Protecting Incorrectly Implemented Web Applications From Online Adversaries

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Eric Yawei Chen

B.S., Electrical and Computer Engineering, University of Toronto
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

July, 2015

Acknowledgements

First, I would like to thank Collin Jackson for introducing me to web security and providing me with a tremendous amount of inspiration, guidance and support throughout my time at Carnegie Mellon University. I owe a great debt of gratitude to my co-advisor Patrick Tague and my mentor Shuo Chen who have helped me immensely on my research and my life. I could not have made it this without their extensive support. I also want to thank my thesis committee, Patrick Tague, Shuo Chen, Collin Jackson, and Anupam Datta for their helpful suggestions and feedbacks. I am deeply grateful to my mentors, Shuo Chen, Helen Wang, Alex Moshchuk, Shaz Qadeer and Rui Wang for expanding my knowledge and enlighten my mind. I would also like to thank my co-authors, Patrick Tague, Shuo Chen, Collin Jackson, Yuan Tian, Yutong Pei, Robert Kotcher, Adam Barth, Charles Reis and Jason Bau for their collaboration on the works included in this thesis. My research is supported by NSF TRUST and Google.

Many friends have brightened my life during my time at Carnegie Mellon University and during my two internships at Microsoft Research. I want to particularly thank Mark Xia, Maxim Siniavine, Avneesh Saluja, Kevin Su, John Vilk, Matt Calder, Yusuke Matsui, Higuchi Keita, Tatsunori Hirai, Yoichi Ochiai, Soramichi Akiyama, and all of the PhD students from Carnegie Mellon University Silicon Valley. Finally, my heartfelt thanks go to my parents for providing me with constant and unconditional support.

Abstract

Implementation errors are commonly found in modern web applications. They can be caused by a multitude of factors, including weaknesses in browsers' security policies and developers' misinterpretations of web protocols (e.g., OAuth and OpenId).

In this thesis, we show that even under the assumption that web applications are implemented incorrectly, their security can be improved through two fronts: 1) Enhancing the application isolation mechanism of web browsers, and 2) securing inter-application communication protocols via program verification. For 1), we created a mechanism called App Isolation to enhance isolation boundaries of web applications running inside a browser. For 2), we created a formal verification framework called Certification of Symbolic Transaction (CST) that verifies the security properties of every web transaction on-the-fly by invoking static verification at runtime. The threat model we consider in this thesis is the standard web attacker with additional capabilities of a malicious user.

The two defenses proposed in this thesis are lightweight and backward compatible. App Isolation can be deployed as an opt-in feature for websites; and we have applied CST to five commercially deployed applications to secure APIs from well-known companies including Facebook, Amazon, PayPal, and Microsoft.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Threat model 2

1.1.1 Web Attacker 3

1.1.2 Malicious User 3

1.1.3 Out of Scope Attacks 3

1.2 Types of implementation errors 4

1.2.1 Errors within an isolated application 4

1.2.2 Errors in multiparty protocol implementations 5

1.3 Proposed defenses 7

1.3.1 App Isolation 7

1.3.2 Certification of Symbolic Transactions 9

1.4 Organization 10

2 Securing isolated applications with App Isolation **11**

2.1 Background 14

2.1.1 Isolation with multiple browsers 14

2.1.2 Isolation with a single browser 15
2.2 Securely isolating websites 16
2.2.1 Benefits of multiple browsers 17
2.2.2 Site isolation in a single browser 20
2.3 Identifying isolated apps 21
2.4 Entry-point restriction 24
2.4.1 Design 25
2.4.2 Implementation 27
2.5 State isolation 28
2.5.1 Design 29
2.5.2 Implementation 30
2.6 Evaluation 31
2.6.1 Security 31
2.6.2 Complexity of Adoption 38
2.6.3 Performance 40

3 Implementation errors in collaborating applications 42
3.1 Related Work 43
3.1.1 Third-party payment 43
3.1.2 Single sign-on 45
3.2 Field-study of mobile OAuth applications 47
3.2.1 OAuth background 48
3.2.2 Dissecting the OAuth specifications 52
3.2.3 Our Study 57

4 Securing collaborating applications via Certification of Symbolic Transactions **66**
4.1 Overview of CST 70
4.1.1 Threat model 72
4.2 An illustrative Example of CST 72
4.2.1 A traditional implementation 74
4.2.2 The CST-enhanced implementation 77
4.3 The CST Certifier 79
4.3.1 Symbolic Transaction 80
4.3.2 Synthesizer 83
4.3.3 Verifier 85
4.3.4 Cache 87
4.4 Applying CST in the Real World 87
4.4.1 Categories of enhanced systems 88
4.4.2 Ambient Predicates 90
4.4.3 Programming 92
4.4.4 Deployment 95
4.5 Evaluation 96
4.5.1 Security 96
4.5.2 Protocol independence 100
4.5.3 Performance 103
4.5.4 Programming effort 105
4.5.5 Related Work 105

5 Conclusion **109**

Bibliography **119**

List of Tables

2.1 Cross-origin attacks mitigated by entry-point restriction, state isolation, and using separate browsers. Same-origin attacks, such as stored XSS, are not mitigated. 18

4.1 Real-world cases studied in our security analysis. 97

4.2 Performance overhead per transaction and one-time costs. 104

4.3 Lines of code that we added or changed in the open-source packages (comment and white lines excluded) 106

List of Figures

2.1 Entry-point Restriction 25

2.2 Entry-point restriction violation found by Alloy model. 34

2.3 Isolation violation found by Alloy model. Request 4 to a non-entry URL, containing credentials issued within the app, is granted if entry-point restriction is absent or incorrect. 36

2.4 Entry points to popular sites observed by 10,551 Mozilla Test Pilot browsers. Numbers in parentheses indicate the total number of incoming links observed. 37

3.1 OAuth 1.0 and OAuth 1.0a. 52

3.2 Two grant types of OAuth 2.0 53

3.3 Pinterest's OAuth dialogue forged using a stolen application secret. 59

3.4 Usages of the Facebook appsecret_proof flow. Parameters inside square brackets are cryptographically hashed using the relying party secret. 61

3.5 Tencent's enhanced implicit grant for authentication. The variables inside the square brackets are cryptographically hashed using a secret key known to only Tencent. 62

4.1 Traditional approach versus CST. 67

4.2 A simple multiparty system. 71

4.3 The basic messages and data for checkout. 73

4.4 A traditional implementation. 75
4.5 The certifier module on TStore.com. 78
4.6 Stitching two method calls. 81
4.7 Program synthesis from SymT. 84
4.8 An example of the synthesizer discarding an untrusted call. (Note that the single colon with the placeOrder call) 86
4.9 Our tool chain. 87
4.10 PayPal Standard. 88
4.11 SSO based on OAuth-2.0. 88
4.12 SSO based on OAuth-2.0. 89
4.13 The gambling system. 90
4.14 A protocol-independent definition of single-sign-on. 90
4.15 Factoring and stubbing in callback.aspx. 93
4.16 An example wrapper service. 96
4.17 Protocol-violating yet secure implementations. 101

Chapter 1

Introduction

The World Wide Web has become a multimedia platform that evolves constantly and rapidly, while web applications are growing increasingly complex and versatile. Modern Internet users rely on web applications to perform a myriad of highly personal and security-sensitive computing tasks, including personal banking, data storage, e-commerce and private communication. Due to the security-sensitive nature of these tasks, it is vital for developers to ensure the safety and privacy of their data.

Traditionally, the security of web applications rests on the shoulders of their developers. However in the recent years, it has become clear that implementation errors in web applications are prevalent and difficult to mitigate. For example, Cross-Site Scripting (XSS) attack has been well-known for over a decade [1]; but the ratio of websites contain XSS vulnerabilities have not diminished [2]. In a recent study by Lekies et. al., 9.6% of Alexa 5000 websites are vulnerable to DOM-based XSS attacks. Besides XSS, implementation errors within individual, isolated web applications gave birth to many other high profile vulnerabilities (e.g., Cross-site Request Forgery, Clickjacking, and Session Fixation).

In addition to vulnerabilities within individual web applications, communication protocols between multiple collaborating applications are also vulnerable to implementation errors. These errors are often fueled by developers' misunderstanding of certain protocol

details and ambiguities within protocol specifications. In recent years, implementation errors have been discovered on a wide range of popular protocols, including OAuth, OpenID, Amazon Simple Pay, and Paypal Standard. Despite the fact that a significant portion of these protocols have gone through extensive security analysis; a large number of developers continue to fail to implement them correctly. For example, our recent survey of mobile applications using OAuth revealed that 59.7% of the top 600 most popular Android applications have vulnerable OAuth implementations.

To reduce the number of implementation errors in web applications, the conventional practice is to educate developers on common vulnerabilities and train them to adhere to secure programming practices. Unfortunately, above evidence suggests that merely educating developers to follow secure programming practices is insufficient (as shown by the prevalence of DOM XSS and faulty OAuth implementations).

In this thesis, we take on a different approach – we assume that it is inevitable for real developers make mistakes when coding their web applications. Our goal is to design systems that would alleviate the damage done to users and/or web applications under the assumption that these web applications can be implemented incorrectly.

1.1 Threat model

There are two common types of adversaries that threaten the security of web applications: web attackers, and malicious users. These two types of attackers are not mutually exclusive (i.e., a web attacker can also be a malicious user). In this section, we define each of these two attacker types in detail.

1.1.1 Web Attacker

The *web attacker* is a malicious entity first proposed by Jackson [3]. The web attacker controls one or more web servers and domains and can also obtain valid certificates for his domains. Furthermore, we assume that the web attacker has the ability to force any victim user's browser into rendering contents from the malicious websites.

1.1.2 Malicious User

The *malicious user* is a real user of the victim's web application. If the victim's web application contains user accounts that require authentication, we assume the attacker controls a valid account and can interact with the victim application as a legitimate user.

Because the user has full control over his own device and user-agent, he can freely modify any client-side application states and network traffic. Note this network capability is different from that of a *network attacker* as described by Jackson [3]. A malicious user can only intercept and modify traffic sent to his own machine whereas a network attacker could intercept and modify traffic sent to a benign user.

1.1.3 Out of Scope Attacks

We assume the DNS is trusted and the attacker cannot break any cryptographic protocols such as TLS and SSL. We also assume that the benign web service is free of any malware or OS-level vulnerabilities such as buffer overflow. Lastly, for tasks that involve multiple collaborating applications, we treat all developers of non-attacker parties to be cooperative.

1.2 Types of implementation errors

In this thesis, we focus on two types of implementation errors. The first type of errors exist within a single, stand-alone application. They allow a web attacker to bypass the browser's existing security mechanism and gain access to a trusted application's private states. Some common vulnerabilities of this category include Cross-site Scripting, Cross-site Request Forgery, and Session Fixation. The second type of errors are logical errors introduced when implementing communication protocols between multiple collaborating applications. Some common vulnerabilities of this category include flaws in OAuth, OpenId, and Amazon Payment. One can imagine the former category of errors as flaws in the isolation of data; whereas the latter category represents flaws in the distribution of data. For the rest of this section, we briefly describe these two types of errors that motivate our work.

1.2.1 Errors within an isolated application

Dating back to the Mosaic browser in 1993, web browsers have a history of employing a monolithic architecture – that is, monolithic browsers group contents from all websites into a single protection domain. They enable web attackers who can compromise one component of the browser to execute code with the user's privilege [4]. To mitigate this drawback, there has been significant research over the years to make browsers more modular. One of the most influential work in this area is the Chromium [5] browser.

The Chromium browser contains two modules with distinctive protection domains: a browser kernel and a rendering engine. The browser kernel runs with a higher privilege to interact with the user's operating system; while the rendering engine runs with restricted privileges and is responsible for executing untrusted web content. Because the browser kernel is much smaller than the rendering engine in terms of code size, the Chromium

architecture makes it harder for an attacker to compromise a user's operating system by exploiting a rendering engine vulnerability.

Although the Chromium browser does well at protecting a user's operating system from malicious websites, it does not protect websites from each other – popular commercial browsers (including Chrome, Safari, Firefox, and IE) use a shared storage to store application states from multiple, distrusting websites. This is a serious limitation because it allows developers to introduce seemingly benign errors in their code that ultimately expose sensitive application states to malicious entities. For example, consider the case of a cross-site request forgery (CSRF) attack, the adversary issues a subresource request within a page she owns in an attempt perform an unintended action on a trusted site on behalf of the user. This attack succeeds because modern browsers use the same cookie storage for both the malicious and the trusted website. Therefore, when the malicious website issues a subresource request to the trusted website, the users session cookie is attached to this malicious request.

This weakness in modern browsers' application isolation scheme is the root cause of many prevalent web vulnerabilities, such as CSRF, reflected XSS, click-jacking, cross-origin resource import, and session fixation. These attacks are possible because the security boundaries of web applications are not clearly defined in modern browsers. In Section 1.3.1, we introduce a novel browser isolation scheme called App Isolation can mitigate the lion's share of these vulnerabilities. App isolation is also described in detail in Chapter 2.

1.2.2 Errors in multiparty protocol implementations

Many modern web applications are multiparty systems that integrate a variety of third-party services, such as single sign-on (SSO), payment, and social networking. Unfortu-

nately, logic errors are often introduced by developers implementing these protocols, hence undermining security. For example, we conducted a field-study of 149 popular mobile applications that use the OAuth protocol and discovered that 89 of them (59.7%) were implementing the protocol incorrectly and thus vulnerable [6]. In addition to OAuth-related implementation errors, previous work revealed similar errors on other types of protocols as well, including OpenID, Amazon Simple Pay, and Paypal Standard [7, 8]. Note these errors are not flaws within protocol specifications, but rather developers incorrectly interpreting and/or implementing these specifications.

We believe two root causes are responsible for most implementation errors found in multi-party protocols. First, web protocols are often designed and written by security experts. It is common for average developer to misinterpret certain details in the protocol specification, subverting its security. Second, we observe that in practice, developers often do not strictly adhere to protocol specifications.

There are many reasons for why protocols specifications are not strictly followed in practice. Consider the case of the OAuth protocol. The OAuth specification was initially written for web applications. However, the boom of mobile application happened shortly after OAuth gained its popularity. Almost inevitably, OAuth became the protocol for implementing authentication and authorization functionality in mobile applications. Unfortunately, the mobile use-case is left out in the original specification, and many web primitives used heavily in the OAuth specification (such as browser redirection) do not exist for mobile platforms. Because of this, many mobile developers are forced to create their own methods of mimicking browser redirection and other web-specific concepts. Most of these home-brewed techniques turn out to have security loopholes.

To address implementation errors in OAuth and other web protocols, we build a formal verification framework called Certification of Symbolic Transactions (CST). CST

reduces the burden of enforcing security for developers by verifying safety properties of each protocol transaction at run-time. We introduce CST in Section 1.3.2 and describe it in detail in Chapter 4.

1.3 Proposed defenses

To alleviate damage caused by implementation errors described in Section 1.2, we propose two security mechanisms: App Isolation and Certification of Symbolic Transactions. One can imagine App Isolation as a mechanism that protects a single application's sensitive states from the outside; while Certified Symbolic Transaction protects the communication channel between multiple applications. Neither of the two proposed security frameworks is designed to mitigate a specific vulnerability. Rather, they act as safety nets to prevent a wide range of implementation errors from causing damage. For the rest of this section, we introduce our two proposed defenses: App Isolation and Certified Symbolic Transaction.

1.3.1 App Isolation

App isolation is a browser isolation mechanism that creates a dedicated protection domain for each security-sensitive web application. It mimics the effect of running each application in its own separate browser.

The idea of app isolation spawns from the observation that many web vulnerabilities can be prevented by using separate browsers for separate websites. However, most users access the web with only one browser. The security benefits of using multiple browsers can be summarized in terms of two concepts: entry-point restriction and state isolation. App isolation is a security mechanism that combines these concepts to provide the same security benefits in a single browser.

The two components of app isolation work in concert to provide security against a wide

range of implementation errors. First, entry-point restriction requires the web application developer to provide a list of landing pages for the app that untrusted applications can link into. These landing pages are also referred to as entry-points. The browser may load a resource from the web application if and only if at least one of the following statements holds true:

- The resource is requested by a page inside the web application.
- The URL of the resource is a valid entry-point.

One prominent vulnerability foiled by entry-point restriction is reflected XSS. In a reflected XSS attack, the web attacker crafts a malicious URL containing an attack string and navigates the users browser to that URL, tricking the honest website into echoing back the attack string in a dangerous context. However, this attack will not succeed with entry-point restriction because the malicious URL is not a valid entry-point, hence the browser would refuse to load it.

The second component of app isolation is state isolation. State isolation isolates both the in-memory and persistent state of the web application from other web sites, using the process and storage architecture of the browser. State isolation is critical for preventing attacks like click-jacking that do not rely upon the attacker navigating the users browser to a maliciously crafted URL [9].

We implement app isolation in the Chromium browser and verify its security properties using finite-state model checking. We also measure the performance overhead of app isolation and conduct a large-scale study to evaluate its adoption complexity for various types of sites, demonstrating how the app isolation mechanisms are suitable for protecting a number of high-value Web applications, such as online banking.

