throughput.

2.7.2 Microbenchmarks

To determine the source of RESIN's overhead, we measured the time taken by individual operations in an unmodified PHP interpreter, and a RESIN PHP interpreter both without any policy and with an empty policy. The results of these microbenchmarks are shown in Table 2.5.

For operations that simply propagate policies, such as variable assignments and function calls, RESIN incurs a small absolute overhead of 4-21ns, but percentage wise, this is about a 10% overhead. This overhead is due to managing the policy set objects.

The overhead for invoking a filter object's interposition method (*filter_read*, *filter_write*, and *filter_func*) is the same as for a standard function call, except that RESIN calls the interposition method once for every call to *read* or *write*. Therefore the application programmer has some control over how much interposition overhead the application will incur. For example, the programmer can control the amount of

Operation	Unmodified PHP	RESIN no policy	RESIN empty policy
Assign variable	0.196 μs	0.210 μs	0.214 μs
Function call	0.598 μs	0.602 μs	0.619 μs
String concat	0.315 μs	0.340 μs	0.463 μs
Integer addition	0.224 μs	0.247 μs	0.384 μs
File open	5.60 μs	7.05 μs	18.2 μs
File read, 1KB	14.0 μs	16.6 μs	26.7 μs
File write, 1KB	57.4 μs	60.5 μs	71.7 μs
SQL SELECT	134 μs	674 μs	832 μs
SQL INSERT	64.8 μs	294 μs	508 μs
SQL DELETE	64.7 μs	114 μs	115 μs

Table 2.5: The average time taken to execute different operations in an unmodified PHP interpreter, a RESIN PHP interpreter without any policy, and a RESIN PHP interpreter with an empty policy.

computation the interposition method performs, and the number of times the application calls *read* and *write*.

For operations that track byte-level policies, such as string concatenation, the overhead without any policy is low (8%), but increases when a policy is present (47%). This reflects the cost of propagating byte-level policies for parts of the string at runtime as well as more calls to *malloc* and *free*. A more efficient implementation of byte-level policies could reduce these calls.

Operations that merge policies (such as integer addition, which cannot do byte-level tracking) are similarly inexpensive without a policy (10%), but are more expensive when a policy is applied (71%). This reflects the cost of invoking the programmer-supplied merge function. However, in all the data flow assertions we encountered, we did not need to apply policies to integers, so this might not have a large impact on real applications.

For file open, read, and write, RESIN adds potentially noticeable overhead, largely due to the cost of serializing, de-serializing, and invoking policies and filters stored in a file's extended attributes. Caching file policies in the runtime will likely reduce this overhead.

The INSERT operation listed in Table 2.5 inserts 10 cells, each into a different column, and the SELECT operation reads 10 cells, each from a different column. When there is an empty policy, each datum has the policy. The overhead without any policy is 229–540μs (354%–403%), and that with an empty policy is 443–698μs (521%–684%). RESIN's overhead is related to the size of the query, and the number of columns that have policies; reducing the number of columns returned by a query reduces the overhead for a query. For example, a SELECT query that only requests six columns with policies takes 578μs in RESIN compared to 109μs in unmodified PHP. The DELETE operation has a lower overhead because it does not require rewriting queries or results.

RESIN's overhead for SQL operations is relatively high because it parses and translates each SQL query in order to determine the policy object for each data item

that the query stores or fetches. Our current implementation performs much of the translation in a library written in PHP; we expect that converting all of it to C would offer significant speedup. Note that, even with our high overhead for SQL queries, the overall application incurs a much smaller performance overhead, such as 33% in the case of HotCRP.

2.8 Deployment

Another important aspect to consider for Resin is deployment. An individual Web site can benefit from adopting Resin in isolation. Resin does not require Web clients to do anything, nor does it require more than one Web site to adopt Resin before it becomes useful. For these reasons, each Web site can decide to use Resin on a case-by-case basis for its own benefit without regard for other Web sites or users. This is beneficial for a new technology like Resin, and should make it easier to penetrate the market.

2.9 Limitations and Future Work

Resin currently has a number of limitations which we plan to address in future work.

2.9.1 Data Flow Assertion Model

Data Integrity Assertions

In its current design, a data integrity assertion can only check whether a write is allowed before the write actually takes place. Therefore, the assertion must have prior knowledge of whether the write will be valid after completion. Currently, an assertion cannot permit the write to proceed and then check whether the write is valid afterward.

One way to improve support for integrity assertions is to use transactions. For example, consider a banking application, where a bank wants to ensure that all money transfers are properly authorized, and that the sum of all debits and credits adds up to zero. The execution of a request, including any database updates, file changes, and memory modifications, would be wrapped in a transaction, and Resin would not commit the updates until a policy object approves them. The policy would check that the sum of all bank account balances remains the same, and that the requesting user had permission to access each account that was touched, before committing the transaction. In a sense, this mechanism would take the integrity constraints often found in databases and run them within the application, with access to all the extra information in the application's runtime.

Internal Data Flow Boundaries

Second, Resin does not have good support for constructing internal data flow boundaries within an application. For example, it would be difficult to implement an assertion to prevent clear-text passwords from flowing out of the software module that handles passwords. Attaching filter objects to function calls is a step in the right direction, but languages like PHP and Python allow code to read and write data in another module's scope as if they were global variables. In the Resin runtimes for PHP and Python, an internal data flow boundary would need to address these data flow paths. Other runtimes, like Java's, have stronger scope enforcement, and might require fewer changes.

Server Boundary

Currently, Resin only checks assertions up to the edge of the Web server. After sensitive data leaves the server, a Web browser has access to the sensitive data, and can perform computation and communication that can violate data flow assertions. One way to address this limitation is to add Resin-like functionality to the browser, which is discussed more in Chapters 3 and 4.

2.9.2 Language Runtimes

Multiple Runtimes

Currently, Resin is limited to the PHP and Python runtimes. Although saving policies persistently allows polices to propagate to different instances of the same runtime (the Resin-aware Apache Web server invokes PHP to check policies), the policy serialization is runtime-specific. For example, in a SQL server, the SQL commands can compute on data, transform it, and save it to the database. Resin's SQL translator understands some simple SQL computations and propagates policies in those cases, but in general, Resin loses track of data within the SQL runtime. Currently, Resin's prototype SQL translator understands a few basic functions, and can compute the policy on the result of a function, like addition, before writing the result to the database, but a general solution would require the SQL runtime, and other runtimes, to be aware of policy objects.

Resin also does not propagate policies across different machines, so Resin will lose track of policies in a distributed system, like a three-tiered Web architecture. One way to address this limitation is to extend Resin to propagate policies between machines in a distributed system similar to the way DStar [90] does with information flow labels.

Runtime Modifications

Adding Resin to a runtime currently requires substantial changes to the runtime, and it might be difficult to persuade runtime developers to adopt those changes. For example, adding data tracking to PHP required modifying the interpreter in 103

locations to propagate policies; ideally, applying these techniques to new runtimes would require fewer changes.

One approach to implementing Resin with fewer changes to the runtime might be to use OS or VMM support. It might also be possible to implement Resin without modifying the language runtime at all, given a suitable object-oriented system. The implementation would override all string operations to propagate policy objects, and override storage system interfaces to implement filter objects.

Static Analysis

Dynamic data tracking adds runtime overheads and presents challenges to tracking data through control flow paths. It may be possible to use static analysis or programmer annotations to check Resin-style data flow assertions at compile time instead of at runtime. However, Resin's use of general purpose code to express assertions does pose a challenge to this approach.

2.9.3 Applications

More Security Applications

One advantage of Resin over existing data tracking systems is that Resin is general purpose and can support many different security policies. One area for future work is to explore the space of security policies and try to implement Resin policies to preserve them. Some possible candidates are HTTP response splitting and cross-site request forgery.

Non-Security Applications

Finally, this work focuses on security as the driving need for Resin, but not all data flow bugs are related to security; some bugs just produce incorrect application behavior. Programmers may be able to use Resin to catch these bugs. For example, a Web store might have the high-level assertion that goods are paid for before sending a request to the shipping department. It may be possible to capture this assertion by attaching a Resin policy object to the request, and interposing on the messaging interface to the shipping department.

2.10 Related Work

Resin makes a number of design decisions regarding how programmers specify policies and how Resin tracks data. This section relates Resin's design to prior work.

2.10.1 Policy Specification

When using Resin, programmers define a data flow assertion by writing policy objects and filter objects in the same language as the rest of the application. Previous work

in policy description languages focuses on specifying policies at a higher level, to make policies easier to understand, manage [7, 17, 21], analyze [30], and specify [3]. While these policy languages do not enforce security directly, having a clearly defined policy specification allows reasoning about the security of a system, performing static analysis [25, 24], and composing policies in well-defined ways [73, 2, 9]. Doing the same using Resin is challenging because programmers write assertions in general-purpose code. In future work, techniques like program analysis could help formalize Resin's policies [5], to bring some of these benefits to Resin, or to allow performance optimizations.

Lattice-based label systems [58, 15, 14, 22, 89, 46, 18] control data flow by assigning labels to objects. Expressing policies using labels can be difficult [21], and can require re-structuring applications. Once specified, labels objectively define the policy, whereas Resin assertions require reasoning about code. For more complex policies, labels are not enough, and many applications use trusted *declassifiers* to transform labels according to application-specific rules (e.g. encryption declassifies private data). Indeed, a large part of re-structuring an application to use labels involves writing and placing declassifiers. Resin's design can be thought of as specifying the declassifier (policy object) in the label, thus avoiding the need to place declassifiers throughout the application code.

Since Resin programmers define their own policy and filter objects, programmers can implement data flow assertions specific to an application, such as ensuring that every string that came from one user is sanitized before being sent to another user's browser. Resin's assertions are more extensible than specialized policy languages [27], or tools designed to find specific problems, such as SQL injection or cross-site scripting [37, 81, 53, 51, 76, 60, 66, 38, 83].

PQL [53] allows programmers to run application-specific program analyses on their code at development time, including analyses that look for data flow bugs such as SQL injection. However, PQL is limited to finding data flows that can be statically analyzed, with the help of some development-time runtime checks, and cannot find data flows that involve persistent storage. This could miss some subtle paths that an attacker might trigger at runtime, and would not prevent vulnerabilities in plug-ins added by end-users.

FABLE [70] allows programmers to customize the type system and label transformation rules, but requires the programmer to define a type system in a specialized language, and use the type system to implement the applications' data flow schemes. Resin, on the other hand, implements data tracking orthogonal to the type system, requiring fewer code modifications, and allowing programmers to reuse existing code in their assertions.

Systems like OKWS [45] and Privman [43] enforce security by having programmers partition their application into less-privileged processes. By operating in the language runtime, Resin's policy and filter objects track data flows and check assertions at a higher level of abstraction, avoiding the need to re-structure applications. However, Resin cannot protect against compromised server processes.

2.10.2 Data Tracking

Once the assertions are in place, Resin tracks explicit flows of application data at runtime, as it moves through the system. Resin does not track data flows through implicit channels, such as program control flow and data structure layout, because implicit flows can be difficult to reason about, and often do not correspond to data flow plans the programmer had in mind. Implicit data flows can lead to "taint creep", or increasingly tainted program control flow, as the application executes, which can make the system difficult to use in practice. In contrast, systems like Jif [58] track data through all channels, including program control flow, and can catch subtle bugs that leak data through these channels. By relying on a well-defined label system, Jif can also avoid runtime checks in many cases, and rely purely on compile-time static checking, which reduces runtime overhead.