1.3.2 Certification of Symbolic Transactions

Besides enhancing web applications' isolation mechanism, one must also secure the communication channel between multiple collaborating applications. One established technique in securing multiparty communication protocols is through formal program verfication. However, the adoption of program verification faces several hurdles in the real world: how to formally specify logic properties given that protocol specifications are often informal and vague; how to precisely model the attacker and the runtime platform; how to deal with the unbounded set of all potential transactions.

We introduce Certification of Symbolic Transaction (CST), an approach to significantly lower these hurdles. CST tries to verify a protocol-independent safety property jointly defined over all parties, thus avoids the burden of individually specifying every partys property for every protocol; CST invokes static verification at runtime, i.e., it symbolically verifies every transaction on-the-fly, and thus 1) avoids the burden of modeling the attacker and the runtime platform, 2) reduces the proof obligation from considering all possible transactions to considering only the one at hand.

In essence, CST acts as an optional extension for any given web protocol (e.g., OAuth, OpenID). Protocol implementations that employ CST have an additional field attached to every protocol message. This field is called *Symbolic Transaction* (or *SymT*). SymT is a string that encodes the following information:

1. The source code of method calls executed so far at each party of the protocol.
2. The sequence of method calls so far and how two consecutive calls are stitched – that is, how the output of one method is fed into the input of the next method.

At the last step of the protocol, the final SymT is used to synthesize a program representing the executed source code of the entire protocol transaction. This final program is

then formally verified to determine if certain safety properties are satisfied. Further details on SymT generation, program synthesis and verification are elaborated in Chapter 4.

We have applied CST on five commercially deployed applications, and show that, with only tens (or 100+) of lines of code changes per party, the original implementations are enhanced to achieve the objective of CST. Our security analysis shows that 12 out of 14 logic flaws reported in the literature will be prevented by CST. We also stress-tested CST by building a gambling system integrating four different services, for which there is no existing protocol to follow. Because transactions are symbolic and cache-able, CST has near-zero amortized runtime overhead. We make the source code of these implementations public, which are ready to be deployed for real-world uses.

1.4 Organization

The remainder of this thesis is organized as follows. In Chapter 2 we describe our first defense – app isolation. Then, we demonstrate a prototype implementation in Chromium and evaluate its ease of adoption, security, and performance. In Chapter 3, we describe several logic flaws commonly found in inter-application communication protocols and present our field study of OAuth applications. In Chapter 4 we explain our second defense – CST, and show how we successfully deployed CST to five real world commercial applications. Next, we evaluate CST's security and performance. Lastly, we conclude in Chapter 5.

Chapter 2

Securing isolated applications with App Isolation

The work in this chapter was done in collaboration with Jason Bau, Charles Reis, Adam Barth, and Collin Jackson

Security experts often advise users to use more than one browser: one for surfing the wild web and others for visiting "sensitive" websites, such as on-line banking websites [10, 11]. This advice raises a number of questions. Can using more than one browser actually improve security? If so, which properties are important? Can we realize these security benefits without resorting to the use of more than one browser?

We seek to answer these questions by crystallizing two key security properties of using multiple browsers, which we refer to as *entry-point restriction* and *state isolation*. We find that these two properties are responsible for much of the security benefit of using multiple browsers, and we show how to achieve these security benefits in a single browser by letting websites opt in to these behaviors.

Consider a user who diligently uses two browsers for security. This user designates one browser as "sensitive" and one as "non-sensitive". She uses the sensitive browser only for accessing her on-line bank (through known URLs and bookmarks) and refrains from visiting the general Web with the sensitive browser. Meanwhile, she uses only the

non-sensitive browser for the rest of the Web and does not use it to visit high-value sites.

Using two browsers in this manner does have security benefits. For example, consider the case of reflected cross-site scripting (XSS). In a reflected XSS attack, the attacker crafts a malicious URL containing an attack string and navigates the user's browser to that URL, tricking the honest web site into echoing back the attack string in a dangerous context. The attack has more difficulty succeeding if the user runs more than one browser because the attack relies on *which* of the user's browsers the attacker navigates. If the attacker navigates the user's non-sensitive browser to a maliciously crafted URL on the user's bank, the attack will have no access to the user's banking-related state, which resides in another browser.

From this discussion, one might conclude that isolation of credentials and other state is the essential property that makes using two browsers more secure. However, another security property provided by using multiple browsers is equally important: entry-point restriction. To illustrate entry-point restriction by its absence, imagine if the attacker could arbitrarily coordinate navigation of the users' two browsers and open an arbitrary bank URL in the sensitive browser. Now, the attacker's maliciously crafted URL and attack string can be transplanted from the non-sensitive browser to the sensitive browser, leading to disaster.

In reality, it is extremely difficult for Web attackers to coordinate the navigation of two different browsers on the users' computer. This isolation between the two browsers provides the entry-point restriction property. Namely, sessions in the sensitive browser with an honest web site always begin with a fixed set of entry points (e.g., the site's home page or a set of bookmarks) and then proceed only to URLs chosen by the web site itself, not those chosen by a third party. Because the bank's entry points are restricted, the attacker is unable to inject the attack string into the user's session with the bank.

State isolation, in turn, augments the security provided by entry-point restriction when using two browsers. State isolation plays a critical role, for example, in preventing clickjacking attacks [9] because these attacks do not rely upon the attacker navigating the user's browser to a maliciously crafted URL. State isolation between browsers can even protect a user's high-value session data against exploits of browser vulnerabilities that give the attacker control of the rendering process [5, 12]. In concert, entry-point restriction and state isolation provide the lion's share of the security benefits of using two browsers.

In our example above, we use a single high-value site to illustrate the security benefit of isolation using *two* browsers, but the isolation benefits extend naturally to accessing multiple sites, each in their own browser. In this chapter, we show that we can realize these security benefits within a single browser by allowing web sites to whitelist their entry points and request isolated storage. This is not a pinpoint defense against a specific attack but rather a general approach that has benefits in a number of attack scenarios.

The security benefits of our mechanism do come with a compatibility cost for certain types of websites, as it places some limitations on deep links and third-party cookies. To avoid disrupting existing websites, we advocate deploying our mechanism as an opt in feature. Furthermore, we hypothesize and experimentally verify the types of websites that are suitable for our mechanism. Our experiments measured the number of entry points used by popular sites in a study of 10,551 browsers running Mozilla's Test Pilot platform [13]. Over 1 million links were included in our study. We discovered that many security sensitive sites such as online banking applications can easily deploy our mechanisms. However, highly social or content-driven applications such as Facebook and New York Times will have difficulties adopting our proposal.

To evaluate the security benefits of app isolation, we model our proposals in the Al-

loy language, leveraging previous work on modeling web security concepts in Alloy [14]. We enrich our existing Alloy model with new concepts, such as *EntryPoint*s and *RenderingEngine*s, to model the essential concepts in our proposal. Our analysis revealed two issues with our initial proposals: one related to HTTP redirects and one related to an unexpected interaction between entry-point restriction and state isolation. We repair these errors and validate that our improved proposals pass muster with Alloy.

2.1 Background

In this section, we examine how the security properties of using multiple browsers have surfaced in related work and compare them to our proposal.

2.1.1 Isolation with multiple browsers

For users who choose to browse the web using multiple browsers, site-specific browsers (SSBs) can make the browsing experience simpler and more convenient. SSBs provide customized browsers that are each dedicated to accessing pages from a single web application. Examples of SSBs include Prism [15] and Fluid [16].

SSBs are simply special-purpose browsers and can provide the security benefits of using multiple general-purpose browsers. However, SSBs can become difficult to manage when users interact with and navigate between a large number of different web applications. We show that a single browser can realize the security benefits of SSBs without the management burden on the user. For example, our proposal allows users to seamlessly and securely follow a link from one app to another, even in a single browser tab.

2.1.2 Isolation with a single browser

The concept of finer-grained isolation inside a single browser has been explored by many researchers. However, prior work has not identified the essential factors needed for a single browser to achieve the same security benefits as using multiple browsers.

Recent browsers have begun employing sandbox technology that protects the local file system from attacks that exploit browser vulnerabilities. For example, Internet Explorer on Windows Vista introduced Protected Mode [17], which protects the local file system from being modified by a compromised rendering engine. The Google Chrome browser's sandbox additionally protects the local file system from being read by a compromised by a rendering engine [5]. Unfortunately, neither of these sandboxing technologies protect *web application* state, such as cookies and local storage data, from being accessed by a compromised rendering engine.

The OP browser [18] isolates plugins from state associated with other applications by enforcing restrictions on the cross-origin request API exposed to plugins. The Gazelle browser [19] goes a step further by restricting the cross-origin request API for the entire rendering engine. Under the Gazelle approach, a *web application*'s state is only visible to the rendering engine containing it. This prevents a malicious web entity from compromising its own rendering engine to gain access to the state of other web applications. However, because Gazelle denies rendering engines from requesting cross-origin resources unless their MIME type indicates a library format such as JavaScript or Cascading Style Sheets (CSS), it imposes a compatibility cost on many web sites [20].

One approach that can mitigate the compatibility costs of restricting the cross-origin request API is to allow an application to explicitly declare the URLs that compose it. One example of this approach is the Tahoma browser [21], which allows applications to specify a manifest file listing which URLs should be included in the same protection domain.

Tahoma uses a separate state container for each application, so state associated with one application will be inaccessible in another. Although Tahoma realized the importance of isolating web application state, it did not incorporate the other benefit of using multiple browsers: restricting non-sensitive web sites from directing the user to a sensitive URL.

OMash [22] only attaches cookies to same-origin requests, effectively isolating state within a particular site. Each new entry into a site creates a new session. This approach mitigates reflected XSS, cross-site request forgery (CSRF), and click-jacking, since another site cannot hi-jack an existing session with a hyperlink or iframe. However, the drawback of OMash lies in its inability to maintain user state across multiple browsing sessions.

Content Security Policy [23] attempts to mitigate XSS by allowing web sites to only execute scripts from whitelisted external JavaScript files. SOMA [24] aims to alleviate XSS and CSRF by making the host of web content mutually approve the content request with the web content embedder. Unfortunately, both of these defenses are geared to counter individual attacks such as XSS and CSRF. They do not achieve the full security benefits as using multiple browsers, such as defenses against rendering engine exploits.

In contrast, our work aims to capture the same underlying properties of using separate browsers for sensitive web apps, gaining the security benefits in a single browser.

2.2 Securely isolating websites

In this section, we investigate exactly which security benefits can be achieved by visiting sensitive web sites in a different web browser than non-sensitive web sites. We classify many common browser-based attacks and show that a large number of them can be mitigated through the use of multiple browsers. We then introduce two new mechanisms in a single browser that can be used to achieve these same benefits, for particular web sites that choose to opt in and accept the compatibility implications.

2.2.1 Benefits of multiple browsers

Suppose a user wishes to protect certain sensitive web sites from more dangerous ones by using two browsers, A and B. To achieve this, she must abide by the following rules:

1. Only type in passwords for sensitive web sites with Browser A. This rule ensures that user state for the sensitive web sites are stored only in Browser A.

2. Never type in URLs or click on bookmarks to non-sensitive web sites with Browser A, and never type into Browser A URLs received from non-sensitive web sites or other untrusted sources. This rule prevents Browser A from leaking any sensitive information to Browser B and prohibits Browser B from contaminating sensitive states in Browser A.

If the user strictly abides by the rules above, all sensitive state would reside in Browser A, isolated from non-sensitive web sites and unable to leak to Browser B. Furthermore, the integrity of Browser A is maintained because untrusted content in Browser B cannot infect Browser A, preventing attacks such as reflected XSS.

These rules prevent the "cross-origin" versions of the attacks listed in Table 2.1. We classify attacks as "cross-origin" if the attack is launched from a different origin than the victim origin, as opposed to "same-origin" attacks (such as one Facebook page trying to mount a CSRF attack on another Facebook page). Using a separate browser does not prevent the same-origin versions of these attacks, nor same-origin only attacks such as stored XSS, because the attacker resides in the same browser as the victim.

We provide a more thorough analysis of the cross-origin attacks from Table 2.1 below. We assume the attacker wishes to attack a victim web site to which the user has authenticated and can lure the user into visiting a malicious web site on a different origin. Furthermore, we assume that the user uses separate browsers according to the rules above.

Cross-origin Attacks	Entry-point Restriction	State Isolation	Separate browser
Reflected XSS	✓		✓
Session Fixation	✓		✓
Cross-Origin Resource Import	✓	✓	✓
Click-jacking		✓	✓
CSRF	✓	✓	✓
Visited Link Sniffing		✓	✓
Cache Timing Attack		✓	✓
Rendering Engine Hi-jacking		✓	✓

Table 2.1: Cross-origin attacks mitigated by entry-point restriction, state isolation, and using separate browsers. Same-origin attacks, such as stored XSS, are not mitigated.

- **Reflected XSS** – In a reflected XSS attack, the attacker lures the user into visiting a malicious URL inside the non-sensitive browser. This URL will allow the attacker's script to execute inside the victim's origin. However, because the user is authenticated to the victim site in the sensitive browser, the attacker's script will not have access to the user's session.

- **Session fixation** – In a session fixation attack, the attacker includes a known session ID inside a victim URL, then lures the user into visiting this URL and tricks her into logging in. Once the user is logged in, the attacker can freely impersonate the user with the shared session ID. However, because the user only types her password into the sensitive browser, the attack will fail.

- **Cross-origin resource import** – In a cross origin resource import attack, the attacker's page requests a sensitive resource from the victim's origin as a script or style sheet. If the user were authenticated to the victim site in the same browser, this request can leak confidential information to the attacker. However, the user is authenticated instead in the sensitive browser, thereby foiling the attack.

- **Click-jacking** – Click-jacking attacks overlay a transparent iframe from a victim page over a part of the attacker's page where the user is likely to click. This aims to trick the user into clicking somewhere on the victim's page (e.g., the delete account button) without realizing it. Because the user is authenticated in the sensitive browser, clicking on the transparent victim iframe in the non-sensitive browser will cause no damage.

- **Cross-site request forgery** – In a traditional CSRF attack, the adversary makes subresource requests within a page she owns in an attempt to change the user's state on the victim's server. This attack succeeds because the user's credentials are attached to the attacker's subresource request. However, because the user authenticates only in the sensitive browser, the malicious request will not have a cookie attached, rendering it harmless.

- **Visited link sniffing** – The attacker's web site might attempt to sniff the user's browsing history by drawing visited links in a different color or style than unvisited ones, and then using JavaScript or CSS to discover which have been visited. Although a possible mitigation has been proposed and adopted by several major browsers [25], new attacks have been discovered that can detect browsing history despite the defense [26]. However, if the user uses separate browsers, these browsers have different history databases, so a web site in the non-sensitive browser is unable to discern the browsing history of the sensitive browser.

- **Cache timing attack** – Similar to visited link sniffing attacks, an attacker can measure the time to load a victim resource to determine if the user has visited it [27, 28]. Different browsers have different caches for their web resources, so web sites in the non-sensitive browser cannot detect cache hits or misses in the sensitive

browser.

- **Rendering engine hi-jacking** – A powerful attacker might exploit a vulnerability in the browser's rendering engine to hi-jack its execution. For browsers with a single rendering engine instance (e.g., Firefox and Safari), this would let the attacker access all the user's state, such as the victim site's cookies and page contents. These attacks still apply to browsers with multiple rendering engine instances, if they rely on the rendering engine to enforce the Same-Origin Policy (e.g., Chrome and IE8). However, if the user logged in to the victim site with a different browser, the victim's cookies and sensitive pages will reside in an entirely different OS process. Assuming the exploited rendering engine is sandboxed, the attacker's exploit is unable to access this process.

2.2.2 Site isolation in a single browser

As shown in the previous section, using a dedicated browser to visit certain sites mitigates a significant number of web attacks. This observation raises a question: which properties of browsing with a single browser make it vulnerable to these attacks? We believe the answer to this question can be summarized in three points:

1. Malicious sites are free to make requests to vulnerable parts of victim's site.

2. Malicious sites can make requests that have access to the victim's cookies and session data.

3. Malicious sites can exploit the rendering engine for direct access to in-memory state and to stored data from the victim site.

Our key observation is that these abilities are not fundamental flaws of browsing with a single browser but rather weaknesses of current browsers. We believe that for many types

of web sites, it is possible to simulate the behavior of multiple browsers with a single browser by solving each of these weaknesses. These changes come with a compatibility cost, however, because benign third-party sites are also prevented from accessing the user's cookies. We evaluate the complexity that different types of sites face for adopting these changes in Section 2.6.2.

In the next three sections, we introduce mechanisms for removing these limitations in a single browser. First, we provide a means for web sites to opt in to this protection if they accept the compatibility implications. Second, we prevent untrusted third parties from making requests to vulnerable parts of these web sites. Third, we isolate the persistent and in-memory state of these sites from other sites. Because our approach works best with "app-like" web sites that contain sensitive user data and few cross-site interactions, we refer to this approach as *app isolation*.

2.3 Identifying isolated apps

App isolation can provide a web site with the security benefits of running in a dedicated browser, but it comes at some compatibility cost. Isolating cookies and in-memory state not only prevents malicious web sites from compromising sensitive data, it can also hinder legitimate web sites from sharing information. For example, Facebook Connect[29] lets web sites access visitors' identifying information via Facebook, which would not work if Facebook was isolated in a separate browser. To remain compatible with web sites that desire this sharing, we employ an opt-in policy that lets web developers decide whether to isolate their site or web application from the rest of the browser.

We must choose the opt-in mechanism carefully to avoid introducing new security concerns, and we must consider the granularity at which the isolation should take effect. In this section, we first show the consequences of an inadequate opt-in mechanism using

HTTP headers. Then, we describe a viable origin-wide approach with *host-meta*, and refine it to support sub-origin level web applications with manifest files.

Bootstrapping with HTTP headers As a straw man, we first consider identifying an isolated app using a custom HTTP header (e.g., X-App-Isolation: 1). If the browser receives this header on an HTTP response, the browser treats all future responses from the origin as belonging to the isolated app.

The primary disadvantage of this approach is that it does not verify that the given response has the privilege to speak for the entire origin. This lack of verification lets owners of portions of an origin (e.g., `foo.com/~username/`) opt the entire origin in to app isolation. A malicious sub-domain owner can use this mechanism to prevent desirable sharing on other parts of the origin, or he can misconfigure the app (e.g., listing a non-existent entry point) to perform a denial-of-service attack. Worse, bootstrapping with a custom HTTP response header might not enforce the policy for the initial request sent to the server, opening a window of vulnerability.

Bootstrapping with Host-meta To avoid attacks that grant the privileges of the entire origin to each resource, the browser can instead bootstrap app isolation using a file at a well-known location that can only be accessed by the legitimate owner of the origin. The *host-meta* mechanism is designed for exactly this reason [30]. With *host-meta*, the owner of the origin creates an XML file containing app isolation meta data located at `/.well-known/host-meta`. This meta data can include configuration information, such as a list of acceptable entry points. Because *host-meta* should be controllable only by the legitimate owner of the origin, an adversary controlling only a directory will not be able to influence the app isolation policy for the entire origin.

It is essential to retrieve *host-meta* information through a secure channel such as

HTTPS. Otherwise, an active network attacker can replace the host-meta entries with bogus URLs, allowing the attacker to conduct denial-of-service or other misconfiguration attacks.

One downside of bootstrapping with *host-meta* is that is has poor performance because an additional round trip is required to fetch a resource if the host-meta file is not in the cache.

Bootstrapping with Manifest File The above proposals work at the granularity of an origin. However, it is also possible to isolate web apps at a finer granularity without violating the security concerns of “finer-grained origins” [31].

In the Chrome Web Store, web application developers package their applications using a manifest file [32]. This method of packaging web applications is becoming common; for example, Mozilla’s Open Web Applications are also packaged using a such file [33]. The file includes a list of URL patterns that comprise the application, together with other meta data such as requested permissions. The manifest file provides extra context to the browser for how to treat the app, allowing it to enforce policies that might break ordinary web content. The Chrome Web Store also supports “verified apps [34],” in which the manifest file’s author demonstrates that she has control over all origins included in the application’s URL patterns.

We use additional syntax in the manifest to let applications in the Chrome Web Store opt in to app isolation. The URL patterns in the manifest might or might not span an entire origin, which would allow a site like Google Maps (e.g., `http://www.google.com/maps`) to opt into isolation features without affecting the rest of the origin. The Chrome Web Store already provides a mechanism for verifying that the manifest is provided by the web site author, which we leverage to prevent a malicious manifest file from bundling

attacker URLs in the same app as a victim site.

The reason this does not run afoul of the typical security concerns of finer-grained origins is that our state isolation effectively separates the application's pages from the rest of the web, including non-application pages in the same origin. Origin contamination via scripts or cookies is blocked because an application page and a non-application page do not share the same renderer process or cookie store.

Both origin-level isolation using *host-meta* and application-level isolation using manifest files are viable opt in mechanisms. We leave it to browser vendors to decide on which method they deem appropriate. In the remainder of this chapter, we refer to the unit of isolation as an *app*, whether designated as an origin or a collection of URLs in a manifest.

2.4 Entry-point restriction

Using multiple browsers securely requires the user to refrain from visiting a sensitive app at a URL that could be constructed by an attacker. Instead, the user always visits the app in the sensitive browser from a known starting point. Simulating this behavior with a single browser requires an intuitive way of visiting URLs of sensitive apps without compromising security. Our proposal for *Entry-point Restriction* provides a way to safely transition between sensitive and non-sensitive pages in a single browser, without altering the user's behavior.

In this section, we present the rules for entry-point restriction and discuss the challenges for selecting appropriate entry points. Table 2.1 lists attacks that are prevented by entry-point restriction.

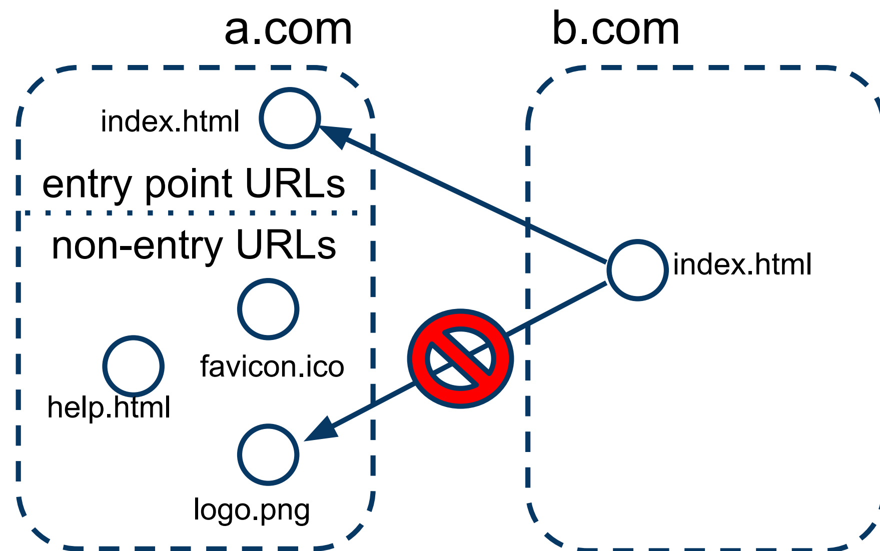


Figure 2.1: Entry-point Restriction

2.4.1 Design

Under the entry-point restriction policy, we define an *entry point* to be a landing page of an app designated by the app's owner. Any app may choose to opt into entry-point restriction by providing at least one entry point.

Once an app opts into entry-point restriction, the browser may load a resource from the app if and only if at least one of the following statements holds true.

- The resource is requested by a page inside the app.
- The URL of the resource is a valid entry point for the app.

Figure 2.1 illustrates how entry-point restriction works in practice. Suppose `a.com` hosts its company logo at `a.com/logo.png`, and specifies an entry point at `a.com/index.html`. Meanwhile, `b.com/index.html` includes a hyperlink to `a.com/index.html` and embeds an image with the source attribute of `a.com/logo.png`. In this scenario, the user

is able to follow the hyperlink because it points to a valid entry point. However, the logo will fail to load because it is a cross origin request to a non-entry-point URL.

Choosing entry points Entry points are whitelisted URLs or URL patterns that the app's owner trusts to load safely even when requested by an attacker's page. Therefore, it is crucial to choose these URLs with care. The goal is to select common landing pages that do not present opportunities for the attacker to exploit the user's credentials. We offer the following guidelines for selecting entry points.

- An entry point URL should cause no sensitive state changes, because an attacker may request it with the user's credentials.
- An entry point URL pattern should be as tightly constrained as possible, reducing the opportunity for attackers to place malicious code in the URL.
- An entry point URL should not return confidential information in a form that could be accessed on a cross-origin page, such as JSON.

In most cases, the default landing page for an app meets these guidelines. Many sites expect their users to arrive via multiple landing pages, creating a need to support multiple entry points. For example, an online banking site may have an English login page and a French login page. For additional flexibility, we also allow web sites to use wildcard characters in their entry point URLs (e.g., www.a.com/*/ where * represents any number of non-'/' characters). Allowing wildcards trades off some security benefits for compatibility because the number of wildcard characters and entry points is directly correlated with the size of the app's attack surface. However, this is still an improvement over having no such policy. We recommend that app owners restrict their policies as tightly as possible and use wildcards only when necessary.

It is important to note that not all web sites are well-suited for entry-point restriction. Some sites depend heavily on deep links to content, such as socially integrated sites like Facebook or content-oriented sites like New York Times. These sites will have a difficult time adopting the policy, because it is extremely difficult to identify all the URLs that legitimate sites may link to. We show in Section 2.6.2 that other types of sites, such as online banks, are amenable to these restrictions and can benefit from entry-point restriction.

Sub-resource restrictions By default, entry-point restriction must deny sub-resources at non-entry-point URLs from loading. This will prevent vulnerabilities such as reflected XSS and CSRF; however, it may also affect legitimate web pages. For example, loading a non-entry-point image will fail despite being typically a safe action.

Fortunately, this usability constraint can be alleviated. Entry-point restriction is only necessary because malicious requests will have the user's authentication tokens attached to them. Section 2.5 describes how State Isolation can be used to isolate these authentication tokens from sub-resource requests. When used in conjunction with state isolation, entry-point restriction can safely allow sub-resources at non-entry-point URLs.

2.4.2 Implementation

We implemented a proof-of-concept entry-point restriction mechanism in the Chromium browser. The entire system consists of less than 100 lines of C++ code. Our implementation enforces entry-point restriction inside Chromium's WebKit rendering engine. More specifically, we modified the *CanDisplay()* function of *SecurityOrigin*, which gets called before every web resource request. If the URL of a web resource violates the entry-point restriction policy, WebKit will not issue the network request.

Storing entry points Like most web resources, it is desirable for the browser to cache entry-point restriction information for performance. The exact caching method may differ depending on how apps opt into entry-point restriction.

If apps use host-meta to bootstrap app isolation, browsers could cache this information like conventional web resources. Users should be able to clear their app isolation information the same way they clear cookies or browsing histories. Most modern browsers offer users with private browsing features that allow them to browse without persistent storage [35]. To be compatible with these private browsing features, the host-meta information in such modes must be treated like cookies or browsing history, not being written to disk.

If apps instead use app manifests (such as the installed apps from the Chrome Web Store), the policies are stored in the browser's persistent app meta data. This essentially permits the browser to permanently cache the app isolation policies as long as the application is installed. To update the policy, developers can use the standard app update process.

2.5 State isolation

The remaining security benefits of using multiple browsers shown in Table 2.1 result from isolating an app's state from other web sites. In traditional browsers, attackers can try to take advantage of persistent state, such as using an app's cookies in a CSRF attack. They can also try to directly access in-memory state by exploiting the browser's rendering engine and then inspecting memory.

We can simulate the state benefits of using a separate browser for an app with a single multi-process browser. This requires isolating both the in-memory and persistent state of the app from other web sites, using the process and storage architectures of the browser.

2.5.1 Design

Once the browser identifies the URLs comprising an isolated app (as discussed in Section 2.3), it can ensure instances of those pages are isolated in memory and on disk.

In-Memory State Any top-level page loaded from a URL in the app's manifest must be loaded in a renderer process dedicated to that app. Any sub-resource requests are then made from the same process as the parent page, even if they target URLs outside the app's manifest.

We treat sub-frames in the same manner as sub-resources. While this may open the app's process to attacks from a non-app iframe, the app does have some control over which iframes are present. Similarly, an app URL may be requested as an iframe or sub-resource outside the app process. The potential risk of this approach, that of framing attacks, is mitigated by persistent state isolation as described below, which ensures that such requests do not carry the user's credentials. This approach has the same security properties as loading the app in a separate browser.

Top-level pages from all other URLs are not loaded in the app's renderer process. Combined with an effective sandbox mechanism [5], this helps prevent an exploited non-app renderer process from accessing the in-memory state present in the app's process.

The browser kernel process can then take advantage of the process isolation between apps and other sites to enforce stricter controls on accessing credentials and other resources. HTTP Auth credentials, session cookies, and other in-memory state stored in the browser kernel is only revealed to the app's process.

Persistent State The browser kernel also creates a separate storage partition for all persistent state in the isolated app. Any requests from the app's renderer process use

only this partition, and requests from other renderer processes do not have access to it. The partition includes all cookies, localStorage data, and other local state.

As a result, a user's session within an app process is not visible in other renderer processes, even if a URL from the app is loaded in an iframe outside the app process.

The storage partition can also isolate the browser history and cache for an app from that for other web sites. This can help protect against visited link and cache timing attacks, in which an attacker tries to infer a user's specific navigations within an app.

Combining with Entry-point Restriction When both entry-point restriction and state isolation are used together, the mechanisms complement each other and we can relax one of the restrictions for entry-point restriction. Specifically, a non-app page can be permitted to request non-entry-point URLs for sub-resources and iframes. This mimics the behavior when using a separate browser for the app, and it still protects the user because credentials are safely restricted to the app process.

2.5.2 Implementation

We implemented state isolation for apps in Chromium with roughly 1400 lines of code. For in-memory state isolation, Chromium already offers stricter process separation between installed web apps from the Chrome Web Store than most web sites. Pages from URLs in an app manifest are loaded in a dedicated app process. In the general case, Chromium avoids putting pages from different origins in the same process when possible, but cross-origin pages can share a process in many cases to avoid compatibility concerns [36].

However, we needed to strengthen Chromium's process isolation to more thoroughly prevent non-app pages from loading in the app's process. First, we needed to ensure that apps are not placed in general renderer processes if the browser's process limit is reached.

Second, we needed to ensure navigations from an app URL to a non-app URL always exit the app's process.

For persistent state isolation, we changed Chromium to create a new URL context (a subset of the user's profile data) for each isolated app. The cookies, localStorage data, and other persistent information is stored on disk in a separate directory than the persistent data for general web sites. The browser process can ensure that this data is only provided to renderer processes associated with the app, and not to general renderer processes.

2.6 Evaluation

In this section, we evaluate state isolation and entry-point restriction in three ways. First, we perform a formal analysis for the security properties of these mechanisms. Second, we experimentally assess the feasibility of various web sites adopting these mechanisms. Finally, we quantify their performance overhead relative to using one or multiple browsers.

2.6.1 Security

We used model-checking to evaluate the combined security characteristics of app isolation using both state isolation and entry-point restriction. Our approach consists of defining the security goals of app isolation, then modeling our implementation, its security goals, and attacker behavior in the web security framework described in [14] using Alloy [37, 38], a declarative modeling language based on first-order relational logic. We then analyze whether the expressed goals were met with the help of the Alloy analyzer software.

Security Goals The broad security goal of both our mechanisms are *isolation*. Isolation protects sensitive resources belonging to the app, such as non-entry URLs, scripts, and user credentials, against unauthorized use by web pages or scripts not belonging to the

app. We distill two *isolation* goals which, if met, will provide the app with defenses against the attacks described in Section 2.2.1. (This property holds because the attacks either require an attacker to gain access to exploitable URLs within the app or use sensitive state from the app, or both.)

These goals are modeled by Alloy *assertions* (logical predicates whose consistency with the model may be checked) analogous to the following statements:

1. Browser contexts (pages or scripts) originating outside an app will not read or overwrite state issued within the app, such as credential cookies.

2. Browser contexts (pages or scripts) originating outside an app will not obtain a non-entry resource within the app.

Isolation Mechanisms We model entry-point restriction as an Alloy *fact* (a logic constraint which always holds), reproduced below. The *fact* states that the *browser* will not issue any cross-origin requests for a non-entry resource in an entry-restricting origin.

```
fact StrictEntryEnforcement {
    all sc:ScriptContext |
        sc.location=StrictEntryBrowser implies
            no areq:sc.transactions.req |
                areq.path=NON_ENTRY and
                isCrossOriginRequest[areq] and
                isRequestToStrictEntryOrigin[areq] }
```

To model state isolation, we refined the browser model of [14] by adding a set of `RenderingEngine`s associated with each `Browser`. Each `RenderingEngine` then runs a set of `ScriptContext`s, as shown in the Alloy *signatures* below:

```
sig Browser extends HTTPClient {
    engines: set RenderingEngine }

sig RenderingEngine {
   contexts: set ScriptContext,
   inBrowser : one Browser }
```

The actual state isolation is modeled by Alloy *facts*. The first fact states that each `cookie` in the model is tagged with the `RenderingEngine` of the `ScriptContext` in which it was first received. The next states that access to `cookie`s are restricted to only `ScriptContext`s from an origin matching the domain setting of the `cookie` executing in a `RenderingEngine` matching the `cookie` tag.

Our app container model also includes a browser behavior relevant to app isolation, as described by Section 2.5.1: it associates a newly opened `ScriptContext` with the existing `RenderingEngine` of an app if the top-level URL of the new `ScriptContext` is within the app.