Resin's data tracking is central to its ability to implement data flow assertions that involve data movement, like SQL injection or cross-site scripting protection. Other program checkers, like Spec# [7, 6], check program invariants, but focus on checking function pre- and post-conditions and do not track data. Aspect-oriented programming (AOP) [77] provides a way to add functionality, including security checks, that cuts across many different software modules, but does not perform data tracking. However, AOP does help programmers add new code throughout an application's code base, and could be used to implement Resin filter objects.

By tracking data flow in a language runtime, Resin can track data at the level of existing programming abstractions—variables, I/O channels, and function calls—much like in Jif [58]. This allows programmers to use Resin without having to restructure their applications. This differs from OS-level IFC systems [22, 89, 88, 46] which track data flowing between processes, and thus require programmers to expose data flows to the OS by explicitly partitioning their applications into many components according to the data each component should observe. On the other hand, these OS IFC systems can protect against compromised server code, whereas Resin assumes that all application code is trusted; a compromise in the application code can bypass Resin's assertions.

Some bug-specific tools use data tracking to prevent vulnerabilities such as cross-site scripting [42], SQL injection [60, 76], and untrusted user input [72, 64, 13]. While these tools inspired Resin's design, they effectively hard-code the assertion to be checked into the design of the tool. As a result, they are not general enough to address application-specific data flows, and do not support data flow tracking through persistent storage. One potential advantage of these tools is that they do not require the programmer to modify their application in order to prevent well-known vulnerabilities such as SQL injection or cross-site scripting. We suspect that with Resin, one developer could also write a general-purpose assertion that can be then applied to other applications.

2.11 Summary

Programmers often have a plan for correct data flow in their applications. However, today's programmers often implement their plans implicitly, which requires the programmer to insert the correct code checks in many places throughout an application. This is difficult to do in practice, and often leads to vulnerabilities.

This work takes a step towards solving this problem by introducing the idea of a data flow assertion, which allows a programmer to explicitly specify a data flow plan, and then have the language runtime check it at runtime. RESIN provides three mechanisms for implementing data flow assertions: *policy objects* associated with data, *data tracking* as data flows through an application, and *filter objects* that define data flow boundaries and control data movement.

We evaluated RESIN by adding data flow assertions to prevent security vulnerabilities in existing PHP and Python applications. Results show that data flow assertions are effective at preventing a wide range of vulnerabilities, that assertions are short and easy to write, and that assertions can be added incrementally without having to restructure existing applications. We hope these benefits will entice programmers to adopt our ideas in practice.

Chapter 3

BFlow

Some web sites provide interactive extensions using browser scripts, often without inspecting the scripts to verify that they are benign and bug-free. Others handle users' confidential data and display it via the browser. Such new features contribute to the power of online services, but their combination would allow attackers to steal confidential data. This chapter presents BFlow, a security system that uses information flow control to allow the combination while preventing attacks on data confidentiality.

BFlow allows untrusted JavaScript to compute with, render, and store confidential data, while preventing leaks of that data. BFlow tracks confidential data as it flows within the browser, between scripts on a page and between scripts and web servers. Using these observations and assistance from participating web servers, BFlow prevents scripts that have seen confidential data from leaking it, without disrupting the JavaScript communication techniques used in complex web pages. To achieve these ends, BFlow introduces an information flow control model for the JavaScript environment, a mapping from that model to the browser's existing security mechanisms, and a new "protection zone" abstraction.

We have implemented a BFlow browser reference monitor and server support. To evaluate BFlow's confidentiality protection and flexibility, we have built a BFlow-protected blog that supports Blogger's third party JavaScript extensions. BFlow's blog is compatible with every legitimate Blogger extension that we have found, yet BFlow prevents malicious extensions from leaking confidential data. We have also built a social networking site that supports third-party JavaScript, and a Web application platform that allows applications to share user data without leaking that data.

3.1 Introduction

Three important trends in Internet-based computing have emerged in recent years. First, Web sites are increasingly hosting sensitive user data and applications; hosted e-mail has been joined by other applications, such as hosted spreadsheets and confidential blogs. Second, large swathes of Web user interface code now run in the browser, as JavaScript and other browser scripting languages. Third, many Web

sites use JavaScript that they might not fully understand, including large imported libraries and even extension scripts written by arbitrary third-party programmers. These extensions can use server-side APIs to access and manipulate users' server-based data, giving rise to application-like third-party extensions on "platform" sites such as Facebook [26] and Blogger [12].

The combination of third-party browser scripts and sensitive user data raises the possibility of scripts stealing confidential data. For this reason, today's Web applications that value user privacy must forbid browser script extensions, or refuse to reveal sensitive user data to extensions. These approaches cut off useful behavior, undermining the value of extensibility. For example, Web applications like Gmail would benefit from third-party JavaScript extensions, but confidentiality problems make them difficult to support. As a substitute, Gmail users modify their *browsers* to do things like optimize Gmail's UI for particular mobile devices and alter the way Gmail renders email [28].

Existing Web sites that support extensions tend to do so with less sensitive, but still confidential data. For example, the Blogger Web site hosts confidential blogs, yet permits users to install third-party JavaScript extensions, that they might not fully understand, on their blogs. These extensions can read confidential data, compute on it, and display it to the user (which is reasonable by itself), but they can also communicate any information they read to outside parties (which can violate the user's privacy). Part of the underlying problem is that the browser security policy gives all scripts that come from a given Web site full privileges with respect to that site.

Recent work [80, 55, 41] proposes improvements to today's browser security policy such as finer-grained separation of privileges between different parts of the browser. But these solutions still force users or developers to make up-front decisions as to whether or not to trust third-party code with confidential data. Mistakenly deciding "no" inhibits extensibility; mistakenly deciding "yes" invites data theft.

This chapter describes BFlow, a new browser security system. BFlow lets browser scripts compute with confidential data while restricting their ability to reveal that data. BFlow uses a reference monitor in the browser to enforce information flow control (IFC), observing the communication of each script with other scripts and with Web sites. These observations help BFlow decide whether each script has seen confidential data (whether directly or transitively through another script) and from what site that data came. The BFlow reference monitor uses the tracking information to restrict how data is revealed: if a script has seen confidential data, it can only communicate with the site whence the confidential data came unless that site explicitly permits communication with other servers. BFlow places few new restrictions on scripts that have not been exposed to confidential data. To take advantage of BFlow, a Web site must cooperate by marking outgoing confidential data with security metadata and recording the confidentiality of incoming data.

The challenges in designing BFlow differ from those solved by operating system IFC systems [11, 54, 19, 22] because the browser has somewhat unusual notions of the principals that own data (Web sites), of the natural code unit at which to apply IFC

(the frame), and of the special flows of information that must be supported (among frames and to Web servers).

We have implemented a prototype BFlow browser reference monitor as a Firefox plug-in. We have also implemented the server part of BFlow as a gateway layer that sits between an Apache Web server and the Web site's application logic. These implementations are intended to be easy to deploy: the Firefox plug-in is easy to install, and the BFlow reference monitor supports the full JavaScript language so that most scripts run with no changes.

To evaluate BFlow's privacy protection and flexibility, we implemented three Web sites that incorporate third-party JavaScript: a Web site compatible with Blogger's third-party extensions, a social networking site that implements common application features in untrusted JavaScript, and Web platform that supports third-party server applications that share confidential user data. The blog example shows that many existing scripts will work with few modifications and that malicious JavaScript that leaks confidential data in Blogger does not leak within BFlow. The social network example shows that BFlow supports a wide range of third-party functionality, and the Web platform demonstrates how developers can use BFlow to build new kinds of Web architectures.

The contributions of this work are a set of information flow control rules that govern the JavaScript communication mechanisms, a mapping from BFlow's IFC rules to the browser's existing JavaScript isolation system, and an abstraction called a protection zone that eases the deployment of existing JavaScript into BFlow. Together, these techniques allow untrusted JavaScript to read, compute with, and display confidential data without the risk of leaking that data.

3.2 Background: JavaScript

Web sites use in-browser JavaScript to provide high-quality user interfaces. This section briefly reviews what JavaScript can do within a browser, focusing on communication.

A browser consists of one or more *frames*, each containing a separate HTML document and JavaScript interpreter. Browser frames can contain sub-frames using the `frame` and `iframe` HTML directives. Each browser window or tab is a *top-level* frame, each frame that embeds a sub-frame is a *parent*, and each sub-frame is the *child* of its parent.

The browser represents the displayed document in each frame as a data structure called the Document Object Model (DOM). JavaScript code is allowed to read and modify the DOM of any frame from the same origin server as the code.[1] Two JavaScript scripts, each running in a different frame, but from the same origin, can communicate with each other via modification to each other's DOMs. Also, JavaScript can communicate with any Web server by fetching a Web document, including HTML pages and images, from that server.

[1] An origin is defined as a triple: domain name, protocol, and port.

The restriction that JavaScript only access DOMs from the same origin is called the *same-origin policy* (SOP). The SOP also only allows a script to send AJAX requests to its origin server. The high-level goal of the SOP is to guard the operation of each Web site and its JavaScript from interference by other sites' JavaScript. The SOP does not restrict JavaScript from interacting with different-origin sites in a number of ways which would be unlikely to interfere with their proper operation. For example, a script can modify its frame's document to fetch an image from any Web site, which allows the script to communicate with the site through the name of the requested image. The SOP also allows scripts to use JavaScript's *intra-browser* channels to send messages to listening scripts from any origin. The result is that scripts that have access to confidential data can leak that data to cooperating outside Web sites and JavaScript.

3.3 Challenges

BFlow requires a stronger policy than the SOP because it must prevent data movement even when untrusted scripts and untrusted servers collude against the user's wishes. BFlow must accomplish this while maintaining support for untrusted JavaScript extensions without encumbering deployment.

3.3.1 Threat Model and Security

BFlow applies to Web sites that both store confidential user data and allow untrusted JavaScript to access that data. The adversary's goal is to read, with his own eyes, data that he should not be able to read according to the Web site's stated confidentiality policy. The adversary's capabilities are limited to creating his own accounts on the Web site, running his own Web servers, and writing JavaScript which the site includes in pages viewed by other users. Neither the site operators nor the users inspect the adversary's JavaScript.

More general adversaries might have other tools at their disposal. They might: compromise the host site; eavesdrop on or corrupt network traffic; infect the user's operating system with malware; infect the user's browser with malware; and use social-engineering attacks like "phishing" to lure the user or her friends into giving confidential data away. BFlow does not defend against these attacks, and its correct operation depends on adequate defenses to them that are outside the scope of this work (e.g. SSL, timely application of O/S security patches, etc.).

The ability to inject arbitrary JavaScript into a page is quite powerful and is commonly referred to as a cross-site scripting vulnerability. While BFlow does not aim to solve all attacks available through XSS, it does aim to prevent XSS attacks from leaking confidential data.

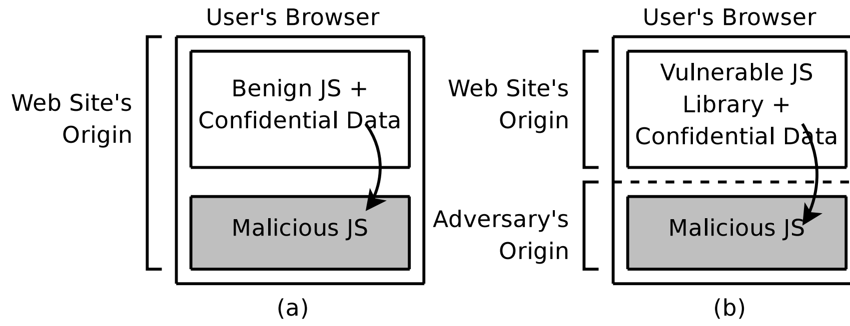

Figure 3-1: Malicious JavaScript reads confidential data (a) via the DOM and (b) by exploiting vulnerable JavaScript.