Finally, our modeling assumes that users will behave conservatively within an isolated app window, meaning attackers cannot get their `ScriptContext`s in the same `RenderingEngine` as an app when separate `RenderingEngine`s exist.

Web and Rendering Engine Attacker We then modeled the abilities of the attacker. As described in [14], the abilities of web attackers include ownership of a web server by which they can introduce `ScriptContext`s under their control into the user's `browser`. Our modeled "rendering engine attackers" can additionally create scripts that compromise the `RenderingEngine` of the user's browser, giving them arbitrary control over other `ScriptContext`s on the same `RenderingEngine`, such as reading cookies, creating new `ScriptContext`s, sending requests, etc. However, our model assumes that storage isolation is enforced by an entity outside the rendering engines, like a browser kernel. Thus,

the "rendering engine attacker" cannot compromise the storage isolation mechanism.

Entry Restriction Results We first checked our implementation of entry restriction against the stated security goals and found that assertion 2 above was violated. We confirmed that this violation also existed in our implementation at that time and note (with some sheepishness) that the implementation bug resembles ones previously found by [14] in Referer validation defenses proposed by [39].

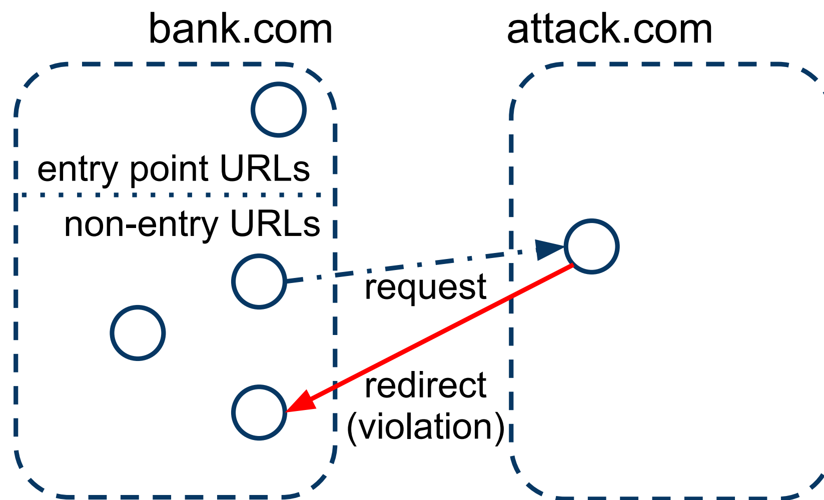


Figure 2.2: Entry-point restriction violation found by Alloy model.

The violation, illustrated in Figure 2.2, occurs because of HTTP redirects. Suppose origin *bank.com* is a victim origin that uses entry-point restriction, and origin *attack.com* is a external origin that does **not** use entry-point restriction. A page created by *bank.com* is allowed by the browser to cause a request for a non-entry resource in *attack.com*, since *attack.com* does not use entry-point restriction. *attack.com* may then issue a redirect to the browser telling it to find the requested resource back at *bank.com*. The browser will then re-issue the request, now to *bank.com*, which will be granted by *bank.com* because

the request was initiated by a context owned by *bank.com*. This violates the integrity goal because the external origin *attack.com* plays a role in redirecting the request back to *bank.com*, thus "requesting" the non-entry resource.

To fix this violation, we updated our implementation to keep track of all redirects experienced by a request and to refrain from sending a request for a non-entry resource to an entry-isolating domain if any external domains are recorded in the request's redirects. We verified that the model containing this fix now upholds the previously violated *integrity* assertion, up to the finite size we tried (up to 10 `NetworkEvent`s, which are either requests or responses).

App Isolation Results We then used the model to check both app mechanisms (entry-point restriction and state isolation) and found that neither mechanism individually was able to uphold the security goals in the presence of the rendering engine attacker. For entry-point restriction without state isolation, there is only a single `RenderingEngine` for all `ScriptContext`s, letting the attacker trivially violate assertion 1 above by compromising the `RenderingEngine` and accessing cookies issued by the app. For state isolation without entry-point restriction, the Alloy analyzer found a violation of assertion 2 above that is very similar to the violation we found in the model with both mechanisms. We will therefore describe both results together in the next paragraphs.

When checking the model with both mechanisms, the Alloy analyzer found no violations to assertion 1 above but did find a violation for assertion 2. This violation occurs because our implementation at the time did not consider the app container as part of its notion of "same-origin" when applied to entry-point restriction. Figure 2.3 illustrates this violation, as well as the violation found for state isolation without entry-point restriction. The scenario is as follows:

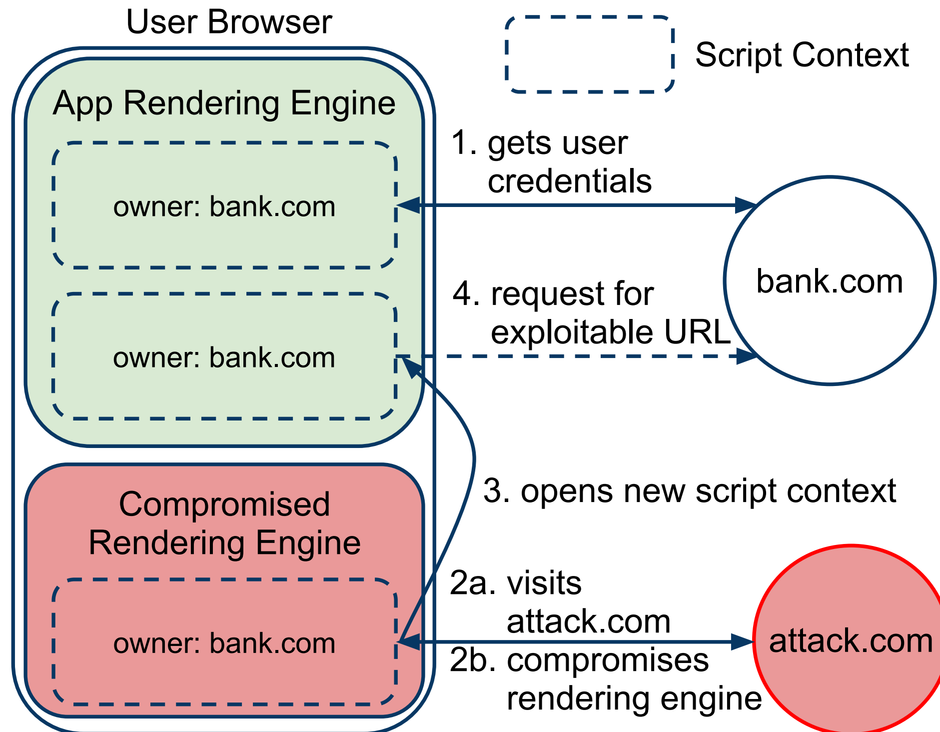


Figure 2.3: Isolation violation found by Alloy model. Request 4 to a non-entry URL, containing credentials issued within the app, is granted if entry-point restriction is absent or incorrect.

Alice opens a session with *bank.com* as an app with a dedicated *app* renderer and receives credentials. Alice also visits *attack.com* with the browser's *ordinary* renderer, causing *attack.com* to send a script which compromises the *ordinary* renderer. The attacker creates a *bank.com* script context in the *ordinary* renderer. Then the attacker causes the *bank.com* script context to open a new window with top-level URL pointed at an exploitable non-entry URL within the *bank.com* app. This new window will open in the *app* renderer, because its initial URL is within the app, and its request for the non-entry URL will pass entry point checks, because the script context which caused the request is "same-origin" (owned by *bank.com*). Similarly, the request will also be sent if entry-point restriction is absent. This last request thus causes the attack to succeed.

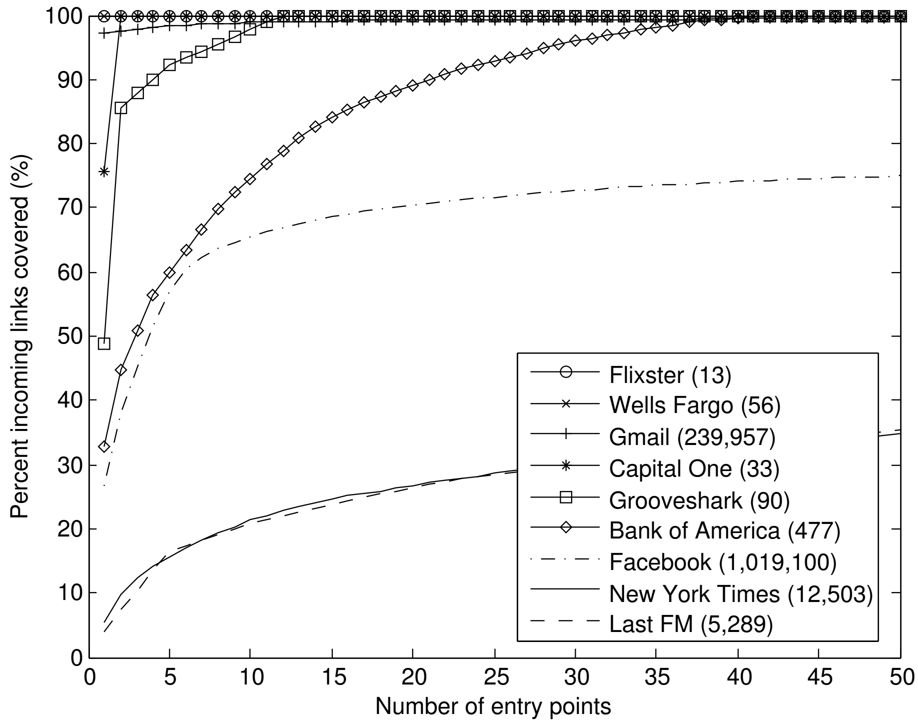


Figure 2.4: Entry points to popular sites observed by 10,551 Mozilla Test Pilot browsers. Numbers in parentheses indicate the total number of incoming links observed.

The discovery of this vulnerability underscores two points regarding app containers. The first is that **both** entry-point restriction and state isolation mechanisms are necessary to stop a rendering engine attacker. The second is that the same-origin policy must be extended to include app containers. In effect, app containers divide a previously atomic origin into two new origins, one inside and one outside the container. As such, the entry-point restriction policy should have rejected the last request in our scenario, because its source was outside the app container and its target was inside. We implemented this updated notion of same-origin in our model and verified that both *assertion*s were now upheld, up to the finite sizes we tried, again up to 10 `NetworkEvents`.

2.6.2 Complexity of Adoption

Entry-point restriction requires web sites to identify all URLs that they wish make public to other web sites. Highly socially integrated sites like Facebook or content-oriented sites like New York Times will have difficulty adopting entry-point restriction due to the inherent complexity in capturing all the possible entry points. On the other hand, we believe non-social web sites such as online banking applications will have an easier time identifying valid entry-points, making it practical to deploy entry-point restriction.

To gain additional insight into the effort required for different web sites to adopt entry-point restriction, we used the Mozilla Test Pilot platform [13]. Mozilla Test Pilot is a Firefox extension installed on more than 3 million active Firefox browsers. Our evaluation was performed on 10,551 of those browsers over a period of 3 days. In our study, we simulated entry-point restriction for 9 web sites shown in Figure 2.4. These sites consist of popular email and web applications, online banking pages of major financial institutions, news and social networks, and popular Chrome Web Store applications.

For each site, we gathered URL hashes of all *incoming links* to the site that appear on all pages that Test Pilot Browsers visited. Each incoming link represents either an actual HTML link or an embedded resource pointing to the site (e.g., `a.com/logo.png` would be an incoming link to `a.com`). Our results demonstrate several ideas. First, they verify our hypothesis that web sites that encourage the sharing of content will have a difficult time opting in to entry-point restriction (e.g., New York Times, Facebook, and Last.fm). Second, web sites with no intention of sharing content can opt in to entry-point restriction with relatively few entry points (e.g., Wells Fargo, Capital One and Flixster, a Chrome Web Store application). For certain sites such as Gmail and Bank of America, the compatibility with entry-point restriction is less clear. While 10 entry points can cover up to 95 percent of the incoming links, fully covering all incoming links appears to

be a non-trivial task. For Gmail, we suspect this to be due to Gmail widgets, multiple login URLs (e.g., `mail.google.com/mail/u/0/`), and web mail portals from numerous organizations hosted by Gmail (e.g., `mail.google.com/a/west.cmu.edu/`). It is difficult to confirm this hypothesis from the data set, as we only collected hashes of the URLs to protect user privacy. For the online banking application for Bank of America, each entry point *path* is valid at a number of regional load-balancing *domain*s, thus accounting for the large number of total entry point URLs.

Web applications that consist of multiple subdomains sometimes face an interesting challenge, if some subdomains are more amenable to app isolation than others. For example, a bank may have some of its login-guarded functionality on a `online` subdomain, while also having a separate `creditcards` subdomain with significant numbers of entry points. If pages on the `creditcards` subdomain can also recognize when a user is logged in, then it is difficult to isolate the two subdomains from each other. Such apps may either face difficulties adopting app isolation or be forced to specify less precise entry points.

To assist web site owners with identifying valid entry points and determining whether app isolation is suitable for their site, we propose a *report-only* mode similar to that of Content Security Policy [40]. Instead of enforcing a policy violation, report-only mode will send a violation report to the app's server. Report-only mode can thus be used to generate a suitable policy file that avoids false positives.

Web developers interested in adopting app isolation should consider the specific feature trade-offs they will be making. Their apps should have limited deep incoming links as entry points. They should not rely on authenticated resources from third parties, and they should not depend on their own authenticated resources being loaded on third party sites. Overall, we found that certain types of sites, including several online banks, do appear to be good candidates for adopting app isolation.

2.6.3 Performance

In this section, we evaluate the performance overhead of app isolation due to entry-point restriction and state isolation. While extra disk space is required for isolated caches, the overhead is generally far less than using multiple browsers.

Navigation Latency

In an entry-point restriction enabled browser, every web resource load for an app is preceded by an entry-point check. This check determines whether the URL of the web resource matches one of the known entry-points. Entry-point lookup can be made efficient using a hash table, imposing negligible cost on navigation latency. We measured the load times of the Alexa Top 100 Web sites with and without entry-point restriction enabled. For an artificially high list of 10000 entry points, the overhead incurred from hash table lookups was small enough to be lost in the noise (less than 0.1 ms per page load).

Besides entry-point lookup, policy files must also be fetched. The fetch of the policy file is done only once at app installation time, and thus we do not include it in the performance overhead.

Storage and Memory Overhead

To see the impact of state isolation, we measured the disk and memory space required for visiting 12 popular sites in their own tabs, similar to the sites used in Figure 2.4. Chromium stores a user's persistent state in a configurable profile directory, so we compared three conditions: all sites in a single Chromium profile, all in a single Chromium profile as isolated apps, and each in a separate Chromium profile. For sites that did not require HTTPS, we used pre-recorded network data to reduce variability. For Gmail, Bank of America, and Chase Bank, we logged into an account. We report the average of

three trials.

Visiting all sites in a single profile required 19 MB of disk space. Using isolated apps required 86 MB, while multiple browsers required 117 MB. Each of these profiles includes a partial download of Chromium's Safe Browsing database (2.6 MB), which is a source of overhead for each additional browser profile.

We were surprised that isolated apps required over 4 times the space of a single profile. This is because Chromium aggressively allocates disk space for each cache. This behavior could be modified to be less aggressive for isolated apps. Users could also opt for an in-memory cache for isolated apps, which retains the security benefits and lowers the disk space required to 9.6 MB.

The total resident memory required for visiting all sites in separate tabs of a single profile was 729 MB.[1] We found that using isolated apps used a comparable 730 MB, while using a separate browser for each site used an aggregate of 1.83 GB memory.

These results show that by using isolated apps rather than multiple browsers, we can reduce the performance trade-off required for our security benefits.

[1] Chromium over counts memory usage by not fully accounting for shared memory, but this is consistent across our three conditions.

Chapter 3

Implementation errors in collaborating applications

The work in this chapter was done in collaboration with Shuo Chen, Yutong Pei, Yuan Tian, Robert Kotcher, and Patrick Tague

Modern web applications are often multiparty systems that integrate third-party services, e.g., single sign-on (SSO), payment, social networking, from different companies. These multi-party applications have significant security requirements. For example, a merchant website that integrates Amazon Simple Pay must ensure that an item is shipped only if it has been paid for. Similarly, an application that integrates Facebook SSO must ensure that only a properly authenticated user must be allowed to log into the application. To implement secure multiparty applications, developers typically follow protocol standards provided by organizations such as OAuth Working Group and OpenID Foundation or API specifications provided by companies such as Amazon, PayPal, and Facebook. However, there is ample evidence that deployed multiparty services are rife with security vulnerabilities. An attacker can purchase without paying [41, 7], sign into other peoples accounts without passwords [42, 8], or get unintended authorizations [43]. The Cloud Security Alliance cites these logic flaws in online services as “Insecure Interfaces and APIs”, the No.4 cloud computing threat [44].

In this Chapter, we demonstrate the prevalence of implementation errors in multi-party protocols. First, we summarize results from existing literature on vulnerabilities with payment and single sign-on protocols. Next, we present our own field-study of 149 popular mobile applications that implement the OAuth protocol. These evidences strongly suggest that implementation mistakes in multi-party protocols are common and difficult to prevent.

3.1 Related Work

There has been ample amount of work focused on finding implementation errors in different types of web services. Two of the most popular web services are third-party payment and single sign-on. Both of these services have been thoroughly studied. Furthermore, vulnerabilities have been discovered in many high-profile websites implementing these services [7, 8]. In this section, we summarize the existing work and illustrate examples of these vulnerabilities.

3.1.1 Third-party payment

Many e-commerce websites accept payments through third-party cashier services. Popular cashier services include Paypal, Amazon Payments, and Google Checkout. Using a third-party cashier service, e-commerce websites typical process customer payment through the following steps:

1. The user arrives at the final checkout page showing the item he wishes to purchase and a button indicating the option to make this purchase through a third-party cashier (e.g., Paypal, Amazon, or Google). For this example, assume that the user wants to make the purchase through Paypal.

2. The user clicks on the Paypal button and his current browser tab is redirected to Paypal. The user then logs into Paypal and accepts the purchase using his banking information stored in Paypal.

3. Paypal then redirects the user back to the original merchant website that the user visited in Step 1. At this point, the merchant website verifies the response it receives from Paypal. If the response is valid, the merchant then acknowledges that the transaction is completed and delivers the item to the user.

Because the threat model for payment protocols involves a malicious user, the merchant website needs to be careful on what information to trust. Unfortunately, the concept of a malicious user is foreign to many web developers. A study done by Wang et al. [7] revealed that multiple popular online stores were implementing payment protocols incorrectly. As a result, they (acting as attackers) were able to purchase various products for free. To get a sense of these implementation errors, we now present 4 real world vulnerabilities taken directly from the aforementioned study.

- **Unchecked order price** – NopCommerce, the most popular .NET-based open source merchant software did not verify the price of the order when the payment was successfully processed on Paypal. This made it possible for an attacker to freely modify the order price when being redirected to Paypal and purchase an expensive item for far less.

- **Unchecked merchant ID** – The same merchant software NopCommerce had a different implementation error for Amazon Simple Pay. In this case, it did not verify the Merchant ID when Amazon successfully processed the payment. This allowed an attacker with a valid merchant account on Amazon to purchase an item

from himself but redirect the payment successful message to the benign merchant using NopCommerce. At this point, the merchant would believe that the attacker successfully paid for the item when the attacker actually paid himself.

- **Update shopping cart after payment** – This is a vulnerability in Interspire's Google Checkout integration (Interspire is a leading e-commerce application). In this implementation, the order generation function and the payment function are not atomic. Meaning, a malicious user could add additional items to his order even after his original payment went through. This additional order would be accepted by Interspire unnoticed.

- **Replay attack** – Interspire's shopping cart implementation was also vulnerable to replay attacks. An attacker could pay for an item once but successfully checkout multiple times (this could result in a benign merchant making multiple deliveries without additional payment).

3.1.2 Single sign-on

Single sign-on (SSO) is another popular third-party service used by many web applications. SSO enables a user to authenticate once with an *identity provider* (e.g., Google, Facebook, or Microsoft) to gain access to multiple *relying party* websites (e.g., websites that allow users to log-in using Facebook) without having to re-enter their login credentials. Authenticating a user though a SSO service provider is done in the following steps:

1. The user arrives at the login page of a relying party website. The relying party presents the user with options to login through an identity provider. For this example, assume the relying party to be *Sears.com* and the identity provider to be *Facebook*.

2. The user clicks on the "login using Facebook" button and is redirected to Facebook (assume he is already logged in). Facebook then prompts the user with a dialog message verifying his intent to authenticate with Sears. The user confirms his intent.

3. Facebook redirects the user back to Sears and sends (through a secure channel) the user's unique Facebook ID to Sears. After obtaining this ID, Sears can do a database lookup to associate this ID with an existing Sears user. At this point, the user is authenticated.

Two of the most popular SSO protocols today are OpenID and OAuth. The security of real world implementations of these protocols have been thoroughly studied in various literature [42, 8, 43]. Unfortunately, all existing literature arrives at the same conclusion – that real world deployments of SSO protocols are rife with security vulnerabilities and securing them is difficult. To get a sense of these implementation errors, we now present 3 real world vulnerabilities taken directly from [8].

1. **Client forge-able user ID** – This vulnerability existed in *Smartsheet.com* which used Google's implementation of OpenID (called Google ID). For this attack, a malicious user could impersonate any user using Google ID by directly tampering with the message sent from Google (the identity provider) to Smartsheet. This attack was possible because the user's identity on Google ID was sent to Smartsheet through an insecure channel (via browser redirection) and it was possible to force Smartsheet into accepting a plaintext version of this message (in the secure version, the user ID was signed by Google).

2. **OpenID's Data Type Confusion** – This vulnerability existed in several online shopping websites that used OpenID, including *toms.com* and *shopgecko.com*.

OpenID allows relying parties to take different data types as IDs for authentication. For example, one data type could be a user's Twitter handle, another data type could be a user's first name registered in Google SSO. Unfortunately, the user himself could change which data-type to use during the authentication process. This allowed an adversary named Bob, a Google user, to register his first name as "AliceOnTwitter", which was coincidentally Alice's Twitter ID, and log in as Alice.

3. **Relying party mis-match** – This was a vulnerability within JanRain, an identity provider wrapper service used by over 350,000 websites. Through JanRain, a user could choose to authenticate through a wide-list of identity providers (e.g., Google, Facebook, or Twitter). After successfully authentication the user, JanRain would return a secret token to the relying party. This token can then be used to fetch the user's JanRain profile and his unique JanRain user ID. Unfortunately, a vulnerability existed that allowed a malicious relying party to impersonate benign relying parties and get unsuspecting users to authenticate. Once successfully authenticated, JanRain would return the user's secret token to the malicious relying party. The attacker could then use this token to login to the user's benign relying party account.

3.2 Field-study of mobile OAuth applications

In order to secure multi-party protocols from implementation errors, it is important to determine the root cause of these errors. Hence, we ask ourselves the question: what makes multi-party protocols in general prone to implementation errors? To answer this question, we start by studying OAuth, which is arguably the most popular web protocol deployed today.

Our study focuses on the top 600 Android applications taken from the Google Play

store, of which 149 (24%) of them used OAuth. The result is worrisome: among the 149 applications in our study, 89 of them (59.7%) were incorrectly implemented and thus vulnerable. The focus of our study was to dissect the rationales behind different OAuth implementations, and to understand why some of these applications are secure while others are seriously vulnerable.

Our analysis reveals that the OAuth specification is extremely extensible, and in a way, underspecified by design. This causes real-world OAuth implementations to be extremely diverse; rarely do two service providers (or even relying parties of the same service provider) share the same protocol flow (e.g., Hulu, Spotify and Instagram all use Facebook for authentication but each have their own protocol flows). We believe that this diversity of OAuth implementations reflects the real issues with OAuth. Not only is the protocol defined over multiple specifications with two different use-cases, but also its mobile usage is poorly defined and underspecified. This forces developers to resort to their own interpretations when implementing the protocol.

In the remainder of this section, we pinpoint the key portions in each OAuth protocol flow that are security critical, but are confusing or unspecified for mobile and web application developers. We then show several representative cases to concretely explain how real implementations fell into these pitfalls.

It has to be noted that although our study focuses on mobile applications, a large portion of our findings are platform agnostic. Specifically, the OAuth related issues discussed in this section apply to both web applications as well as mobile applications.

3.2.1 OAuth background

Originally, OAuth was designed to provide a secure *authorization* mechanism for *websites*. It defines a process for end-users to grant a third-party website the access to their private

resources stored on a service provider. The third-party website is often referred to as the consumer [45], client [46], or relying party [8] (we will use the term "relying party" exclusively in this thesis). There are two versions of OAuth protocols, OAuth 1.0 and OAuth 2.0 [45, 46]. They are both actively use by real-world websites.

Ever since OAuth was successfully adopted by the industry, major companies have re-purposed OAuth for *authentication* as well (i.e., single sign-on). That is, the protocol enables a user to prove his/her identity to a relying party, utilizing his/her existing session with the service provider. The web industry's trend to obsolete other protocols and move toward OAuth is decisive – the new authentication mechanisms provided by the aforementioned companies are all OAuth-based. Therefore, despite the fact that neither OAuth 1.0 nor OAuth 2.0 documentation is explicitly geared for authentication, OAuth is now a de-facto authentication and authorization protocol.

OAuth 1.0 and OAuth 1.0a

When the first version of OAuth was drafted, there existed another popular protocol called OpenID for third-party user authentication [47]. Hence, OAuth was mainly designed to address an issue that was not covered by OpenID – secure API access delegation (i.e., authorization). While the term "API authentication" was occasionally used to describe the functionality of OAuth [48], the protocol specification itself was never meant for user authentication.

The OAuth 1.0 protocol flow is illustrated in Figure 3.1. All dashed lines in our figures represent browser redirection and solid lines represent direct server-to-server API calls (e.g., a SOAP or REST API call). In addition, parameters inside square brackets are signed using shared secrets, which we describe in detail in Section 3.2.2. For now, we present a summary of the protocol flow.

- **Unauthorized request token** (Step 1,2) – First, the relying party obtains a request token from the service provider using a direct server-to-server call.
- **Authorized request token** (Step 3-5) – Then, the relying party redirects the user to the service provider (possibly via a browser redirection) with the request token as a URI parameter. Then, the user grants the relying party access to his/her protected resource and is redirected back to the relying party.
- **Access token** (Step 6,7) – At this point, the relying party can exchange the request token for an access token using another direct server-to-server call with the service provider.
- **Protected resource** (Step 8,9) – Finally, this access token is used to obtain the user's protected resource.

Two years after the release of the OAuth 1.0 draft, a session fixation attack was discovered against the request token approval step of the protocol [49]. To fix the vulnerability, a revision to the original protocol was released (called OAuth 1.0a), which included a verification code to the final request token response (Step 5 of Figure 3.1). This code is used during the access token exchange to prove that the user completing the access token exchange is the same user who granted access. For simplicity, we will use the term "OAuth 1.0" to refer to OAuth 1.0a for the rest of this chapter.

OAuth 2.0

OAuth 1.0 (and OAuth 1.0a) had several notable limitations for its usage scenarios. However, instead of augmenting the existing protocol, the working group decided to alter the specification completely to create a different protocol – OAuth 2.0. This decision was the

result of a "strong and unbridgeable conflict" between different interest groups, according to a departing lead author of OAuth 1.0 [50].

One major change introduced by OAuth 2.0 was the concept of *bearer tokens* [51]. That is, a user's access token was no longer bound to a relying party; any party in possession of this token could freely access the user's protected resource. In addition, OAuth 2.0 also offers four methods for exchanging access token; these methods are referred to as *grants* and they can be viewed as different "versions" of OAuth 2.0. Our study reveals that out of the four grant types in OAuth 2.0, only two were used in practice for authorization and authentication – implicit grant and authorization code grant. We illustrate these two grants in Figure 3.2a and Figure 3.2b, and briefly describe them below.

- **Implicit grant** – The implicit grant is the shortest of all OAuth flows. It consists of two steps. First, the user is redirected to the service provider to grant the relying party access to his/her protected resource. After the permission is granted, the service provider redirects the user back to the relying party along with an access token. The relying party can then use this access token to exchange for the user's protected resource.

  There are two core features that differentiate the implicit grant from other OAuth flows. First, with exception to the final protected resource request, every message in the protocol is exchanged through the user agent (e.g., using browser redirection). Second, the implicit grant does not require the relying party to present a shared secret to the service provider. This is ideal for the mobile environment, where the relying party resides on an untrusted device.

- **Authorization code grant** – The authorization code grant augments the implicit grant by adding an additional step to authenticate the relying party. After the user

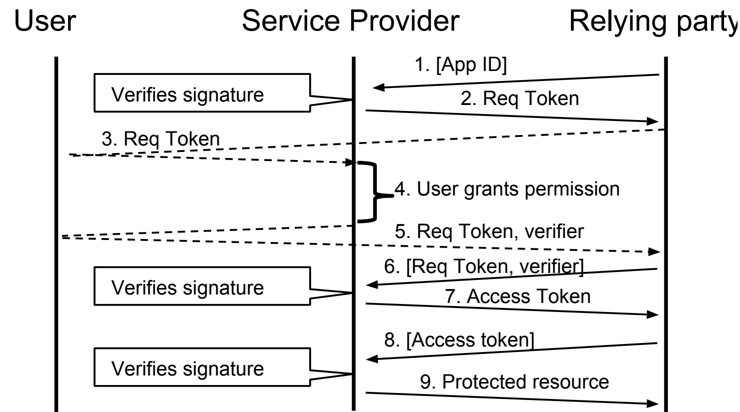


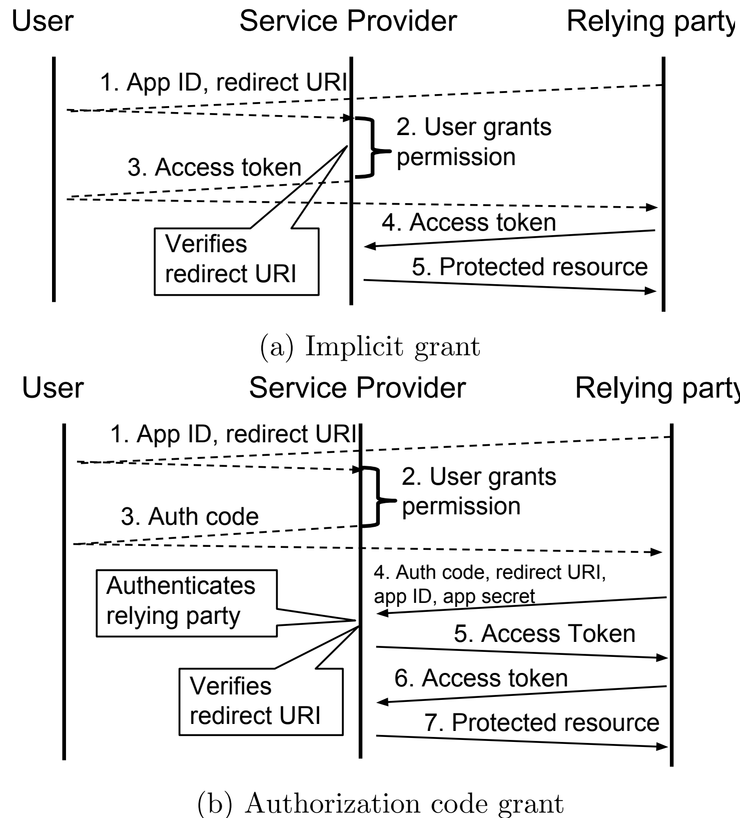
Figure 3.1: OAuth 1.0 and OAuth 1.0a.

grants permission to the relying party, the service provider redirects the user back to the relying party with an *authorization code* (instead of an access token). Then, this authorization code is used to exchange for the user's access token through a direct server-to-server call. In this access token exchange step the relying party has to include its own identity, so the service provider can verify that the authorization code is granted to the same party.

3.2.2 Dissecting the OAuth specifications

Our analysis is focused on authorization and authentication. For each of these two problems, we identify key elements within the two specifications (OAuth 1.0 [45] and OAuth 2.0 [46]) that account for their security. We focus on three OAuth protocol flows: OAuth 1.0 (Figure 3.1), OAuth 2.0 implicit grant (Figure 3.2a) and authorization code grant (Figure 3.2b).

Note that, as a prerequisite of any OAuth protocol flow, the relying party must obtain an ID and a secret from the service provider. This is typically done by registering the relying party application through the service provider.



(a) Implicit grant

(b) Authorization code grant

Figure 3.2: Two grant types of OAuth 2.0

Authorization

Authorization is a process that enables an end-user to grant a relying party access to his/her protected resource stored on a service provider. The security audience for authorization is the service provider. That is, the user's sensitive information is located on the service provider, and the service provider must verify that the protected resource is sent to the same party that the user had granted access to.

Although the descriptions of the three protocol flows in the OAuth specifications are fairly complicated, we believe that each has a few key elements for secure authorization, as we identify below.

OAuth 1.0 – The OAuth 1.0 specification requires every token request and protected

resource request to be signed by the relying party using the secret obtained during the application registration stage. The security for authorization is achieved between Steps 8 and 9 of the protocol (see Figure 3.1) when the service provider verifies the signature of the protected resource request. Assuming that the relying party secret is only known to the relying party and the service provider, this step ensures that the receiver of the protected resource is the same party that the user granted the request to.