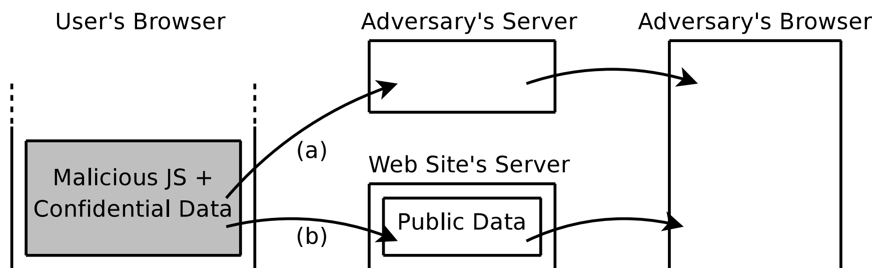

Figure 3-2: After reading confidential data, the malicious JavaScript leaks confidential data to an adversary via the (a) adversary's server (b) Web site's public data.

Attack Paths

Once the adversary injects JavaScript into the Web site's pages and a user views a page, the JavaScript can attempt to read the confidential data displayed on the page and leak it to the adversary.

There are two possible scenarios for reading the confidential data in this model. In the first case, the malicious JavaScript runs in the same origin as the confidential data. This could occur for many reasons; today, Web sites incorporate large JavaScript libraries like Scriptaculous [69] or Google Maps [34] into their site's origin and platforms like Blogger.com inline completely unaudited third-party scripts. In this case, the JavaScript can read the confidential data directly from the DOM as shown in Figure 3-1a. In the second case, malicious JavaScript can steal data even if there is no malicious code in the same origin as the confidential data. Today's browsers now support intra-browser communication between scripts from different origins and developers are already building libraries to use these channels [75]. If libraries like these are buggy, then malicious JavaScript running in the browser from a different origin (and a different frame) could exploit their bugs to read the confidential data as in Figure 3-1b.

After reading confidential data, the malicious JavaScript can send it to a Web server using an HTTP request, either to the Web site's own server or to a third-party *external* server. For example, the JavaScript can encode the data in an image name to be fetched from a server the adversary controls (see Figure 3-2a).

Even if the same-origin policy applied to all types of requests and the script could only send HTTP requests to the Web site's server, the malicious JavaScript could leak data via the Web site's own server. The malicious script could craft an HTTP request that stores the confidential data back onto the server in a public area. Since the server no longer realizes that the data is confidential, the adversary can read it with his own browser (see Figure 3-2b). Similarly, malicious JavaScript could write confidential data into a browser cookie and then any other code that comes from the same domain could read the data.

3.3.2 Flexibility and Adoption

The second challenge is to design a system that is easy for developers, Web sites, and users to adopt.

One aspect of this challenge lies in preventing data leaks while preserving features popular among JavaScript developers, such as `eval()`, communication among concurrent browser scripts, and communication with remote Web servers. This last JavaScript use is particularly commonplace and dangerous. Today's browser scripts routinely load images and data from multiple independently-administered servers. In the context of BFlow, such requests can encode confidential information. If one considers (as one should) a large majority of Web servers to be untrustworthy receptacles for data leaks, BFlow must block requests (e.g, image loads) to such servers by scripts privy to confidential information. At the same time, BFlow can allow such requests from scripts that have not seen confidential data. In sum, BFlow should allow harmless requests to external servers, allow requests that release information if the release is the intention of the site owning the data, and detect and forbid accidental or malicious releases.

The design of BFlow should also be easy for users to install, site developers to adopt, and extension developers to adopt (in that order of priority). Some level of complexity is inevitable, but the goal is that deployment effort should be limited to: 1) users installing a browser plugin, 2) site developers deciding which data on their site is confidential and rearranging the site's HTML to partition data by confidentiality constraints, and 3) third-party developers designing extensions that handle confidential data to live within BFlow's communication restrictions.

3.4 Design

The goal of BFlow is to enforce two properties on how a browser handles data. First, if confidential data arrives from a Web site, only the human user and that origin Web site should see any information derived from the data unless the site specifically allows it to go to another Web site. Second, if the browser sends information derived from confidential data to the origin Web site, the information must be marked as confidential unless the site specifically allows the removal of the confidentiality marking. The main tension in the BFlow design is the enforcement of these properties in a way that is compatible with how developers use JavaScript in complex Web pages.

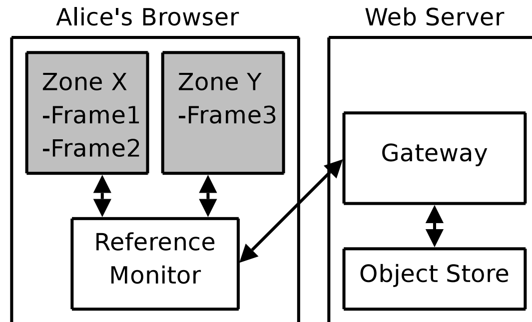

Figure 3-3: BFLOW overview. Untrusted protection zones are shaded.

In outline, the BFLOW design is as follows. The BFLOW browser reference monitor watches how data flows into, out of, and within the browser. A BFLOW-aware server sends a *label* along with data it sends to the browser to tell the reference monitor whether the data is confidential. The reference monitor uses a form of information flow control [18] to enforce a confidentiality policy, tracking what data within the browser might be derived from confidential data. Each browser script runs in a browser frame, and frames are grouped into *protection zones*. BFLOW tracks data at the granularity of a protection zone (see Figure 3-3). When data is about to leave the browser via the network, the reference monitor enforces a safety property on the data's label; if the data is going to its origin Web site, the reference monitor includes the label; otherwise, if the label indicates the data is confidential, the reference monitor forbids its release unless an explicit *declassification* exception applies.

3.4.1 Information Flow Control

The BFLOW reference monitor's information flow control system keeps track of what categories of confidential data the JavaScript in each protection zone may have seen. The reference monitor (RM) maintains a *label* for each zone. A label is a set of *tags*. A tag is an opaque token supplied by a server that indicates a particular category of confidential data. The meaning of a zone having a label containing a tag t is "the JavaScript or HTML in this zone may have observed information derived from data with confidentiality category t." A label with multiple tags indicates that the zone may have observed data in multiple confidentiality categories.

To ensure that a zone's label reflects the categories of confidential data it has seen, the RM enforces some rules relating to communication across zone boundaries. The effect of the rules is that, if information is to flow from zone S to zone R, R's label must be a superset of S's. In the special case of data flowing from a server to a zone, the zone's label must be a superset of the label provided with the data. Table 3.1 summarizes this and BFLOW's other IFC rules described below.

A zone explicitly asks to change its own label and specifies which tags to add; BFLOW does not automatically change zone R's label in response to the data R receives. BFLOW always permits a zone to add any tag to its label. This is safe because the communication rules described above get strictly more restrictive as the

Sender	Receiver	Default Rule	Exception
Script in trusted zone	Any	Allow	N/A
Script in zone S, frame F, from server W	Script in W's trusted zone	Allow	N/A
	Script in zone S	Allow	N/A
	Script in zone R, sub-frame of F	$L_S \subseteq L_R$ (always true)	N/A
	Script in zone R, not sub-frame of F	$L_S \subseteq L_R$	Trusted zone proxy.
	Source server of W	Allow	N/A
	External server E	$L_S = \{\}$	$L_S \subseteq D_E$
Source server W sending data with label L	Script in W's trusted zone	Allow	N/A
	Script in zone R	$L \subseteq L_R$	None

Table 3.1: Default IFC communication rules and declassification exceptions; zones S and R are untrusted. The prototype implements these rules for communication through `postMessageBF`, the FID channel and HTTP requests, but it is more restrictive than these rules for shared DOM variables and cookie communication across zones.

sender's label grows. In practice, BFlow adds some further restrictions which we describe in Section 3.4.2. The RM imposes the IFC rules inside user u's browser to prevent buggy or malicious scripts from leaking u's data. At the same time, it is the server's responsibility to avoid sending data to u's browser that u is not permitted to read because u could have modified her browser to extract all the data available to it.

The ultimate source of each tag is a particular BFlow-aware Web site. The browser RM internally adds the source server identity to each tag so that two tags from different servers are always unique. In typical use, a zone's label will either be empty (indicating that the zone has seen no confidential data) or contain just one tag. A label might contain multiple tags if a zone has consulted multiple categories of confidential data. A zone's label cannot contain tags from different Web sites because it would violate the flow invariant described in Section 3.4.2

A Web site decides what its tags mean. A typical Web site might associate a different tag with each user, or a tag with each category of confidential data a user owns. For example, a Web site might store both a confidential photo album and a confidential blog for user Alice, and associate a different tag with each kind of data. Then, if the site sends blog data to Alice's browser, and some JavaScript that examined the data communicates with the site, the site will know that the communication (and any resulting stored data) should have the same tag as Alice's confidential blog.

3.4.2 Protection Zones

One of the challenges in designing an information flow model for JavaScript comes from how developers use JavaScript today. Often, developers will construct Web pages out of many sub-frames, each containing its own JavaScript. Furthermore, within a single page different sub-frames may have different purposes. For example, a top-level page may contain a chat tool and an email tool, each contained in its own individual sub-frame. Each of those tools may in turn contain its own sub-frames. For example, the chat tool may use two separate sub-frames, one for showing messages and one for data input.

Existing multi-frame modules like the chat tool typically read shared variables and call functions across frame boundaries. Modules expect these features to be reliable, so BFlow should accommodate this behavior; if one sub-frame in the module reads confidential data, then it should still be able to communicate with the other frames in the module without excessive coordination. BFlow addresses this challenge by applying IFC at the granularity of a protection zone.

A protection zone is a group of one or more browser frames, including their DOMs and the JavaScript running inside of them, plus its own set of browser cookies. All the scripts and data within a zone share a common label. Grouping frames into zones gives developers an easy way to modularize their scripts. Once the scripts are in a common zone, they can communicate with each other regardless of any label changes, even if a script in one of many sub-frames changes the zone's label unilaterally.

A Web site also has a special *trusted* zone which always has an empty zone label; JavaScript running in the trusted zone can bypass BFlow's browser constraints. A Web site uses the trusted zone in cases where confidential data is allowed to leave the system by a browser script, but the Web site developers must inspect such scripts carefully.

To create a new zone, JavaScript in an existing zone requests a new zone id from BFlow and then loads a document from the server (specifying the new zone id) into one of the zone's existing frames. When the HTTP response arrives, the RM recognizes that the zone id is new, and creates its local representation of the zone. However, not all frames have their own zone; when a parent creates a sub-frame, by default the RM places the sub-frame in the same zone as the parent as shown by $Z1$ in Figure 3-4.

Flow Invariant

BFlow maintains a *flow invariant* over the browser's frames and zones: first, the browser's top level frame must be in the trusted zone and all its sub-frames must be able to legally send messages to the top level frame. Second, if a parent frame P has child frames C_i, then the P must be able to send messages to each of its children legally. More specifically, if P has label L_P and P's children have labels L_{C_i}, then $\forall i, L_P \subseteq L_{C_i}$. This invariant must hold regardless of what zone each frame is a member of. The BFlow RM preserves the flow invariant by checking the target frame F and target zone Z before changing a zone's label.

When a zone Z changes its label, all other scripts running in Z will have the new label even if they are running in other frames; no zone other than Z will experience a label change. However, adding t to L_Z may permit another zone Z_P to add t to its label because of the invariant, if adding t to L_Z means all of Z_P's children now contain t.

Maintaining the invariant slightly limits the kinds of frame hierarchies possible: an untrusted frame cannot contain tags from different Web sites and a parent frame with $L_P = \{t\}$ cannot contain a child frame with $L_C = \{\}$, but it ensures that BFlow can support existing methods of JavaScript communication described in Section 3.4.3.