OAuth 2.0 implicit grant – The OAuth 2.0 framework requires the relying party to provide a redirection URI when registering itself to the service provider. This redirection URI is an essential element for achieving security for authorization using the implicit grant. In Step 2 of the implicit grant (see Figure 3.2a), the redirection URI provided by the relying party is checked against the registered redirection URI in the service provider's database. If the two URIs match, it means that the user is granting access to the same relying party that the access token is sent to.

OAuth 2.0 authorization code grant – The authorization code grant augments the implicit grant by adding an additional step that allows a service provider to verify the identity of a relying party. Security for authorization is achieved in between Step 4 and Step 5 of the protocol (see Figure 3.2b). At this point, the service provider authenticates the relying party using its application ID and secret provided in Step 4 (recall that this ID-secret pair is obtained by the relying party during the application registration stage). After authenticating the relying party, the service provider must verify that this relying party is the same as the one that the user had granted access to in Step 2 of the protocol. This step ensures that the protected resource is sent to the correct relying party.

Authentication

Although many websites and applications use OAuth for authentication, these use-cases are unspecified by both OAuth standards (1.0 and 2.0). This section provides our insights on the protocol details that are important to achieve secure authentication.

Authentication is a process where an end-user signs onto a relying party account by proving to the relying party that he is a certain user on the service provider. Unlike authorization, the security audience for authentication is the relying party. That is, the protected resource is located on the relying party (i.e., the user's relying party account), not the service provider.

To leverage OAuth for authentication, a relying party uses the OAuth authorization flow to request the user's account ID from the service provider. Once this account ID is retrieved from the service provider, it can be used to identify the user on the relying party. However, because the security goals of authentication are very different from those of authorization, not all OAuth protocol flows are secure for authentication.

In general, in order to determine whether an authorization protocol flow can be used for authentication, there are two properties that the relying party must ensure. First, the relying party must ensure that the user ID received from the service provider cannot be tampered with by the user. Otherwise, an adversary can impersonate arbitrary users. Second, the relying party must check that OAuth tokens used to retrieve the user ID is granted to the same relying party. If this check is not done, an adversary could use tokens issued to a malicious application to sign onto users' benign relying party account.

We now examine the three canonical OAuth protocol flows and analyze whether they can be used for authentication.

OAuth 1.0 – There are two aspects of the OAuth 1.0 protocol that make it secure for authentication. First, the user ID exchange in Step 8 and 9 of the protocol (see Figure 3.1)

are done using server-to-server API calls. These calls cannot be tampered by the user. Second, the signature check in Step 8 ensures that the access token used to retrieve the user ID is granted to the same relying party. That is, the user is using this access token to sign onto the same relying party.

OAuth 2.0 implicit grant – Unfortunately, the implicit grant is insecure for authentication. Since access tokens in OAuth 2.0 are no longer bound to relying parties (i.e., anyone with a valid access token can use it to exchange for the user ID), it is impossible to verify whether the user is using an access token to sign onto the same relying party. A malicious relying party could obtain access tokens from users signing onto itself, then use these access tokens to log into a benign relying party, impersonating these users.

OAuth 2.0 authorization code grant – We mentioned previously that the reason why the implicit grant cannot be used for authentication is because OAuth 2.0 access tokens are not bound to relying parties they are issued to. This problem is mitigated in the authorization code grant using an additional parameter called the *authorization code*. A service provider implementing the authorization code grant does not send access tokens to relying parties using browser redirection. Instead, access tokens are delivered using server-to-server API calls (Steps 6 and 7 of Figure 3.2b), which cannot be tampered by a malicious user. Access tokens are exchanged using authorization codes (Steps 4 and 5 of Figure 3.2b), where each authorization code is bound to the relying party it was issued to. By verifying the redirection URI associated with the authorization code in Step 4 of the protocol, the service provider can make sure that the access token is always sent to the relying party that the user tries to authenticate to.

3.2.3 Our Study

In order to understand how real-world developers interpret and implement OAuth, we conducted a comprehensive study on 149 popular mobile applications. These applications included 133 Android applications from the following Google Play store categories: top 300 free applications in all categories, top 200 free applications in social and top 100 free applications in communication. In addition, we manually selected 16 popular iOS and Android OAuth applications (e.g., Quora and Weibo) that were not included in the top charts. 25 (16.8%) applications used in our study were service providers, 126 (84.5%) were relying parties and 2 (1.3%) were both service providers and relying parties. Furthermore, 52 (41.3%) of the relying parties were using OAuth for authorization, the remaining 74 (58.7%) were using it for authentication. Our study revealed that 59.7% of these protocol implementations were faulty and vulnerable to attacks. These results confirm our suspicion that, for a large population of developers, how to use OAuth securely on mobile applications is indeed unclear. We now explain a set of representative cases among the vulnerable applications that we studied.

Storing relying party secrets locally

One common issue was that many developers fail to understand the purpose of the relying party secret, and thus store it locally in the client-side. We believe that the terminology of OAuth confuses developers significantly – the relying party secret is referred to as the "consumer secret" by OAuth 1.0 and "client secret" by OAuth 2.0, where the terms consumer and client are used by each specification to describe the relying party. These names are extremely misleading for developers who have never studied the specifications. For application developers, it is very reasonable to believe that the term consumer or client is referred to the user. Hence, many relying parties believed that it was safe to

bundle their secrets with their mobile applications (or in client-side JavaScript files of their web applications).

Two notable developers making this mistake were Pinterest and Quora. Both Pinterest and Quora used Twitter as a service provider for authentication, and both of them bundled their relying party secrets with their mobile applications. To make the matter worse, after obtaining their secrets using simple reverse engineering, we discovered that the same secrets were used for the Pinterest and the Quora web applications. Since Twitter mainly used OAuth 1.0, this gave an adversary the ability to generate arbitrary OAuth request tokens for Quora and Pinterest. Using these tokens, we (acting as the attacker) could direct users (both web and mobile users) to a legitimate Twitter page with the dialogue box shown in Figure 3.3. Once the user clicks authorize, Twitter's access token will be sent to the attacker, giving him/her full access to the user's account.

We have reported this problem to Pinterest and Quora, both of them have acknowledged this issue and revoked their application secret. Quora also took the extra step of disabling their Twitter authentication mechanism completely for its Android application.

Using authorization flows for authentication

Another common confusion amongst mobile developers is treating authorization and authentication as the same problem. Vulnerabilities due to this issue were described in the literature [52, 43]. As we mentioned in Section 3.2.2, the use-case of authentication is not considered in both the OAuth 1.0 and the OAuth 2.0 specifications. However, the terms "authenticate" and "authentication" are still used frequently in both specifications. For example, Section 3 of OAuth 1.0 [45] and Section 2.3 of OAuth 2.0 [46] describe a method called "client authentication". In actuality, these sections describe the method for which the relying party proves its identity to the service provider using its application

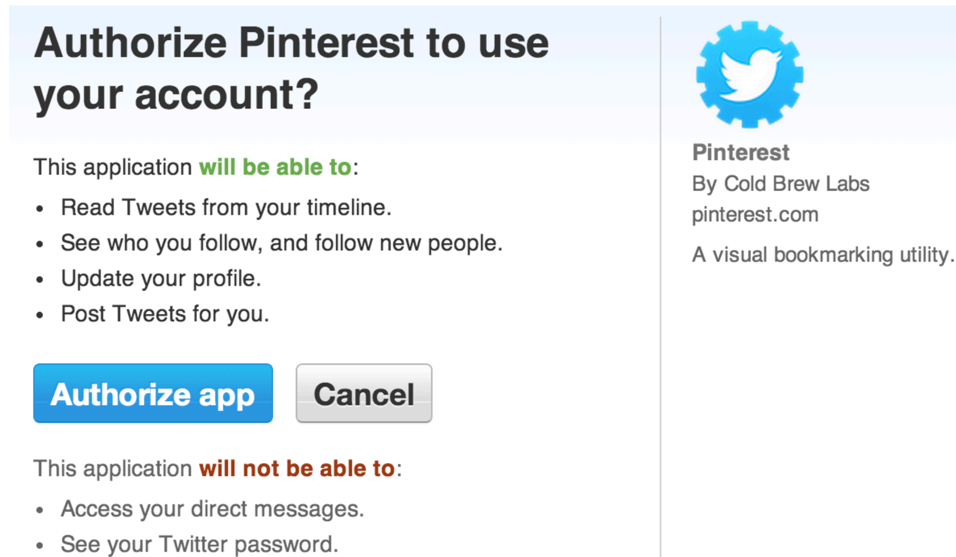


Figure 3.3: Pinterest's OAuth dialogue forged using a stolen application secret.

ID and secret, not the method to identify the user. However, it is easy to see that without contexts, this can lead to developer confusions. Many relying parties using various service providers fall victim to this misunderstanding. In this section, we primarily focus on Facebook's relying parties, but the same concept can be applied to others as well.

Facebook advocates its mobile relying parties to use the implicit grant to access core APIs. The implicit grant is a simpler method for authorization, and in a way provides better immunity against developer mistakes since it does not rely on the secure storage of relying party secrets (since this secret is not used in the implicit grant). However, as mentioned previously in Section 3.2.2, the implicit grant is not safe to use for authentication. This is because the access token used in the implicit grant is not bound to its intended relying party. Hence, an adversary could use a user's access token issued to the malicious application to login as the user for the benign relying party's application.

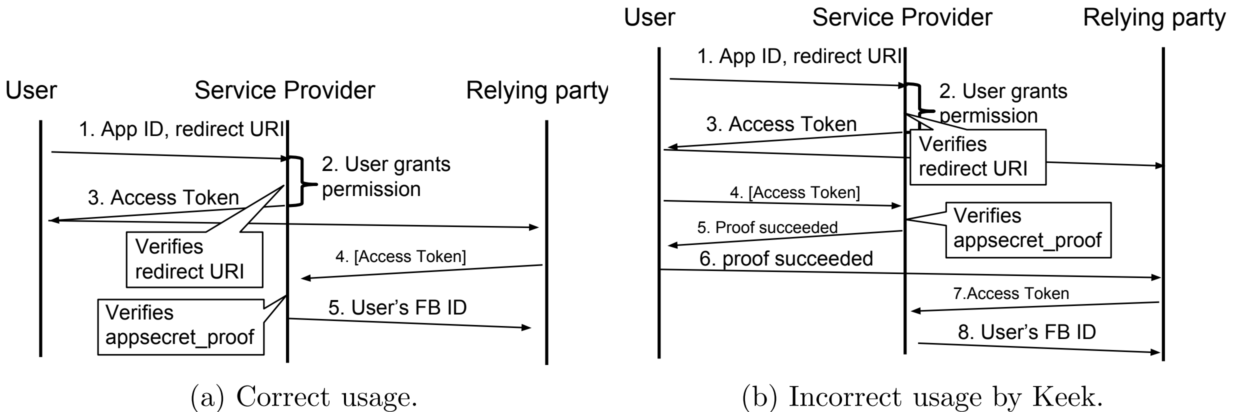
Facebook realized this issue, and added an additional step to enhance its implicit grant for authentication, called the *appsecret_proof*. This step essentially transformed

the implicit grant into a hybrid between itself and the OAuth 1.0 flow. This security-enhanced implicit grant is shown in Figure 3.4a. The key difference between the security-enhanced implicit grant and the regular OAuth 2.0 implicit grant is the addition of an appsecret_proof parameter. This parameter is a cryptographic hash of the access token using the relying party secret as its key that is included with every Facebook API call. To provide security for authentication, Facebook verifies during each API call (Step 4 of Figure 3.4a) that the principal that provides the appsecret_proof is the same principal that the access token was issued to (in Step 2). Unfortunately, because the use-case of authentication is not well understood by many mobile developers, the security-enhanced implicit grant is seldom used in practice by Facebook's relying parties.

We now present our observations on the different Facebook protocol flows real-world mobile developer were using for authentication. Our study included 72 relying parties in total, all of which were using Facebook for authentication.

Usages of the regular implicit grant. We observed a large number (61 out of 72, or 84.7%) of the relying parties using the standard implicit grant for authentication. All of these applications were vulnerable to the attack previously described in Section 3.2.2 against the implicit grant. Some notable applications included Avast Mobile Security & Antivirus and Instagram. We have reported our findings to all the vulnerable applications. So far 22 (36%) of them have acknowledged our reports, of which 11 have fixed their issues by switching to the security enhanced version of the implicit grant. Facebook rewarded us with a $5000 bounty for the vulnerability we discovered in Instagram.

Correct usages of appsecret_proof. Of the 11 relying parties that used the security enhanced implicit grant (i.e., using appsecret_proof) for authentication, 10 had the correct implementation (e.g., following the protocol flow described in Figure 3.4a). Examples of these applications include Hulu and airbnb.



(a) Correct usage. (b) Incorrect usage by Keek.

Figure 3.4: Usages of the Facebook appsecret_proof flow. Parameters inside square brackets are cryptographically hashed using the relying party secret.

Incorrect usage of appsecret_proof. Interestingly, we discovered a different interpretation of the appsecret_proof flow by the mobile application Keek (a social video application with more than 60 million users), illustrated in Figure 3.4b. In Keek's flow, the appsecret_proof was sent from the user's mobile device (i.e., Keek's mobile application) to Facebook's server. When this proof was accepted by Facebook and returned to Keek's mobile application, Keek informed its server that the appsecret_proof was accepted by Facebook. At this point, Keek fell back to the standard implicit grant, and chose to complete the user ID exchange using the standard implicit grant. Unfortunately, the issue with this flow is that a malicious user launching an attack against the implicit grant could forge Facebook's response in Step 6 of Figure 3.4b (since the attacker is also the user), and convince Keek that Facebook accepted the appsecret_proof. This would negate the purpose of the appsecret_proof check and downgrade the protocol into an implicit grant. We have reported this problem to Keek, but they have not responded. Our contact in Facebook offered to follow up with all vendors who are subject to this issue, including Keek.

Although we focused on Facebook's relying parties in this section, the problem is

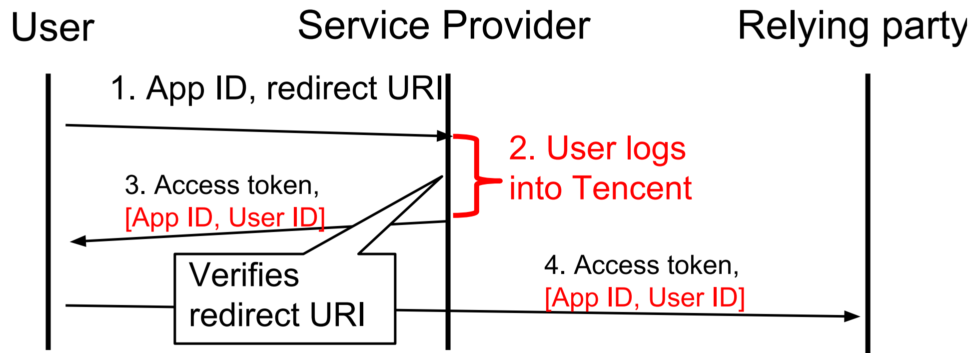


Figure 3.5: Tencent's enhanced implicit grant for authentication. The variables inside the square brackets are cryptographically hashed using a secret key known to only Tencent.

universal. For example, 71.4% of Google relying parties also had the same misconception, and were using authorization flows for authentication.

Inventing home-brewed protocol flows

Since the OAuth specification does not specify the use-case of authentication, instead of leveraging existing authorization flows, several service providers have decided to come up with their own "OAuth-based" protocol flows. In this section, we study one of these home-brewed OAuth protocols and demonstrate the difficulties in designing a completely secure authentication flow.

Tencent is a popular Chinese OAuth service provider that owns Tencent Weibo (a micro-blogging platform with 825 million users [53]) and QQ (an instant messaging application with 798 million active accounts [54]). Tencent claimed to provide authentication using the OAuth 2.0 implicit grant. Upon investigation, we discovered that the implicit grant used by Tencent is actually a modified version of the OAuth 2.0 implicit grant. We illustrate Tencent's implicit grant in Figure 3.5 and analyze it below.

Tencent's developers seemed to understand that because access tokens in OAuth 2.0 are not bound to their relying parties, the standard implicit grant is inherently insecure

for authentication. In order to make the implicit grant safe for authentication, Tencent added a new ID hash parameter into the protocol flow. This ID hash is a concatenated string of the relying party's application ID and the user's Tencent ID, cryptographically hashed using a secret key that is only known to Tencent. For authentication purposes, instead of using the access token to exchange for the user's Tencent ID, relying parties can simply use this ID hash directly *as* the user ID. Since the value of the ID hash is different for each application, an adversary cannot utilize a user's ID hash generated for one application to sign onto the user's account for another application.

Another unique attribute of Tencent's OAuth flow is how Tencent authenticates the user in Step 2 of the protocol. In the canonical OAuth 2.0 implicit grant, this step is defined as follows [46]:

> The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

For most service providers, this step involves the user to first sign onto the service provider using her log-in credentials, then manually click through a permission dialogue box with the relying party's name and the permission scope (i.e., the type of protected resource the relying party wishes to access). Only then can the protocol transaction proceed. However, Tencent interpreted this step differently: it issued the user's ID hash to the relying party immediately after authenticating the user (without prompting the user with an additional permission dialogue box). This step seemed innocuous at first – since if the user voluntarily decided to log into the relying party application by entering her Tencent credentials, an additional permission dialogue might seem unnecessary. Unfortunately, when the protocol transaction is performed inside an embedded browser (i.e., WebView [55, 56]), the following attack is feasible:

1. A user signs onto a malicious application using Tencent in a WebView. However, the adversary supplies Tencent with the application ID and redirection URI of a benign relying party.