3.4.3 Controlling Intra-browser Communication

Tracking the flow of confidential data between scripts within the browser is critical to preventing leaks because BFlow can only prevent a script from leaking data if it knows what data the script has seen. This section describes which channels are available in BFlow between scripts in the same zone and in different zones. We focus on the Firefox 3.0 browser in which JavaScript has four techniques to communicate between scripts (other browsers may have other techniques). They are DOM variables, browser cookies, the `postMessage` channel, and the fragment-ID (FID) channel.

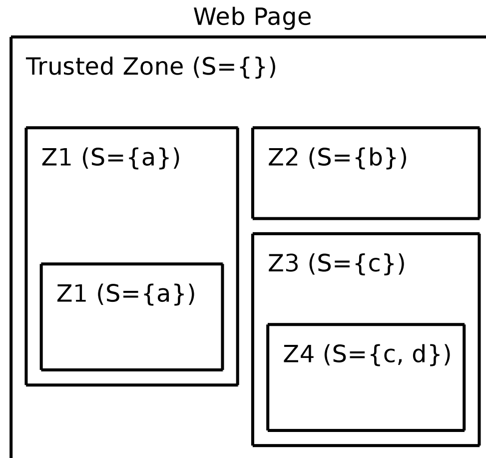

Figure 3-4: Web page frame hierarchy with zones and labels. Each box is a frame.

Within One Zone

BFlow need not restrict communication between two scripts in the same zone, since all of the JavaScript, frame DOMs, and cookies within a zone share the same zone label. It is important that BFlow accommodates scripts from different frames that read and write each other's DOM variables, since many sites have scripts that use that feature.

Between Two Zones

Since two scripts in different zones can have different labels, BFlow must restrict communication between two such scripts according to the IFC rules shown in Table 3.1. It does so through a combination of unconditionally forbidding some operations between scripts from different zones, and allowing other operations only when the zone labels allow.

Although today's browsers allow scripts in the same origin to read and write each other's DOMs, BFlow unconditionally forbids JavaScript in two different zones from reading or writing each other's DOM variables or cookies. This is a conservative restriction due to our implementation and the only restriction BFlow places on code that has not seen confidential data. A better implementation would allow a script in zone S to write to variables and cookies in zone R if R's label were a superset of S's label.

Instead of using shared DOM variables and cookies, BFlow allows scripts in different zones to send explicit messages to one another using an API function called `postMessageBF`. To preserve the IFC rules, the RM only delivers the message if the sender's label L_S is a subset of the receiver's label L_R; if not, it will drop the message. BFlow's `postMessageBF` replaces the `postMessage` API found in HTML5 because `postMessage` does not enforce the IFC rules.

The fourth intra-zone communication method is the FID channel which is an artifact of a script's ability to set the location of both its sub-frames and the top

level frame. Setting the location of frame F communicates data to frame F [8]. BFlow does not specifically restrict the FID channel; instead, BFlow ensures that any use of the FID channel is legal according to the IFC rules in Table 3.1 because BFlow preserves the flow invariant. Without the invariant, a sub-frame P with label $L_P = \{t\}$ that has read confidential data could leak it to a child frame C with $L_C = \{\}$ that does not have the proper label, i.e. $L_P \nsubseteq L_C$.

These IFC rules alone might be too strict for an untrusted script that handles both confidential and public data, and also needs a way to reveal the public data. For example, an untrusted script might need to read a user's confidential email address with label $L = \{t\}$ and also need to save public data with $L = \{\}$ to the server. BFlow supports this using an exception to the strict IFC rules called *browser declassification*. BFlow permits a script running in a zone from server W to send messages to scripts in the trusted zone of server W and vice-versa, so the untrusted script with label $L_R = \{\}$ can request the email address from the trusted script and the trusted script can respond with the email address despite R's label if the site developers allow it to.

3.4.4 Controlling Browser-Server Communication

In addition to data flowing within the browser, data can also flow between the browser and Web servers in HTTP requests and responses. To track these flows, the BFlow reference monitor interposes on requests sent out by the browser and on responses that arrive at the browser. When handling an HTTP request from a zone that has seen confidential data from server W, BFlow treats the *source* server W differently from any other *external* server E_i. Since W sent the confidential data in the first place, BFlow can safely send HTTP requests containing the confidential data back to W. Sending to any other server E_i requires a declassification exception, whether E_i is BFlow aware or not.

Source Server Protocol

For communication between the browser and the source server, the BFlow RM and the server include labels in each HTTP request and response. The server labels responses so that the browser RM will know what label to apply to each zone. Similarly, the browser RM labels requests so that the server will know what data is confidential otherwise, attacks like that shown in Figure 3-2b might succeed.

When a browser script makes an HTTP request, the BFlow RM sets the label of the request equal to the script's zone label, i.e. $L_{req} = L_{zone}$. Labeling the request according to the script's label ensures that the server will know what confidential data the request may contain. If the request causes the server to store data, the server should store the label along with the data and return the label if a subsequent request reads it.

By default, the server's HTTP response will have the same label as the request ($L_{req} = L_{resp}$). This ensures that any confidential data contained in the request will propagate to the response and the label of the zone that receives the response will reflect the confidential data in its label. To avoid inappropriately leaking confidential

data, the server should not use any data with tag t to generate the response unless the response's label will contain t.

Also, since any user's browser can ask to add t to a zone's label (including users who do not have permission to read data with tag t), before sending data with tag t to the browser, the server first checks whether the user logged into the browser has permission to read the data.

In addition to asking the RM directly, a script can also add a tag t to the target zone's label as part of an HTTP request. This allows a parent frame to load a page into one of its sub-frames in a different zone with a different label. It is short-hand for first loading a script into the sub-frame, having the sub-frame change its own label and then requesting the additional confidential data. The server then adds t to the response's label $L_{resp} = L_{req} \cup \{t\}$. This method only works if the frame that makes the request has permission to load a page into the target frame which implies that the requester can send a message to the target; either the two frames are in the same zone, or the target frame is a sub-frame of the requester.

Propagating the information flow labels to the server and back ensures that the client cannot leak data by bouncing it off the server. In IFC terms, if a script in zone X tries to send data to zone Y via an HTTP request through the server, the RM will update Y's label with the server response's label $L_Y \leftarrow L_Y \cup L_{resp}$ and therefore the communication will abide by the IFC rule $L_X \subseteq L_Y$.

External Servers

BFLOW forbids communication from scripts that have seen confidential data to external servers, conservatively assuming that they are not trustworthy. This applies both to image loads and to AJAX requests. The RM permits a script to send a request to an external server if the script has not seen confidential data.

This rule is too restrictive for some Web sites. Applications such as mashups may need to request data from external servers in a way that the request itself necessarily leaks confidential information. In such sites, the developers can create a *request declassification* rule which allows certain kinds of confidential data to exit to certain external servers.

For example, a Web site W might want to fetch the weather forecast for a user based on the user's postal code even though the postal code is confidential. If W's developers trust the weather server E enough to reveal its users' postal codes, then W can add a request declassification rule that says "any data tagged with tag t_i may be sent to E" and BFLOW will permit scripts that have read data with t_i (but only t_i) to send HTTP requests to E. More precisely, the site administrator would add t_i to E's declassification set D_E (see Table 3.1).

3.5 Visible Model

Developers and users must understand some aspects of BFLOW.

3.5.1 Developer Visible Model

Labels

An application developer must create a labeling scheme for the application's data, an arrangement of the application's HTML and scripts into frames and zones, and plan for labeling the zones. Zone labels are usually predictable: for example, the developer knows that a certain frame will display the user's confidential postal address and that its zone will always have exactly the corresponding label. This predictability prevents unexpected increases in labels and surprise violations of BFlow's rules.

How many tags a site uses and what the tags correspond to are largely application-specific, and BFlow does not prescribe any particular approach. In general, for each collection of data that some users and/or some external sites should be able to see, but others should not, it is likely that a tag should be associated with that data. Many sites will have a handful of tags for each user, for example one for the user's contact details and one for the user's confidential blog.

Frames

A typical BFlow Web page will consist of several frames. The top level frame will always be in the trusted zone. It will have sub-frames, each with a zone and label, to contain untrusted scripts. Scripts that need to see different kinds of confidential data will be in separate zones. A particularly common case will be separate frames that display images from external servers but handle no confidential data, and frames that handle confidential data. Existing applications may need to re-factor their HTML in order that scripts that handle data with different confidentiality tags are in separate frames and zones.

As an example, a page that allows a user to edit both his confidential phone number and his public personal profile would contain two frames in separate zones: one containing the phone number, and one containing the personal profile. Because the zones are separate, the user can edit his profile without the risk of a script reading the confidential phone number and inserting it into his public profile.

Data that the user enters into a form field takes on the label of the zone surrounding the field. Thus, even if a frame does not initially contain confidential data, if the frame contains a form field into which the developer knows the user may enter confidential data, the developer should put the field in an appropriately labeled zone.

Developers can also privilege-separate large pieces of code into a small portion running in a trusted zone and a large portion running in an untrusted zone. The two portions can communicate using browser declassification. For example, the trusted portion could provide a limited API to access external Web servers.

Linking

If an untrusted page has not seen confidential data, it can link to external Web sites, but if it has seen confidential data, it can only link to external Web sites if the destination server has a request declassification rule.

Since the top level frame in a BFlow Web page must be in the trusted zone, when an untrusted page with label $L = \{t\}$ loads a new page into the browser's top level frame, the BFlow does not propagate tag t to the top level frame. Since this is equivalent to declassifying the t tag, the trusted page should not transmit any unique data from the HTTP request such as `POST` parameters to an untrusted frame unless its label also contains t.

Confidential Data and External Servers

As described in Section 3.3.2, today's browser scripts sometimes load images and data from external servers after seeing confidential data.

One example of this is a confidential blog page that loads a static background image from an untrusted photo Web site E. Since the HTML contains confidential data and JavaScript, BFlow cannot determine if the request for the image has been influenced by confidential data or not. If the script requested the image after computing on the confidential blog content, the HTTP request would be leaking data to E. However, in this scenario, the image that the page is loading is static and is not based on the confidential data. To build such a page, the site developer can pre-declare a set of external Web documents which BFlow prefetches directly from the external servers and then caches on the blog's server. Since the requests have not been influenced by confidential data, they will not leak any data to the external servers. When the browser loads the image, it fetches it from the blog server, not the photo server E, thus decoupling the request made by the browser from the request that arrives at the photo server and protecting the blog's confidential content.

Prefetching does not work for all Web applications: a script may not know what data it needs until after reading confidential data, or the potentially-needed data may be too large to prefetch. For example, a mashup script that displays a user's location on an externally-fetched map will not know what map images to fetch until after it reads the confidential address. In this type of mashup, BFlow cannot protect the privacy of the addresses from the map server. However, keeping the address confidential is an unrealistic security requirement because the map server cannot function efficiently without the address. A more realistic security requirement is that the mashup only sends the confidential address to the map server, and not to other external servers. BFlow can enforce this requirement using request declassification as described in Section 3.4.4.

Script Changes

Depending on the Web site, untrusted scripts and libraries may or may not need to understand the information flow system. For some Web sites, the site programmers may be able to determine what label an untrusted script should run with, so that the untrusted script need not be aware of BFlow. For example, if a Web site imports a JavaScript library like Scriptaculous [69] and never expects the library to contact external servers or communicate with different zones, the site could just use the correct non-empty label and import the library without modifications. For scripts

that only read data and render it to the user, the site can just load the script with a label containing all the tags the user can read.