2. The user authenticates with Tencent, thinking that she is signing onto the malicious application. Unfortunately, unbeknownst to the user, Tencent treats this authentication request as one that comes from the benign relying party. Before proceeding, Tencent verifies that the redirection URI supplied in Step 1 matches with the registered URI for the application specified by the app ID in Step 1 (both of which are the correct information of the benign relying party provided by the attacker).

3. Tencent redirects the user to the redirection URI that belongs to the benign relying party. This redirection request includes the user's ID hash for the benign relying party.

4. At this point, the malicious application can directly read the user's ID hash associated with the benign relying party from its WebView using the getURL() function in Android and currentWebView.request.URL in iOS.

After the attacker retrieves the user's ID hash for the benign relying party, she can use this to sign into the benign relying party's application as the user.

It is important to note that this attack was enabled by two implementation details that were not well-defined in the OAuth specifications. First, Tencent used the same service provider website for both web and mobile OAuth flows. This forced its mobile relying party applications into using WebView for authentication. Second, because Tencent's user authentication step did not include a permission dialogue box, users could not determine the identity of the relying party application they were signing onto.

After we reported our findings, Tencent immediately acknowledged this issue and patched their user authentication mechanism by adding an additional permission dialogue box.

Chapter 4

Securing collaborating applications via Certification of Symbolic Transactions

The work in this chapter was done in collaboration with Shuo Chen, Shaz Qadeer, and Rui Wang

We believe that the prevalence of security flaws in multiparty online services calls for rigorous engineering supported by formal program verification. However, despite being advocated by researchers for years, program verification is rarely put in actual engineering practice. This chapter presents an approach that significantly lowers the hurdles for real developers to build provably secure multiparty online services.

Protocol-independent security goal. The first hurdle in verification is to understand what security goal to verify. This is hard for developers because protocol documentation and API specifications are, for practical considerations/limitations, often informal, jargon-laden and not comprehensive. It is unclear what exactly each party is supposed to achieve. For example, an Amazon payment protocol was not explicit about whether the payees identity was ensured by the cashier or the merchant [7]; many Facebook's relying party websites did not know which piece of Facebook data should be obtained to securely

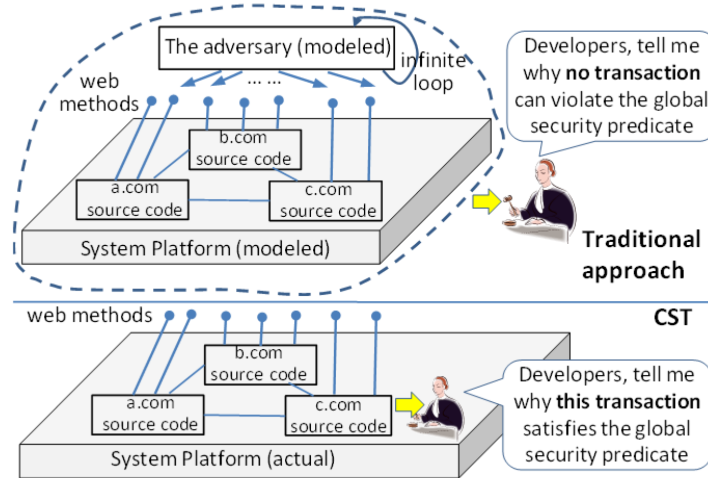


Figure 4.1: Traditional approach versus CST.

authenticate a user [43].

We realize that security is much more intuitive to be defined as a global property (rather than per-party properties) independent of specific protocols. For example, no matter what a payment protocol instructs a merchant and a cashier to do individually, the two parties should jointly ensure an intuitive global property: for any order to be checked out from the merchant, it must have been paid for on the cashier. This property applies to all payment protocols. We will show concretely that such protocol-independent properties can be defined for various scenarios, which guard an implementation against logic flaws in the protocol and developers misunderstandings of the protocol.

Certified symbolic transaction (CST). Once a developer has the property, the next hurdle is how to verify it. Traditional approaches perform verifications offline (see Figure 4.1 (upper)) and face two challenges: (1) the adversary and the system platform need to be modeled precisely (over- or under-permissiveness will lead to false positives or false negatives). The modeled adversary should be able to make arbitrary calls to the web methods on all the services, with all possible argument values that the adversary is able

to create or obtain. Meanwhile, the system platform should be modeled to constrain the adversarys arbitrariness. The modeling demands considerable insight about the system and the property to be verified. (2) The verifier faces a difficult proof obligation: the security predicate must not fail even though the adversary can make an unbounded number of arbitrary calls with arbitrary argument values. The proof requires inductive reasoning about an unbounded number of possible executions. In general, a developer may need to help the verifier by supplying loop invariants and auxiliary lemmas to establish a valid proof. This is especially hard for real-world source code.

Our technique is called *Certified Symbolic Transaction (CST)*. It achieves the same verification goal in a different a manner, shown in Figure 1 (lower). It treats every protocol execution (referred to as a *multiparty transaction* from now on) as *a runtime process for creating a proof obligation* for a static program verifier, which we call *the certifier*. The certifier logically examines whether the sequence of computations on all parties *in this transaction* collectively ensures the global predicate. Compared to fully static approaches, the CST approach demands much less from developers: (1) No modeling is needed for the adversary or the system platform, because every verified transaction is created dynamically at runtime. (2) Since transactions that fail to be certified are rejected, developers are only required to make sure that every intended transaction is logically sound. This requires the developer to think only about the expected rather than the unexpected scenarios.

An interesting novelty of the CST approach is the use of static program verification at runtime, i.e., to symbolically verify transactions on-the-fly. In Section 2, we motivate why static verification is essential for representing and certifying multiparty transactions. Although program verification is prohibitively expensive in general, CST combines static verification and caching to ensure near-zero amortized overhead.

Real-world demonstration. We demonstrate that it is practical to apply CST to real-world systems. We have secured five commercially deployed systems using CST by adding only tens (or 100+) lines of code per party. These systems are based on real-world frameworks, such as PayPal and Amazon Payments services, Microsoft LiveID single-sign-on SDK, the OpenID framework in DotNetOpenAuth, the OAuth template in Visual Studio ASP.NET MVC 4, and NopCommerce software. We also challenged the CST approach by implementing a gambling system integrating four services, for which there is no existing protocol to follow. We have evaluated CST along four dimensions:

- **Security** – We analyzed 14 cases of real vulnerabilities reported in the literature. CST will foil the exploits or avoid the vulnerabilities in 12 out of the 14 cases, the remaining two being out-of-scope issues.

- **Protocol-independence** – We show that different protocols, e.g., Amazon Simple Pay vs. PayPal Standard for payment and OpenID vs. OAuth vs. Live Connect for single-sign-on, can be held to the same global predicates. More interestingly, we show several implementations that blatantly violate the OAuth 2.0 protocol, but satisfy the end-to-end global predicate for single-sign-on. They all turn out to be as secure as the protocol-conformant ones.

- **Performance** – Because symbolic transactions are cacheable, CST incurs only near-zero amortized runtime overhead. Therefore, CST is suitable for real-world deployed services.

- **Programming effort** – We report the line of code that we added or changed for every open-source package.

4.1 Overview of CST

The CST approach is a unique combination of concrete runtime execution with symbolic program verification: it verifies each multiparty transaction at runtime but attempts to verify it not just on the actual inputs seen in that transaction but for all possible inputs. To understand this aspect of the design of CST, we first need to recognize two basic characteristics about typical multiparty services:

1. **There is no global data storage** – Different parties do not fully trust each other despite their cooperation on specific transactions. Consequently, each party holds its own data structures (e.g., databases) locally without a globally-shared storage. For example, PayPal's payment record database is not exposed to any merchant website, because it contains transactions related to other merchants. Similarly, the database of Facebook's single-sign-on service is not publicly shared with every relying party, because it contains many secret strings and IDs of users of other relying parties.

2. **Security is a global property** – As discussed in the introduction, security is a property across different parties. For example, "secure checkout" requires that an order on the merchant website has a corresponding payment record on the PayPal server. For this reason, the security predicate to be examined by CST needs to refer to data structures of different parties. We call it an ambient predicate. Figure 4.2 shows a simplified system that preserves the essence of a realistic system. Data structures `A[]`, `B` and `C` are defined in the source code of three different companies, respectively. A multiparty application attempts to maintain the ambient predicate $(C == true) \iff (\exists i.A[i] == B)$ that refers to data structures at every party. Obviously, the ambient predicate cannot be concretely checked, because `A[]` and `B` are not shared with `c.com`. An alternative approach and the core idea of CST is

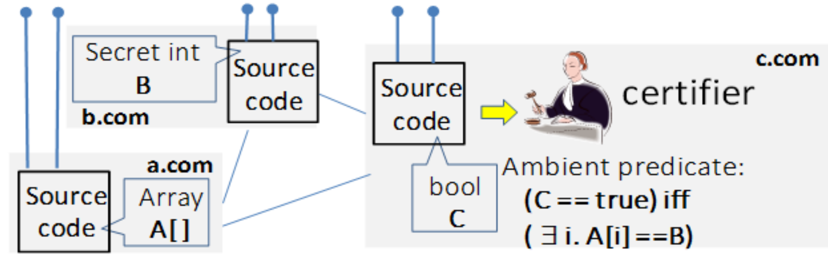


Figure 4.2: A simple multiparty system.

to check it symbolically, i.e., to examine whether the sequence of computations of a multiparty transaction logically implies the ambient predicate.

CST achieves this symbolic verification using a message field called SymT (i.e., Symbolic Transaction), attached to every message to collect the source code executed on each party. In general, disclosing source code to another party may lead to intellectual property infringement or reverse-engineering. For CST, this is not a serious concern, because the disclosed source code only consists of a few methods that implements the protocol that every party has agreed on.

When a transaction completes, the certifier uses the final SymT value to synthesize a program representing the executed source code of the entire multiparty transaction. Symbolic verification then checks that the program satisfies the ambient predicate. The collection of the source code and the synthesis of the program to be verified is elaborated in Sections 3 and 4. The complexity of verifying a symbolic transaction depends on the expressiveness of the ambient predicate and the program fragments executed by the different parties. We found first-order logic to be adequate for the services we studied and use an off-the-shelf automated program verifier based on satisfiability- modulo-theories [21] for the verification.

In general, symbolic verification is expensive. However, for CST, it incurs an extremely low amortized cost, even lower than what a concrete checking would incur (which would

need network messages). The certifier achieves this efficiency by caching the theorem proved by it about a symbolic transaction. Since the theorem holds for all inputs, a future identical symbolic transaction is deemed convincing immediately regard-less of the data values on which it computes. If the source code is unchanged, this caching results in near-zero amortized runtime overhead. Furthermore, the caching is over all transactions generated by all users. Most likely, developers themselves are the users who trigger the verification, and real users enjoy the caching.

4.1.1 Threat model

We assume that the attacker has browsers and his own servers, but does not control the servers of non-attacker parties. Developers of non-attacker parties are cooperative, and do not lie about the executed source code. The network traffic is protected by HTTPS, so the attacker cannot read or tamper with data in transit. General programming bugs such as buffer overrun, cross-site scripting and cross-site request forgery, are orthogonal to the type of logic flaws that CST targets. Many techniques have been proposed and deployed to address these issues; these techniques can be used in conjunction with CST.

4.2 An illustrative Example of CST

We now give an example about secure checkout to explain the CST approach. We first define the ambient predicate for the secure checkout problem. Next, we illustrate a real-world vulnerability that, when exploited, violates the ambient predicate. Finally, we show how CST would have caught the error at runtime.

The basic steps in every checkout transaction are as follows: 1) place an order on the merchant site; 2) make a payment on the cashier site; 3) complete the order on the merchant site. We refer to the sequence of these steps at runtime as a multiparty

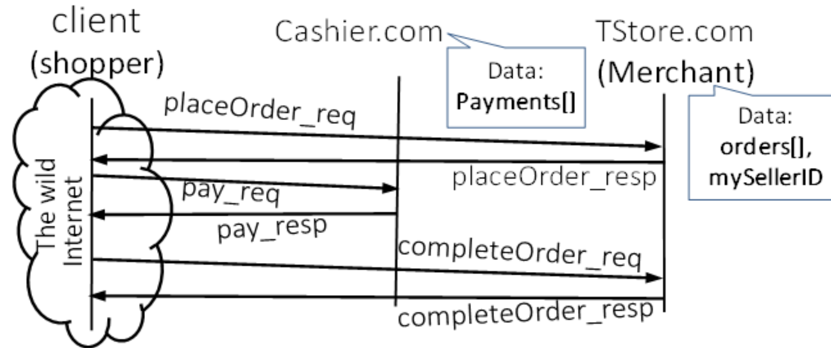


Figure 4.3: The basic messages and data for checkout.

transaction.

Figure 4.3 shows the three parties in the transaction the client and two servers. We assume that Cashier.com is the cashier site, and TStore.com is the merchant site. The client is a greedy shopper who wants to check out without making a full payment or any payment at all. As the adversary, the clients behavior is arbitrary. Essentially, the "client" can be thought of as the wild Internet that can send arbitrary HTTP requests in any arbitrary order, even those not conforming to Figure4.3. A secure implementation must guard against such a malicious client.

There are three web methods, `placeOrder`, `pay`, and `completeOrder`. Each is invoked by an HTTP request, and returns an HTTP response. The request and response for `placeOrder` are denoted `placeOrder_req` and `placeOrder_resp`, respectively. Request and response for other methods are named similarly.

Data structures. Every transaction involves the data structures on the two servers: `orders[]` is an array to store all orders, indexed by the identifier of each order (i.e., `orderID`); `mySellerID` is the merchant's identifier registered on the cashier; `payments[]` is the payment records on the cashier. A real implementation may use database tables instead of arrays. Section 4.4.3 will explain how we convert database operations into array

accesses by defining “stub methods”.

Security predicate. The security predicate is defined over a fixed multiparty transaction and refers to the fields of request and response of invoked methods and the data structures on the servers. The predicate given below defines secure checkout (the line numbers are added for easy reference).

```
placeOrder_req.orderID == completeOrder_req.orderID &&                1
∃ i. (                                                                2
    Cashier.payments[i].status == "Paid" &&                           3
    Cashier.payments[i].total                                         4
              ==TStore.orders[placeOrder_req.orderID].gross &&        5
    Cashier.payments[i].payee == TStore.mySellerID &&                 6
      Cashier.payments[i].orderID == placeOrder_req.orderID)          7
```

This predicate holds for a particular transaction iff there is a payment record at the cashier for the item being bought. Note that this predicate is stated with respect to our problem definition and is not specific to any protocol for establishing it.

4.2.1 A traditional implementation

The predicate above specifies the security objective. However, it is not locally checkable because it is about data relations across different parties. For example, `payments[]` is the cashiers data structure, while `orders[]` is the merchant's. Therefore, it is an ambient predicate. Protocol specifications today do not explicitly define their ambient predicates. Instead, a protocol simply instructs each party how to check a set of locally checkable predicates and respond to other parties. It is hoped that security is achieved as a result of all these local checks. As mentioned in the introduction, this is a fallacy in reality.

Figure 4.4 shows a simplified example of a traditional implementation. It defines the data structures explained earlier, and implements `placeOrder()`, `completeOrder()` and `pay()` to handle `TStore.com/placeOrder.aspx`, `TStore.com/completeOrder.aspx` and `Cashier.com/pay.aspx`. Lets assume the client checks out a $35 order with orderID 123.

```
class Merchant {
    order_record_t[] orders;
    string mySellerID = "JohnSmith1234";
    PlaceOrderResp_PayReq placeOrder(PlaceOrderReq req){
        PlaceOrderResp_PayReq resp;
        int orderID = req.orderID;    resp.orderID = orderID;
        orders[orderID].status = "Pending";
        resp.redirectionURL = "https://Cashier.com/pay.aspx";
        resp.total = orders[orderID].gross;
        resp.returnURL = "https://TStore.com/completeOrder.aspx";
        sign(resp); return resp;
    }
    public bool completeOrder(PayResp_CompleteOrderReq req){
        if (VerifySignature(req)==false) return null;
        if (req.signer != "Cashier.com" || req.status != "Paid" ||
          orders[req.orderID].status != "Pending") return false;
L1:     orders[req.orderID].status = "Complete"; return true;
    }
}
```

```
class Cashier {
    payment_record_t[] payments;
    PayResp_CompleteOrderReq pay(PlaceOrderResp_PayReq req){
        if (VerifySignature(req)==false) return null;
        i=getAvailableIndex();
        payments[i].payee = req.signer;
        payments[i].orderID = req.orderID;
        payments[i].total = req.total;
        PayResp_CompleteOrderReq resp;
        resp.redirectionURL = req.returnURL;
        resp.orderID = req.orderID;    resp.status = "Paid";
        sign(resp);    return resp;
    }
}
```

Figure 4.4: A traditional implementation.