Server Code

A server that supports BFlow scripts must be able to record the label of data arriving from a script, and emit that label when it later serves the same data to a script. A straightforward approach is to store a label with each file or database entry. Though not necessary, it might also be helpful for the server to use an IFC-aware operating system or server framework [22, 89, 46].

Debugging

To debug applications written for BFlow, developers test their HTML and JavaScript in a BFlow-enabled browser which reports error messages pertaining to BFlow's information tracking system.

3.5.2 Users Visible Model

End users interact with a BFlow site much like they do with Web sites today. Depending on the Web site, a user may need to understand that a sub-frame may have a different privacy policy from the rest of the page. For example, a Web site that includes confidential content may also include an untrusted JavaScript widget running in a sub-frame that has not read confidential data. In this case, it is the Web site's responsibility to indicate to the user that any data he types into the sub-frame may be visible to the public. This responsibility is more explicit in BFlow, but it already exists in any Web site that includes content from untrusted programmers whether using sub-frame isolation or not.

3.6 Implementation

BFlow requires browsers to confine browser JavaScript into protection zones and to exchange security metadata with servers in each HTTP request. Since today's browsers do not implement these features, and replacing the installed base of Web browsers is difficult, the major challenge in implementing BFlow is making it easy to deploy to browsers.

3.6.1 Client Implementation

To ensure that our BFlow client modifications are easy to install for end users, we implemented the client-side reference monitor as a Firefox 3 plugin. The plugin is a portable JavaScript and XML package that runs on any platform that supports Firefox 3; users can install the plugin with only two mouse clicks. Firefox does not provide many security related hooks in the plugin interface, but it does implement the same-origin policy which provides fairly strong isolation between different origins. The

prototype plugin's implementation currently works only with Firefox's plugin API, but it should be possible to implement similar plugins for other browsers.

The BFlow plugin takes advantage of the existing SOP in the browser to implement basic isolation between protection zones. It associates each zone with a unique unforgeable domain name, and each different BFlow Web site has its own disjoint set of zone domain names. Zone domains are of the form `Z.site` where `Z` and `site` are the respective unique names of the zone and Web site. BFlow uses the form `Z.site` rather than `Z.site.com` because browsers permit a script to remove its host prefix from its domain name before the SOP comparison; using `Z.site.com` would allow two scripts with zones `Z1.site.com` and `Z2.site.com` to remove `Z1` and `Z2`, and thus communicate based on the common name `site.com`.[2] Separating zones into different domains uses the SOP to prevent scripts in one zone from reading and writing DOM variables and cookies in another zone.

However, the SOP alone does not prevent JavaScript in two different zones from colluding to leak confidential data; a script in one zone can communicate with a script in another zone using cross-domain channels like the fragment-ID channel and `postMessage` described in Section 3.4.3. BFlow's Firefox plugin disables `postMessage`, and the flow invariant described in Section 3.4.2 ensures that all available fragment-ID channels in Firefox 3 are also legal data flow paths according to BFlow's information flow rules. The BFlow prototype relies on the FID descendant policy in Firefox 3 and other recent browsers that limits the channel to parents sending data to children and frames sending data to the top-level frame [8].

When the browser makes an HTTP request to a zone domain on a BFlow aware server W, the browser RM directs the request to a Web proxy server running on W which then forwards it to an Apache Web server process on W. Using a proxy prevents the browser from attempting to resolve the zone's DNS name which is not an actual DNS domain name; however, the proxy is specific to our prototype and the same functionality could be built into the Web server.

The browser plugin is 1003 lines of JavaScript and 89 lines of XML including comments. To intercept HTTP requests for inspection and modification we use Firefox's "http-on-modify-request" and "http-on-examine-response" hooks in its XPCOM observer service. These hooks are called before sending each HTTP request and before returning the response to the rendering engine respectively.

3.6.2 User Authentication

A user can initially authenticate himself to a BFlow site using any technique, but any script used in a login Web page should be a trusted script. It could be possible to use an untrusted script on the login page with a tag to protect the password data, but the site would need to generate a new tag for each login attempt, or else a script could transmit the username and password to another user that attempts to log into the system later.

[2]The RM uses Firefox's SOP implementation, so it handles domains like `cnn.co.uk`.

After logging in, the user authenticates each subsequent HTTP request using an authentication cookie. The cookie is confidential data, but BFlow does not protect it using the information flow system because the browser must authenticate the user for all HTTP requests, even requests for public data where $L = \{\}$, so the cookie cannot have its own tag, otherwise a public page would also be protected by the cookie's tag. Instead, BFlow associates the cookie with the Web site's real domain name, for example, `site.com`.

Untrusted JavaScript running in a protection zone cannot read the Web site's authentication cookie because the untrusted zone's domain is of the form `Z.site` and the authentication cookie is from the domain `site.com`. Since the domains do not match, or share a suffix, the same origin policy prevents the untrusted JavaScript from reading the authentication cookie. However, a standard browser will not send the authentication cookie for requests originating from `Z.site` for requests to `site.com` because of the SOP, so the BFlow RM attaches the cookie to these HTTP requests.

3.6.3 Server Implementation

In the BFlow prototype, the server implements the interface described in Section 3.4.4 with server processes called gateways. The client sends raw tag values to the server in the headers of each HTTP request, and the server response with tag values in the response headers.

The server uses a gateway process to handle each request which in turn invokes application logic. The gateway launches the application logic with the read privileges of the user, so it can only read the data that the end user may read. This ensures that the user will not receive data he does not have permission to read.

Although it is not necessary for a BFlow server to use an IFC operating system, the prototype's gateways and application logic both run in the Flume IFC system [46] which provides IFC within the Linux operating system. Running the application logic in an IFC OS has the advantage that untrusted code can safely run both in the client and in the server in a unified IFC space.

At a lower level, each gateway is a long-running Python FastCGI process. The gateway serves static files directly off the file system and queries application request handlers, which are Flume-confined FastCGI processes, to serve dynamic HTTP requests. The gateway is 4144 lines of Python including comments.

3.6.4 Server Storage

As described in Section 3.4.4, a BFlow server can allow untrusted scripts to store data on the server as long as the server associates a label with the data when writing and reading. The BFlow server prototype implements a key-value storage system within its IFC environment. Untrusted browser scripts can read and write data to server storage using AJAX HTTP requests.

When an AJAX request stores data on the server, the storage system labels the data with the label of the request. Later, when an HTTP request reads that data, the storage system only reads data whose label is a subset of the HTTP response's

label. The underlying storage system is an IFC database wrapper built on top of PostgreSQL that resembles the SeaView [52] data model.

Although the prototype storage system runs in an IFC operating system, it is not necessary to use one. In many cases, it should be sufficient for the server to store a label alongside the data and apply the label when reading the data. Together the IFC database wrapper and the HTTP storage request handler are 3288 lines of Python including comments.

3.7 Applications

To demonstrate that BFlow preserves privacy and is flexible enough to build Web platforms, we implemented two Web applications within the BFlow framework and a collection of untrusted JavaScript extensions.

3.7.1 BF-Blogger

Blogger [12] is a popular blog hosting service that supports confidential blogs that only specific users can read. Blogger allows a blog's author to install third-party JavaScript extensions that run in the browsers of all viewers of the blog. These extensions can use confidential data, such as recent posts in the current blog. Other extensions talk to external Web servers: for example, one extension displays random images from a photo-sharing Web site. All JavaScript runs in the same browser frame with access to the blog's confidential data, including the blog posts and the reader's browser cookies making it possible for malicious scripts to leak the data.

BF-Blogger is derived from Blogger's HTML, JavaScript, and third-party extensions, but it runs in BFlow. In a BF-Blogger blog, the top-level trusted zone contains one child and protection zone for the main blog content (including Blogger's JavaScript) and a separate child and zone for each extension. BF-Blogger associates the data from a confidential blog with tag t.

The main blog content's zone contains the blog's confidential content, so it starts with the label $L = \{t\}$. Each extension zone starts with an empty label $L = \{\}$. An extension can make an HTTP request to the server to read confidential blog contents, thus changing its label to $L = \{t\}$.

We ported seven Blogger extensions to BF-Blogger. The *Twitter* and *Flickr* extensions fetch data from external Web servers; they do not read the confidential blog contents, so BFlow permits them to fetch the external data. The *Recent Posts* extension fetches the current blog's contents, computes a set of post snippets, and displays them to the user. The *Cbox* extension implements a multi-user chat room. Cbox consists of multiple cooperating frames, each with its own JavaScript and the individual frames read and write the other frame's DOM. BF-Blogger runs Cbox as if it had read confidential data ($L = \{t\}$) because it stores data on the server, and users might chat about the confidential blog contents. Cbox consists of multiple frames, but since BF-Blogger groups them into a single protection zone, BF-Blogger can set the zone label just once. This changes the label for all of Cbox's frames without

BF-Blogger being aware of all of Cbox's sub-frames. Because the chat contents might be confidential, we modified Cbox to store its data in BFLOW server storage with label $L = \{t\}$. We also wrote two *Evil* extensions that run in both Blogger and BF-Blogger; their goal is to leak data from a confidential blog (see Section 3.8.1).

Extension developers for BF-Blogger need not understand the details of BFLOW other than that they may not make external HTTP requests after reading confidential data.

3.7.2 BF-Socialnet

BF-Socialnet is a multi-user social network that uses BFLOW to protect privacy. Each user has a profile and a set of friends. BF-Socialnet permits JavaScript extensions to run within its pages with access to the user's profile and friend list. We implemented two JavaScript extensions, a *profile comparison* tool and a *messaging* tool to exercise BFLOW's support for different communication patterns and privacy policies.

BF-Socialnet's base *friend* privacy policy is that user Alice's profile and friend list is only visible to Alice's friends. In addition, BF-Socialnet supports *personal* data which only Alice may read and *pairwise* data that a particular pair of users may read. To implement these policies, BF-Socialnet uses a set of tags for each user, one tag for personal data that only Alice can see (t_{alice}), one tag for the Alice's friend-visible data ($t_{alice:friends}$), and one tag for each of Alice's friends for pairwise-visible data; for example if Alice is friends with Bob, BF-Socialnet would use the tag $t_{alice:bob}$.

The BF-Socialnet page has a trusted root page that contains different sub-frames for each third-party extension. The root page has multiple frames for each extension, each with a different confidentiality mode. For example, in one frame, the messaging extension runs in a mode that allows it to read all data that the user can read. In a separate frame, the messaging extension runs with a pairwise tag determined by the root page. The user selects who to send a message to using a drop down box in the root frame, and the root frame adjusts the label on the frame accordingly. The profile comparison tool only reads data, and therefore only runs in a mode that allows it to read all data that the user can read. It uses AJAX requests to read the profiles of all the user's friends, compares them in the browser, and outputs a list of friends with similar interests.

User and Developer Visible Model: In BF-Socialnet, an application writer needs to know what confidentiality mode his application will run under and what data it hopes to read. However, he does not need to understand labels, tags, or the information flow model. Similarly, users should be able to understand that the different sub-frames abide by different confidentiality modes because data that they input to a sub-frame will abide by the frames confidentiality mode. This decision is similar to the decision that users make currently when choosing their profile's privacy policy, so we expect users will be able to understand it.

Implementation

BF-Socialnet is implemented as 283 lines of Python and 124 lines of HTML using the Django Web framework [20]. BF-Socialnet runs as a Flume confined process and saves data on the server in the IFC database wrapper described in Section 3.6.3. The profile comparison tool and the messaging tool are, respectively, 104 and 103 line Django applications.

3.7.3 W5

W5 is a Web server platform in which any third-party programmer can write an application and deploy it to a utility-like W5 server. W5 serves as an example application made possible by combining BFlow with an operating system level IFC system. The unique property of W5 is that any application running on a W5 server can read data that other applications store on the same server even if that data is confidential to the user (such as a user's confidential photos). However, the W5 server prevents applications from leaking that confidential user data to unauthorized recipients, even to the author of the application. W5 accomplishes this by running applications in an IFC environment on the server, running the application's HTML and JavaScript in BFlow, and integrating the two IFC systems in a single information flow realm.

W5 needs to use BFlow in the browser because the third-party code running on the server can generate arbitrary JavaScript. Without BFlow, that JavaScript could leak confidential data from the server to an adversary by sending HTTP requests that contain confidential data to an adversary's Web server.

Reusing and repurposing of data is an advantage to applications that want access to existing data from other applications because an application writer need not persuade users to enter or upload their data into his application if it already exists in another W5 application. Users benefit because they can try new applications without the overhead of reinserting their data, and users can use applications without worrying that the application will steal their data.

The main challenge in building W5 is to support different types of applications and different ways to share data between applications. Using a high-level IFC policy helps to address this challenge because the applications can share data with each other as long as the high-level confidentiality policy is upheld. This section demonstrates how to combine BFlow and Flume to construct W5.

High-level Architecture

Figure 3-5 shows the high-level W5 architecture. W5 involves three main entities: *providers*, *developers*, and *end-users*.

The W5 provider supplies a server[3], database, and file system, and runs the W5 framework to control how applications use these resources. To enforce the framework's security policies, the provider supplies "gateway" processes that reside on the server

[3] For clarity, we use the term *server*, but W5 could generalize to a cluster of servers.

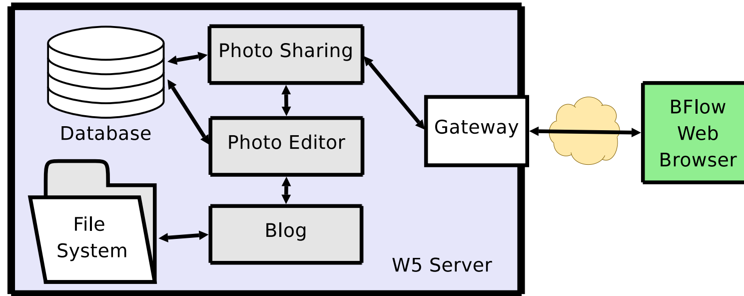

Figure 3-5: W5 overview showing three applications.

and govern all communication between the W5 server applications and the end-user browsers.

Developers deploy application software on the W5 server. They can upload binaries, libraries, and scripts to W5, and assemble them into Web applications. W5 gives developers wide latitude in how to engineer their applications, allowing use of third-party libraries and plugins along with most of the facilities of the underlying operating system (e.g. Linux). Applications can read and write shared data, and applications can exchange data with external Web servers. The intent is that anyone can be a W5 developer.

End-users interact with W5 sites through Web browsers. When establishing an account, logging on, or configuring her security preferences, the user interacts with the W5 gateway via an application start page. Otherwise, developer-written code handles her data and requests via the gateway.

W5 can enforce open-ended user- or provider-specified policies. The default policy lets a user mark data as one of three levels of privacy: private, friends-only, and public. Any application may read a user's private data but may only cause it to be revealed to that user's browser. An application may reveal a user's friends-only data to the browser of someone on the user's friends list. An application may reveal public data to anyone.

Implementation

The W5 server prototype runs on the Flume operating system [46], which provides DIFC extensions to a standard Linux operating systems. Third-party applications run as sandboxed Flume processes. As such, they can access the core Linux API (e.g., `fork`, file I/O, pipes, etc) but cannot access those that would allow data leaks or privilege escalation (e.g., `ioctl`, `ptrace`, `bind` in certain circumstances, etc). When Flume does allow API calls, it tracks user information as it flows between processes, files and the database. W5 uses Flume's DIFC file system and BFlow's database for persistent labels.

The W5 gateway also runs as a process in user space, but owns many sensitive privileges (such as the ability to export user data) and must therefore run outside a strict sandbox. In this sense, the gateway sits at the security perimeter of the server:

third-party applications must go through the gateway to communicate with clients or other outside network hosts.

At a lower level, the gateway is a long-running Python FastCGI process. The gateway serves static files directly off the file system and forks a new Flume-confined CGI process to serve each dynamic HTTP request. W5 currently supports applications written in Python as well as binary executables. The gateway is 4641 lines of Python and the W5 database is a 3400 line Python wrapper around a PostgreSQL database.

W5 **Applications**

To demonstrate the feasibility of the W5 platform, we built a number of extensible applications that exercise different parts of W5. All of these applications are written in Python and use the Django framework, which explains why they require only a small amount of source code.

Calendars: W5 has two separate calendar applications, Calendar and WeatherCalendar. Together, they demonstrate how two mutually distrustful applications can work together, and how an application can communicate outside the W5 server within the W5 security policy. The Calendar is a standard calendar program that stores event dates and times. WeatherCalendar is similar to the Calendar and reads Calendar's database entries, but also periodically fetches data from an online weather database [85] and displays the weather alongside the calendar data. The Calendar application is 206 lines of code, and the WeatherCalendar is 321.

W5 allows WeatherCalendar to read Calendar's database entries and use them in its event listing, but W5 ensures that WeatherCalendar cannot leak the user's confidential Calendar entries, despite communicating with the remote weather database, even if the WeatherCalendar is malicious.

Blog: W5 has a blog application that demonstrates W5's support for sharing data between users, read access control and search functionality. A blog author can configure each individual blog and blog post to be publicly readable, or to have restricted read access using different levels of privacy as described above.

The Blog also supports searching through the blog posts by keyword. It might seem that an application in W5 could not support an application that searches through all the posts because they all have different secrecy labels. A process that reads each post and checks for matches would likely read a post that the user is not allowed to read and thus become unable to send its results back to the client. To implement keyword searching, W5's database creates a view for the user that contains only data that the user has permission to read. This ensures that the keyword matching query will only return the posts that are both readable and match the search query. The Blog application is 268 lines of code.

W5 enforces the blog's read access controls, so users need not trust that the blog application implements the correct access control checks; users only need to trust that W5's gateway and IFC system are implemented correctly. Users configure their read

access controls through an interface to the W5 gateway, and the W5 enforces it on the blog application.

Photo Sharing and Editor: The W5 photo sharing application illustrates data sharing between applications. Users create albums, upload photos, and view albums. Like the blog, users can also view other users' albums, if the album's privacy policy allows. The photo sharing application is 451 lines of code.

The W5 photo editor works with the photo sharing application to show how separate applications can share code and writable data on the server. The photo sharing application and the photo editing application are written by different developers, yet the photo editor can edit the photos in the photo sharing application. To implement sharing, the photo editor imports software modules related to the data format from the photo sharing application. The photo editor also uses an open-source imaging library, including C extensions to read and modify images. The photo editor is 119 lines of code, and the C extension is 45,258 lines of code.

As in the calendar and blog applications, W5 permits data sharing between users and applications. W5 also prevents the photo editor from leaking the users' confidential photos, whether by accident or maliciously.

3.8 Evaluation

This section evaluates how well BFlow achieves its two main goals: prevention of confidential data leaks from in-browser JavaScript, and compatibility with existing developer uses of JavaScript. We focus on these topics rather than performance because the performance penalty of the browser extension should be minimal and the HTTP proxy can be eliminated by moving its functionality into the Web server.

3.8.1 Security

Attack Analysis

This section explains how BFlow prevents the example attacks described in Section 3.3.1, Figures 3-1 and 3-2.

In Figure 3-1a, malicious JavaScript resides in the same frame (and thus the same zone) as the confidential data. BFlow ensures the a zone's label includes tag t before it allows the zone to read confidential data with tag t, therefore the malicious script will be running in a zone with tag t. This label constrains the malicious script so that it can display data only to the browser's human reader and the source Web server. The former is not a leak, since the source server would not have sent the data unless the browser's user had permission to read it. The latter is not a leak because BFlow propagates tag t along with the data, so that the source server will know it is confidential.

In Figure 3-1b, the confidential data (and benign JavaScript) is not in the same zone as the malicious JavaScript. If the benign JavaScript accidentally tries to communicate with the malicious JavaScript, the BFlow reference monitor will forbid the

communication unless the malicious JavaScript's zone's label is a superset of the label of the zone with the confidential data. In the latter case the malicious JavaScript will be restricted from leaking as described in the previous example.

Attack Examples in Blogger

In order to verify that BFlow fixes existing security problems, we implemented two JavaScript extensions for Blogger that steal confidential information.

The first extension contains a cross-site scripting (XSS) attack that exploits a typical script injection vulnerability. We wrote this attack, but we believe that XSS attacks in the wild would use the same leak technique since today's Web sites do not usually use any counter measures. In this attack, the adversary tricks user A into placing the extension on his blog so that viewers of his blog execute the extension's script. When some user B views A's blog, the extension reads user A's confidential blog contents and user B's Blogger cookie and sends it to an external server using an image request, thus leaking A and B's confidential data. This attack works when run on the real Blogger Web site, but the extension is unable to leak data when run on BF-Blogger, since BFlow forbids the extension from contacting the external server because its zone has seen confidential data.

The second attack is meant to approximate the one pictured in Figure 3-1b. We believe this is a new style of attack and are unaware of such attacks in the wild because intra-browser JavaScript APIs are currently uncommon. The attack consists of two parts: the *listener* and the *leaker*. The leaker takes the place of a vulnerable script API and the listener takes the place of an adversary that tricks the vulnerable script into reading confidential data and sending it to the listener. In this attack the listener script resides in a frame in the adversary's origin, and listens for a message from the leaker. The leaker runs in the same origin as the confidential Blogger page, and sends confidential data to the listener using `postMessage`. Again, this attack works when run on the real Blogger Web site, but the leaker is unable to send data to the listener with `postMessageBF` in BF-Blogger, because BFlow forbids the leaker (who has seen confidential data) from messaging the listener (who has an empty label $L = \{\}$).

3.8.2 Adoption

In order to evaluate the complexity of developer adoption, we ported several existing Blogger widgets [10, 74, 29] to BF-Blogger. They fall into three categories:

- Those that load data, images, or libraries from external servers, or link to external servers.
- Those that read the blog's confidential content using the blog's JSON feed.
- Those that do both of the above.

Extensions in the first category, such as the *Flickr*, *Twitter*, and *Buzz* extensions required no changes to work on BF-Blogger. These extensions need no confidential

Extension	LOC	LOC Included	LOC Changed	Confidential Data?
Twitter	6	19	0	No
Flickr	10	0	0	No
Buzz	1	0	0	No
Blogger JS	60	851	0	No
Youtube	1282	610	0	No
Calendar	804	1141	0	No
Weather	2993	797	0	No
Popular Posts	16	0	1	Yes
Commenters	15	0	1	Yes
Recent Posts	9	65	2	Yes
Random Post	34	0	2	Yes
CBox	801	0	89	Yes

Table 3.2: Lines of code (LOC) changed to port existing widgets to BF-Blogger and whether they see confidential data.

data, so they can be loaded in frames that have an empty label, and are free to fetch data from external servers.

The *Recent Posts* extension is in the second category. It fetches the blog's most recent posts and displays a list of them on the blog's side bar. The original version loads a JavaScript file from an external site, which fails because the script reads the blog content before making the external HTTP request for the JavaScript file. To make this extension work in BF-Blogger, we copied the content of the external JavaScript file into the extension.

The two extensions we found in the third category, namely *Popular Posts* and *Top Commenters* are a form of mashup. They use an external server (Yahoo Pipes [84]) to process the content of the blog's confidential comments and then display the results in the page. They illustrate how a mashup sometimes trusts an external server with confidential data. To add support for these in BF-Blogger we added a comment feed to the blog and made the feed available to only the Yahoo Pipes client host. This feed policy is an explicit declassification of the confidential comments to the Yahoo Pipes host.

We also examined a number of Google Gadgets [32]. The twenty most popular Google Gadgets don't act on confidential data, and just import data from external sites or from Google's platform. We ported the generated JavaScript of three Google Gadgets to run on our platform: *Youtube Search*, *Google Calendar*, and *Current Weather*. All worked without changes.

The *Cbox* messaging system required more code changes since it stores persistent data to the server; it was modified to read included files from our platform and to store messages using our server storage API.

3.9 Deployment

When considering deployment and adoption, it is clear that BFlow faces more hurdles than Resin because both the Web site developers and the end users must adopt BFlow before it can be useful. However, adding support for BFlow in a Web site does not make that Web site incompatible with today's browsers; BFlow is backwards compatible with browsers. A Web site can adopt BFlow even if none of its clients buy-in.

To deploy BFlow, a Web site would add support internally, and then enable BFlow features like third-party JavaScript only for browser clients that support BFlow. This way, the site will still work for non-BFlow clients, but the users may be enticed to install the BFlow's browser extension to use the Web site features. As a concrete example, Google could integrate BFlow's reference monitor into the Google toolbar, and a few popular applications like Gmail. Then users can adopt BFlow incrementally without requiring the installation of a new browser, or requiring all users to install BFlow at once.

3.10 Limitations and Future Work

BFlow currently has a number of limitations which we plan to address in future work.

3.10.1 Information Flow Control

Cross-Zone Communication

The current BFlow prototype isolates zones using the browser's same-origin policy. This means that two different zones cannot read and write each other's DOM variables and cookies, but two zones should be able to read and write cookies and DOM variables from other zones as long as their labels would allow.

It should be possible to add cross-zone DOM variable access through the Firefox extension interface. One approach is to add a function call interface similar to `postMessageBF` for variable access, but it may also be possible to provide direct language integration to avoid changing the JavaScript API. BFlow could use the same techniques to provide access to cookies in different zones.

Distributed IFC in BFlow

As designed, BFlow only tracks information on a per-Web site basis. For example, a single protection zone cannot contain tags from two different Web sites, and transferring confidential data from one site to another site in a mashup will remove the tag information from the data. Currently BFlow does not allow this because the prototype preserves JavaScript's ability to send messages to the top-level frame through the FID channel.

However, it should be possible to extend BFlow so that zones can read confidential data from different sites and compute on it, as long as it does not then send data to a server. This would require a way to close the FID channel after the JavaScript adds a cross-site tag to its label and more lenient rules for sending requests to external servers.

Granularity

Currently, BFlow limits the way programmers can design their Web pages due to BFlow's coarse grained IFC. Since BFlow only tracks data at the granularity of frames, a single untrusted browser frame cannot simultaneously handle confidential data and public data without marking the public data as confidential. In order to protect the confidential data, a BFlow application would label the frame with $L = \{t\}$, but then the public data would also be labelled with $L = \{t\}$ and be unavailable to the public. This is a scenario where finer grained information tracking [57] would help. Site developers might also have to refactor their HTML to partition data into frames to separate confidential data with different tags.

Browser Plugins

Another limitation of BFlow is that it does not apply to browser plugins. For example, BFlow does not support Flash [1] or Java [36] plugins. It may be possible to integrate BFlow-like IFC to plugin like these.

3.10.2 User Interface and Understanding Labels

Given that BFlow uses frames to do privilege separation, users might be confused that frames have different security labels and type sensitive data into frames with $L = \{\}$ which would leak the data. Web sites can help by marking frames, but BFlow does not currently provide a solution for this. Future versions of BFlow could use different user interface annotations to mark frames with label information.

When designing a label based confidentiality scheme, reasoning about labels is not always straightforward and errors in designing a scheme can result in data leaks. BFlow does not provide assistance for using labels, but other projects have made progress in this area [21].

3.10.3 Applications

Applications for Third-Party JavaScript

As we explain in Section 3.1, most of today's Web sites do not support third-party JavaScript for security reasons. However, given BFlow, more Web sites could safely take advantage of third-party JavaScript including widget-like extensions. Web-based email, calendar, and finance systems could support extensions such as encryption, page formatting, and layout customization. Many of the popular Greasemonkey [49] extensions could also work in a BFlow environment.

In-Browser JavaScript APIs

The example applications given in Section 3.7 do not use cross-zone messaging because those applications communicate through the server. In the future, applications may use more intra-browser messaging, as evidenced by the new JavaScript API libraries [75] that aim to ease cross-domain messaging. BFlow can provide better assurances to programmers who want to limit their exposure to such in-browser APIs; programmers would label their data differently depending on whether they want to expose it to the API or not. For example, a banking application might be willing to send information about one month's payments, but not the account balance, to a widget that graphs one month's expenses.

3.10.4 Out of Scope Attacks

There are a number of attacks for which BFlow does not offer a solution; the following challenges are left open for future work. If a malicious script with label $L = \{t\}$ uses a covert channel [48] like CPU modulation to send data to a script with label $L = \{\}$, it can leak the confidential data. If a malicious script uses a phishing attack to trick a user into revealing his password the attacker can subsequently login as the user and read all his confidential data.

As described in Section 3.3.1, BFlow does not protect against a compromise in the servers, browsers, operating systems, or the BFlow software itself. For example, if an attacker can trick a user into installing his malicious Firefox extension, he could disable BFlow. Similarly, Web sites with weak user authentication are vulnerable in ways that BFlow does not fix.

If an attacker is able to cause a trusted zone in BFlow to load and run his malicious code, then the script will act with the privileges of the trusted zone and will be permitted to leak confidential data. However, trusted zones are intended to be very carefully validated and to never run third-party code; BFlow protects data in all non-trusted zones from leaks.

3.10.5 Design Variations

Intermediate Designs

Although BFlow is meant to be backwards compatible, Web site developers might be reluctant to use all of BFlow's elements, or browser developers might be reluctant to implement the necessary browser changes to support BFlow. One direction for future work is to take the BFlow's goals and try to implement them using fewer changes to the browser, server, or both. For example, it might be possible to get most of BFlow's benefits by running untrusted JavaScript within a browser sandbox, running with limited outgoing communication channels.

Beyond Backwards Compatibility

BFlow incorporates a number of design choices that preserve backwards compatibility with existing JavaScript and browsers; redesigning BFlow without regard for backwards compatibility would likely result in a different design. For example, to use BFlow, a Web site developer needs to partition JavaScript into different frames and protection zones depending on whether the Web site developers trust the JavaScript; developers also need to communicate between these zones using message passing. Using zones as the IFC granularity is an advantage when reusing existing browsers because browsers already provide some isolation between frames. However, if there were no legacy browsers, it might be more convenient if Web site developers did not need to partition JavaScript. Instead, BFlow could use finer-grained, language-level IFC as in systems like Jif [57] or Resin, and communication between trusted and untrusted JavaScript could use shared variables and function calls rather than message passing. Future research is necessary for a clean-slate design for untrusted third-party browser scripts, although projects like Caja [55] do show promise.

3.11 Related Work

One way to understand existing work is in two broad categories: discretionary access control (DAC) (including capabilities-based systems and least-privilege isolation techniques) and mandatory access control (MAC) (including language-based and runtime IFC).

3.11.1 Discretionary Access Control

Works like Tahoma [68], Google Chrome [33] and MashupOS [80], Caja [55], and Bitfrost [63] all fit the DAC model.

Tahoma isolates applications from each other using virtual machines so that even buggy browsers running malicious code cannot tamper with cookies or DOM objects in other browsers. Users can choose to share data across Web sites with explicit whitelists of all other hosts that can be contacted as the page is rendered and as the JavaScript (or other plugins) run. Thus, Tahoma offers all-or-nothing sharing at the discretion of the original Web site; it does not allow a Web site to safely give confidential data to potentially malicious scripts. The Chrome browser implements the same style of isolation between browser windows, but with process-based rather than VM based isolation.

MashupOS proposes changes to Web browsers and servers to isolate third-party JavaScript code with more flexibility than today's browser frames and finer granularity than inlining scripts today. MashupOS proposes HTML extensions such as `<Sandbox>` and `<OpenSandbox>`, which occupy a middle ground: they allow the caller and callee to communicate but only along well-understood channels (as opposed to across the whole DOM under the status quo). However, MashupOS has the same limitations that DAC-based operating systems have: the user (or the *integrator* in MashupOS's terminology) must still decide *a priori* whether to trust a third-party

or not with sensitive data because sandboxed scripts in MashupOS can leak data to external servers. In BFlow, untrusted scripts can decide whether to read private data at runtime.

Other works like Caja follow MashupOS's lead. Caja confines a subset of JavaScript into an object-capability model. As in MashupOS, the goal is to allow finer-grained sharing of data between cooperating browser components. Like Caja, Bitfrost allows an application writer to confine her own application so that they can only access certain operating system services. For example, the author of a single-user card game would configure the game to voluntarily, and irreversibly give up access to the network and local storage, at install-time. This way, even if the card game is compromised, it cannot read confidential data from local storage and send it over the network. BFlow differs because it does not require application writers to make this choice at install-time, instead the application can decide at runtime whether it needs these resources or not.

3.11.2 Mandatory Access Control

By contrast, MAC systems allow untrusted software to compute with confidential data, while preventing that software from exposing it. MAC has long been a technique at play in programming languages [18] and operating systems [11, 54, 19], which modern research [22, 89, 46, 57] suggests is practical for server-side Web applications. The same tools apply in the context of browser-based security.

The SIF system [15] uses language-based information flow control to maintain privacy constraints between browser and server, but assumes no malicious or buggy JavaScript. The Swift system [14] uses IFC to automatically split Web applications into trusted server-side Java and untrusted browser-side JavaScript. BFlow applies similar information control analysis, but at runtime. BFlow retains a similar correctness property, that code will produce a fail-stop error instead of leaking data. While Swift only applies to JavaScript output by the Swift compiler, BFlow's reference monitor applies to all JavaScript code, such as legacy and hand-written libraries. However, BFlow does make trade-offs; firstly, it has coarser-grained security compartments (browser zones) while Swift tracks information flow per variable. Secondly, BFlow requires users to install a browser plugin and Swift-like system would not. Using a browser plugin enables BFlow to ease the adoption burden placed on site developers at the expense of the end users.

Vogt et al. [78] also track information flow control at runtime to prevent cross-site scripting attacks. However, they have limited their system to client-side changes only, and therefore cannot prevent attacks that move data back and forth between the browser and server. Spectator [50] tracks taint between browsers and servers, but its goal is to detect JavaScript worms, not protect privacy.

Other work proposes curtailing JavaScript's power to solve traditional XSS problems. BrowserShield [67] rewrites arbitrary (potentially malicious) JavaScript to a safer core. BEEP [41] firewalls unsafe JavaScript by limiting which servers it can contact as it executes. Hallaraker et al. [39] audit JavaScript execution, and use intrusion-detection techniques to sense anomalous execution patterns. These veins

of work show promise against traditional XSS attacks but do not handle data leaks which involve sending data back and forth to the origin server.

A complementary way to build Web extensions is on the server-side, rather than on the browser. Facebook [26] and OpenSocial [35] give third-party developers access to server-based data, allowing them to customize and extend existing server-based features. The Menagerie [31] system presents an interface to make server data more accessible. All of these systems use discretionary security controls, requiring users to either trust or reject third-party code. W5 [47] proposes to achieve similar features with MAC, but a W5 implementation would need to solve the security challenges discussed in Section 3.3 to allow third-party server-side extensions to push unvetted JavaScript to browsers.

3.12 Summary

Many of today's Web sites currently use JavaScript that they might not understand, including large libraries and third-party extensions. The combination of these possibly buggy or malicious scripts and confidential data leaves that data open to attack. BFlow is a novel browser based information flow control system that allows mostly unmodified legacy JavaScript to read, compute with, and write confidential data without the risk of compromising user privacy.

Chapter 4

Integrating Resin and BFlow

BFlow and Resin are independent systems, but a Web site can use them together to its advantage. As argued in Section 2.2, it is difficult for programmers to understand all the data flow paths within a complex application, yet BFlow relies on the Web site programmer to propagate IFC labels from an HTTP request to any response that is derived from that request (see Section 3.4.4).

As a solution, the Web site programmer can use Resin to attach a policy object to data in an HTTP request, for each tag in the request's label. Resin will propagate the policy to variables and persistent storage. Finally, when the application prepares to send an HTTP response, it can check for policy objects on the data in the response and then attach a tag for each policy object in the response's data.

Chapter 5

Conclusion

Building secure Web sites today is difficult and error prone, despite the growing maturity of Web technology. Web server software continues to exhibit security vulnerabilities such as cross-site scripting, SQL injection, HTTP response splitting, data leakage, and forgotten authorization checks. Web sites use increasing amounts of JavaScript, much of which they do not write. In some cases, Web sites sacrifice data confidentiality in order to support third-party JavaScript.

At a high level, these vulnerabilities are due to data flowing where it should not, and this work shows that by tracking data flows, it is possible to prevent these faulty data flows, and the vulnerabilities they cause.

5.1 RESIN

RESIN provides programmers with tools to convert an implicit data flow plan into an explicit data flow assertion, and then have RESIN check that assertion on all data flow paths, even where the programmer may have forgotten. The assertions allow a programmer to reason about the security of the system as a whole and enforce a high-level security plan without having to worry about every possible data flow path in the bulk of the system.

The contributions of RESIN are the idea of a data flow assertion; a method for implementing data flow assertions using filter objects, policy objects, and data tracking; and finally, an evaluation showing that data flow assertions are concise, effective, and incrementally deployable.

5.2 BFLOW

BFLOW is a system that makes it possible for Web sites to incorporate untrusted JavaScript and allow the JavaScript to compute with confidential data without the risk of leaking that data. To accomplish this, BFLOW adds information flow control to the browser, and to the browser-server interactions using an in-browser reference monitor and small changes to the server. Using information flow control, BFLOW determines whether untrusted JavaScript may have seen confidential data, and if so,

BFlow prevents the JavaScript from leaking that data to users who lack permission to read it.

The contributions of BFlow are a set of information flow control rules that govern the JavaScript communication mechanisms, a mapping from BFlow's IFC rules to the browser's existing JavaScript isolation system, and an abstraction called a protection zone that eases the deployment of existing JavaScript into BFlow. Together, these techniques allow untrusted JavaScript to read, compute with, and display confidential data without the risk of leaking that data.

5.3 Summary

This dissertation presents Resin and BFlow, two systems that can improve the state of Web security today through data tracking. We hope that programmers will adopt this work, extend it to suit their needs, and find new applications for the technology.

Bibliography

[1] Adobe. Flash. http://www.adobe.com/products/flash, January 2009.

[2] Gail Ahn, Xinwen Zhang, and Wenjuan Xu. Systematic policy analysis for high-assurance services in SELinux. In *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, Palisades, NY, June 2008.

[3] Anne H. Anderson. An introduction to the Web services policy language (WSPL). In *Proceedings of the 2004 IEEE Workshop on Policies for Distributed Systems and Networks*, Yorktown Heights, NY, June 2004.

[4] Jeremy Bae. Vulnerability of uploading files with multiple extensions in phpBB attachment mod. http://seclists.org/fulldisclosure/2004/Dec/0347.html. CVE-2004-1404.

[5] Steve Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, Stresa, Italy, June 2009.

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects*, Amsterdam, The Netherlands, November 2005.

[7] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, Marseille, France, March 2004.

[8] Adam Barth, Collin Jackson, and John C. Mitchell. Securing browser frame communication. In *Proceedings of the 17th USENIX Security Symposium*, pages 17–30, San Jose, CA, USA, July 2008.

[9] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, Chicago, IL, June 2005.

[10] Beautifulbeta. Blogger widgets. http://beautifulbeta.blogspot.com, January 2009.

[11] David E. Bell and Leonard La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, USA, March 1976.

[12] Blogger.com. Site. `http://www.blogger.com`, January 2009.

[13] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Computer and Communications Security Conference (CCS)*, pages 39–50, Alexandria, VA, October 2008.

[14] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, Stevenson, WA, October 2007.

[15] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in Web applications. In *Proceedings of the 16th USENIX Security Symposium*, pages 1–16, Boston, MA, August 2007.

[16] CWH Underground. Kwalbum arbitrary file upload vulnerabilities. `http://www.milw0rm.com/exploits/6664`. CVE-2008-5677.

[17] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Proceedings of the POLICY 2001 Workshop*, pages 18–38, Bristol, UK, January 2001.

[18] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[19] Department of Defense. *Trusted Computer System Evaluation Criteria (Orange Book)*, dod 5200.28-std edition, December 1985.

[20] Django Software Foundation. Django. `http://www.djangoproject.com`, May 2009.

[21] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 301–313, Glasgow, Scotland, March 2008.

[22] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, Brighton, UK, October 2005.

[23] Emory University. Multiple vulnerabilities in AWStats Totals. `http://userwww.service.emory.edu/~ekenda2/EMORY-2008-01.txt`. CVE-2008-3922.

[24] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, October 2000.

[25] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[26] Facebook. Site. `http://www.facebook.com`, January 2009.

[27] David F. Ferraiolo and D. Richard Kuhn. Role based access control. In *Proceedings of the 15th National Computer Security Conference*, October 1992.

[28] Firefox. Add-ons. `https://addons.mozilla.org/`, January 2009.

[29] Flickr. Badge. `http://www.flickr.com/badge.gne`, January 2009.

[30] Scott Garriss, Lujo Bauer, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, Estes Park, CO, June 2008.

[31] Roxana Geambasu, Cherie Cheung, Alexander Moshchuk, Steven D. Gribble, and Henry M. Levy. Organizing and sharing distributed personal Web service data with menagerie. In *Proceedings of the 17th International World Wide Web Conference*, pages 755–764, Beijing, China, April 2008.

[32] Google. Gadgets. `http://www.google.com/webmasters/gadgets/`, January 2009.

[33] Google. Google chrome: a new web browser for windows. `http://www.google.com/chrome`, January 2009.

[34] Google. Maps API. `http://code.google.com/apis/maps`, January 2009.

[35] Google. Open Social. `http://code.google.com/apis/opensocial`, January 2009.

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[37] William G. J. Halfond and Alessandro Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th ACM International Conference on Automated Software Engineering*, pages 174–183, Long Beach, CA, November 2005.

[38] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 2006 ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–185, Portland, OR, November 2006.

[39] Oystein Hallaraker and Giovanni Vigna. Detecting malicious JavaScript code in Mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, Shanghai, China, June 2005.

[40] Norman Hippert. phpMyAdmin code execution vulnerability. `http://fd.the-wildcat.de/pma_e36a091q11.php`. CVE-2008-4096.

[41] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International World Wide Web Conference*, pages 601–610, Banff, Alberta, Canada, May 2007.

[42] Shinya Kasatani. Safe ERB plugin. `http://agilewebdevelopment.com/plugins/safe_erb`, January 2009.

[43] Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX track*, pages 273–284, San Antonio, TX, June 2003.

[44] Eddie Kohler. Hot crap! In *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems*, San Francisco, CA, April 2008.

[45] Maxwell Krohn. Building secure high-performance Web services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.

[46] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, Stevenson, WA, October 2007.

[47] Maxwell Krohn, Alexander Yip, Micah Brodsky, Robert Morris, and Michael Walfish. A World Wide Web without walls. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networks*, Atlanta, GA, USA, November 2007.

[48] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[49] Anthony Lieuallen, Aaron Boodman, and Johan Sundstrm. Greasemonkey. `https://addons.mozilla.org/en-US/firefox/addon/748`, June 2009.

[50] Benjamin Livshits and Weidong Cui. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 335–348, Boston, MA, USA, June 2008.

[51] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Baltimore, MD, August 2005.

[52] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The seaview security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, 1990.

[53] Michael Martin, Benjamin Livshits, and Monica Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 2005 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383, San Diego, CA, October 2005.

[54] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, 1992.

[55] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript, 2008. `http://code.google.com/p/google-caja/downloads/list`.

[56] MoinMoin. The MoinMoin wiki engine. `http://moinmoin.wikiwikiweb.de/`, May 2009.

[57] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, Saint-Malo, France, October 1997.

[58] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems (TOCS)*, 9(4):410–442, October 2000.

[59] Myphpscripts. Login session script. `http://www.myphpscripts.net/?sid=7`, May 2009. CVE-2008-5855.

[60] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 295–307, Chiba, Japan, May 2005.

[61] Osirys. wPortfolio arbitrary file upload exploit. `http://www.milw0rm.com/exploits/7165`. CVE-2008-5220.

[62] Osirys. myPHPscripts login session password disclosure. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-5855`, May 2009.

[63] One Laptop per Child. Bitfrost. `http://wiki.laptop.org/go/OLPC_Bitfrost`, June 2009.

[64] Perldoc. Perl taint mode. http://perldoc.perl.org/perlsec.html, May 2009.

[65] phpMyAdmin. phpMyAdmin 3.1.0. http://www.phpmyadmin.net/.

[66] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, pages 124–145, Seattle, WA, September 2005.

[67] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 61–74, Seattle, WA, November 2006.

[68] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural principles for safe Web programs. In *Proceedings of the 6th ACM Workshop on Hot Topics in Networks*, Atlanta, GA, USA, November 2007.

[69] script.aculo.us. Library. http://script.aculo.us, January 2009.

[70] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 369–383, Oakland, CA, May 2008.

[71] The MITRE Corporation. Common vulnerabilities and exposures (CVE) database. http://cve.mitre.org/data/downloads/, May 2009.

[72] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.

[73] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 160–169, Lake Tahoe, CA, June 2006.

[74] Twitter. Badge. http://twitter.com/badges/blogger, January 2009.

[75] Malte Ubl. Xssinterface: JavaScript library for secure cross browser JavaScript messaging. http://code.google.com/p/xssinterface/, January 2009.

[76] Wietse Venema. Taint support for PHP. http://wiki.php.net/rfc/taint, May 2009.

[77] John Viega, J T Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, February 2001.

[78] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the 14th ISOC Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[79] Thomas Waldmann. Check the ACL of the included page when using the rst parser's include directive. `http://hg.moinmo.in/moin/1.6/rev/35ff7a9b1546`. CVE-2008-6548.

[80] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for Web browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, Stevenson, WA, October 2007.

[81] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, San Diego, CA, June 2007.

[82] Web Application Security Consortium. 2007 Web application security statistics. `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`, May 2009.

[83] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, pages 179–192, Vancouver, BC, Canada, July 2006.

[84] Yahoo. Yahoo! Pipes. `htpp://pipes.yahoo.com`, January 2009.

[85] Yahoo. Yahoo! Weather. `http://developer.yahoo.com/weather`, January 2009.

[86] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 233–246, Nuremberg, Germany, March 2009.

[87] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP) (to appear)*, Big Sky, MT, October 2009.

[88] Aydan Yumerefendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 159–172, Cambridge, MA, Apr 2007.

[89] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, November 2006.

[90] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, San Francisco, CA, April 2008.