In a non-malicious scenario, the messages are as follows (readers can walk through the code in Figure 4.4 to understand how the messages are generated). For brevity, every message is represented by enclosing data fields in angle brackets after the message name, e.g., the first message stands for TStore.com/placeOrder.aspx?orderID=123.

1. placeOrder_req:

   placeOrder_req<orderID=123>

2. placeOrder_resp and pay_req (a browser redirection):

   pay_req<orderID=123,total=35,returnURL=https://TStore.com/completeOrder.aspx,signature=[TStore'ssignatureforthewholerequest]>

3. pay_resp and completeOrder_req (a browser redirection):

   completeOrder_req<orderID=123,status=Paid,signature=[Cashier's signature for the whole request]>

Assuming that signing and signature checking are done correctly, readers can confirm that the message sequence above can drive the code to Line L1, where the order is marked Complete. However, there is a problem: the ambient predicate we care about is nowhere to be found in Figure 4.4. The developers' hope is that the local checks in these methods have collectively ensured "security". Is it really so?

A real-world vulnerability. In fact, this example is based on the real Amazon Simple Pay payment method. An exploitable logic flaw was detailed in Section III.A.2 of reference [7]. In the exploit, the attacker has his own seller account Mark and server MarkStore.com, and is able to purchase from the victim TStore by only paying to MarkStore. Specifically, when he receives placeOrder_resp from TStore, he discards the signature and re-signs it as MarkStore. This message is sent to the cashier (Amazon) as pay_req. From

the cashier's point of view, it would seem as if the attacker was purchasing an order from MarkStore, so Mark gets paid. However, the redirectURL points to TStore.com, so TStore receives the completeOrder_req signed by the cashier. TStore does not expect the cashier to notify it about an irrelevant payment (i.e., a payment made to Mark), and is fooled to complete the order. NopCommerce, a popular e-commerce software, is subject to this flaw.

4.2.2 The CST-enhanced implementation

CST enhances the implementation by requiring a SymT field in each message, which contains SHA-1 hash values of the source code of invoked methods. For example, the source code hash of `placeOrder` is `f8f8bd5b0fe4711a09731f08c06c3749d240580c`. For readability of this chapter, we show a hash value as a hash symbol "#" with a method name, e.g., #`placeOrder`, but a real SymT does not contain "#" or method names.

SymT is now attached to every message shown earlier (parentheses and colons to be explained in Section 4.3.1, and ϵ to represent an empty string):

1. placeOrder_req:

   placeOrder_req<orderID=123,**SymT=**ϵ>

2. placeOrder_resp and pay_req (a browser redirection):

   pay_req<orderID=123, total=35, returnURL=https://TStore.com/completeOrder, **SymT=TStore.com::#placeOrder()**,signature=[TStore's signature for the whole request]>

3. pay_resp and completeOrder_req (a browser redirection):

   completeOrder_req<orderID=123,status=Paid,**SymT= Cashier.com::**

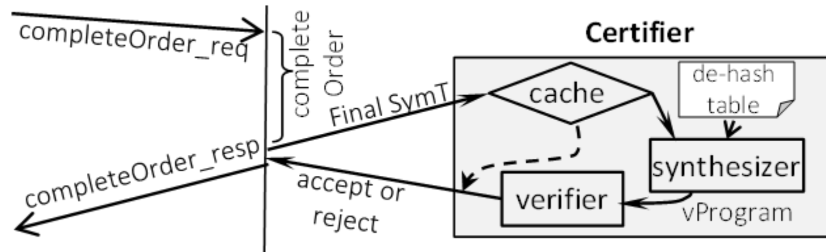


Figure 4.5: The certifier module on TStore.com.

#pay(TStore.com::#placeOrder()), `signature=[Cashier's signature for the whole request]>`

At the end, `completeOrder()` is invoked by the request `completeOrder_req` (see Figure 4.5), in which `SymT` is `TStore.com::#completeOrder(Cashier.com::#pay(TStore.com ::#placeOrder()))`. This is the *final SymT*, which is given to the certifier (which runs on *TStore.com*). For now, let's ignore the cache in the figure. When the synthesizer gets the final SymT, it synthesizes a program, namely *vProgram*, which is formally verified against the ambient predicate. If the verification succeeds, the transaction is approved (e.g., the order is marked *Complete*). Otherwise, it is rejected. The synthesis of the program requires the certifier to recover source code texts from their hash values. This capability relies on a "de-hash" table, which contains hash-to-source-code mappings. Anyone can submit a source code text to the de-hash table so that its hash value will be computed and associated with it.

A preview of security. The synthesized vProgram for the SymT from the previous paragraph, "`TStore.com::#completeOrder(Cashier.com::#pay(TStore.com::#place Order()))`", has the following steps. First, it invokes the method placeOrder with an arbitrary input value. Next, it invokes the method pay with the return value of placeOrder as the argument. Finally, it invokes the method completeOrder with the return value of placeOrder as the argument. Referring to Figure 4.4, the reader can see that the ambient

predicate is satisfied at the end of this sequence of execution.

On the other hand, if the logic flaw explained in Section 4.2.1 is exploited, the final SymT will be `TStore.com ::#completeOrder(Cashier.com::#pay(MarkStore.com::#placeOrder()))`. The difference between this SymT and that for the correct transaction is only that the call to method `placeOrder` is performed at *MarkStore.com* rather than *TStore.com*. In the next section, we will explain that the program synthesis is done with respect to a set of trusted parties, which in this case comprises only *TStore.com* and *Cashier.com*. Because the first step (i.e. `placeOrder`) is made by an untrusted party, the synthesized vProgram will ignore this step, and begin directly with the method pay with an arbitrary value as argument. Consequently, the certification fails and the transaction is rejected. Note that *MarkStore* can even hide its presence by providing `SymT=`ϵ in `placeOrder_resp`. If so, the final SymT will be `TStore.com::#completeOrder(Cashier.com::#pay())`. All our discussion is still valid in this case.

4.3 The CST Certifier

In this section, we describe the design and implementation of the certifier that validates a symbolic transaction in the CST approach. In particular, we elaborate the synthesizer, the verifier and the cache. Overall, the certifier is a method with three arguments and a Boolean return value:

```
bool certify (
    string FinalSymT,
    string AmbientPredicate,
    string[] TrustedParties
)
```

The arguments `FinalSymT` and `AmbientPredicate` are self-explanatory. The argument `TrustedParties` is an array to specify which parties are considered trusted for this ambient predicate. For the example discussed earlier, the certifier (on *TStore.com*) only

needs to trust *TStore.com* and *Cashier.com*, i.e., the validity of the ambient predicate should not depend on any other party. Similarly, in the single-sign-on scenario, the certifier on the relying party *foo.com* should only trust *foo.com* and the identity provider (e.g., *facebook.com*), but no one else. As stated earlier, the client (browser) is always an untrusted party, involved in all transactions. `TrustedParties` decides which computation steps the certifier should take into account. Computations performed on other parties, including the client, are ignored in the synthesized program.

4.3.1 Symbolic Transaction

The symbolic transaction, SymT, is the basis of the CST approach. A symbolic transaction makes a multiparty transaction, hitherto only an informal notion in the mind of a protocol designer, explicitly represented in protocol messages.

SymT needs to record not only the sequence of method calls, but also how two consecutive calls are stitched, i.e., how the output of a call (referred to as `method1` on *a.com*) is fed into the input of the next call (referred to as `method2` on *b.com*). Specifically, the main question is why *b.com* should believe that the input of `method2` indeed comes from *a.com*. There are only two possible reasons: 1) the input is signed by *a.com*; 2) *b.com* itself makes a direct server-to-server call to method1 to obtain the input for `method2`.

Therefore, SymT must precisely encode the stitching scenarios. Figure 4.6 shows three SymT values, in which we highlight certain symbols for discussion. In scenario A, the output of `method1` is not signed (denoted by the highlighted single-colon), and is supplied to `method2` by an unnamed party (denoted by the highlighted parentheses). An unsigned browser redirection is a example of scenario A. The only difference in scenario B is that the input of `method2` (i.e., the output of method1) is protected by *a.com*'s signature, so `b.com` is confident that it is generated by `a.com`, untampered. The signing

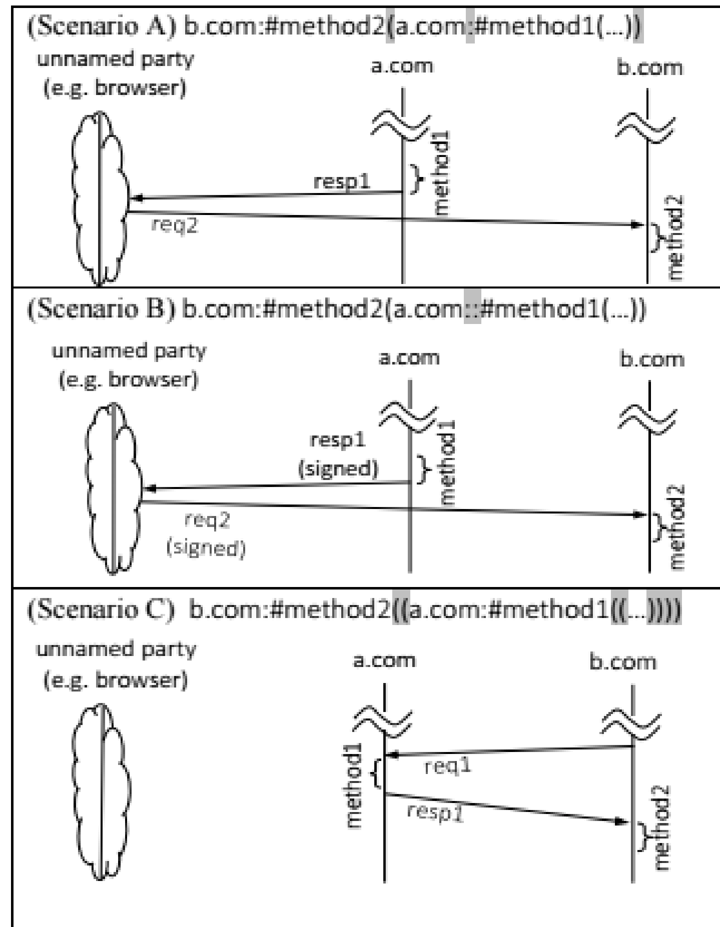


Figure 4.6: Stitching two method calls.

is denoted by the double-colon "::". In scenario C, `method1` is called from *b.com* using a direct server-to-server call (e.g., a SOAP or REST API call), so b.com of course has the confidence that the input of `method2` comes from a.com. The server-to-server call is denoted by two pairs of double-parentheses "(())". Another valid SymT, not shown in Figure 4.6, could be `b.com:#method2((a.com::#method1(())))`, representing a server-to-server call returning a signed response. It is equivalent to scenario C, as the signing is unnecessary.

Signature checking. CST relies on the underlying messaging system to guarantee

that all signatures are correctly checked at runtime. If a signed message carries the SymT `a.com::#method1()`, the receiver must ensure that the signature is generated by a.com. We also note that: 1) CST incurs no additional signing operation, i.e., any unsigned message in the original implementation will remain unsigned in the CST-enhanced implementation; 2) For any signed message, *the SymT field itself must be covered by the signature.*

Grammar. In general, SymT is a nested sequence of method calls specified by the production rules in Listing 4.1. The rules we use in our implementation are more comprehensive; they also accommodate multiple arguments in a method call and a message with selective fields covered by a signature.

Listing 4.1: Production rules for SymT

```
SymT → ϵ
     | PARTY–ID : METHOD–CALL
     | PARTY–ID :: METHOD–CALL
METHOD–CALL → SRC–HASH ( SymT )
     | SRC–HASH (( SymT ))
PARTY–ID → a.com | b.com | amazon.com |
```

Semantics. When SymT a.com:fa(b.com:fb(c.com:)) is attached to message M, it represents the following recursive claim about the message (we add brackets to indicate the scoping):

```
The message is M, which a.com claims is
{
    the result of executing f_a() on input M_a, which b.com claims is
    {
        the result of executing f_b() on input M_b, which c.com claims is
        {. . .}
    }
}
```

If the above SymT contains double-colons and double-parentheses, the meaning of the claim will not be changed. However, they will affect how the synthesizer trusts each layer of the claim, which will be explained next in Section 4.3.2.

It is worth emphasizing that SymT *must not* be interpreted as follows:

```
a.com claims that {
    the message is M,
        which is the result of a.com executing f_a() on input M_a,
            which is the result of b.com executing f_b() on input M_b,
                which is the result of c.com executing
}
```

This interpretation is wrong because `a.com` is not able to make a claim about the whole sequence of calls. The correct interpretation is a recursive claim, in which each party only makes a claim about one call.

4.3.2 Synthesizer

Think of SymT as an onion potentially rotten inside – each layer is a claim, which, if untrusted, implies that everything inside is bogus. Thus, the synthesizer needs to identify the outermost layer where the trust cannot be established, and discard it with everything inside. Specifically, the synthesizer examines the SymT string from left to right. It looks for the first call which is: 1) performed at an untrusted party (i.e., `PARTY-ID` $\notin$ `TrustedParties`), or 2) not tamper-proof (i.e., when the pattern `‘‘(PARTY-ID:METHOD-CALL)’’` is matched, such as in Scenario A).

If such a call is found, it is discarded and replaced with the empty string ϵ. The resulting SymT, basically a hollow onion, is trusted. The vProgram can be directly generated from it without any further considerations regarding trust. It takes arbitrary input values.

Figure 4.7 shows the synthesized program corresponding the SymT `‘‘TStore.com::#completeOrder(Cashier.com::#pay(TStore.com::#placeOrder()))’’`. The method to be verified is the static method main(). The local variables of this method, such as placeOrder_req, pay_req, etc., and the global objects, such as TStore and MyCashier, are initialized with non-deterministic values; this initialization is not shown in the figure.

```
class vProgram {
    static Merchant TStore=new Merchant();
    static Cashier MyCashier=new Cashier();
    static void main() {

/* The program for a normal transaction will contain L1 and L2.
The program for the attack described in Section 3.1 will not contain L1
and L2. */

L1:  placeOrder_resp=TStore.placeOrder(placeOrder_req);
L2:  pay_req = placeOrder_resp;
L3:  pay_resp = MyCashier.pay(pay_req);
L4:  completeOrder_req = pay_resp;
L5:  bool completeOrder_resp=
                         TStore.completeOrder(completeOrder_req);
L6:  if (!completeOrder_resp) return;
L7:  Contract.Assert( placeOrder_req.orderID ==
                                   completeOrder_req.orderID);
L8:  Contract.Assert(
        Contract.Exists(0,MyCashier.payments.Length, i =>
              MyCashier.payments[i].status == "Paid" &&
              MyCashier.payments[i].total ==
                      TStore.orders[completeOrder_req.orderID].gross &&
              MyCashier.payments[i].payee ==
                                          TStore.mySellerID      &&
              MyCashier.payments[i].orderID ==
                                          completeOrder_req.orderID
));
       }
}
```

Figure 4.7: Program synthesis from SymT.

Since all method calls happen at trusted parties and no method call is an instance of Scenario A, lines L1-L6 compose all the method calls. Lines L7 and L8 are reached only if the order is completed. These lines assert that the ambient predicate holds on the preceding computation. On the other hand, the SymT for the attack from Section 4.2.1 is "`TStore.com::#completeOrder(Cashier.com::#pay(MarkStore.com::#placeOrder()))`". In this SymT, the call `MarkStore.com::#placeOrder()` is at an untrusted party and is therefore replaced by ϵ, causing lines L1 and L2 to be dropped, so the assertions will fail. Similarly, if the first step was `TStore.com:#placeOrder()`, which has a single-colon and is not tamper-proof, it would result in the same vProgram without lines L1 and L2. Figure 4.8 illustrates the synthesis steps graphically.

4.3.3 Verifier

We demonstrated CST on systems implemented using ASP.NET and C#. The focus on .NET is only because we want to use an off-the-shelf program verifier for .NET. The CST technique is equally applicable to any programming language. Figure 4.9 shows the tool chain. The program generated by the synthesis tool is compiled by the C# compiler of Visual Studio. The output is an executable file of .NET byte code. ByteCodeTranslator (BCT) [57] is a tool to translate a .NET byte code program into a Boogie program. Boogie is an intermediate verification language [58]. We use the Corral system [59] as the verifier. In addition to the input Boogie program, the Corral verifier expects a non-negative number to establish a bound for the unfolding of loops and recursion in the program. Corral outputs exactly one of three results: the program is verified, or the program is verified with respect to the bound, or the program is falsified. In the final case, Corral also presents a counterexample witnessing the error in the program. Our certifier certifies a transaction *only if Corral returns the first output, i.e., the transaction*

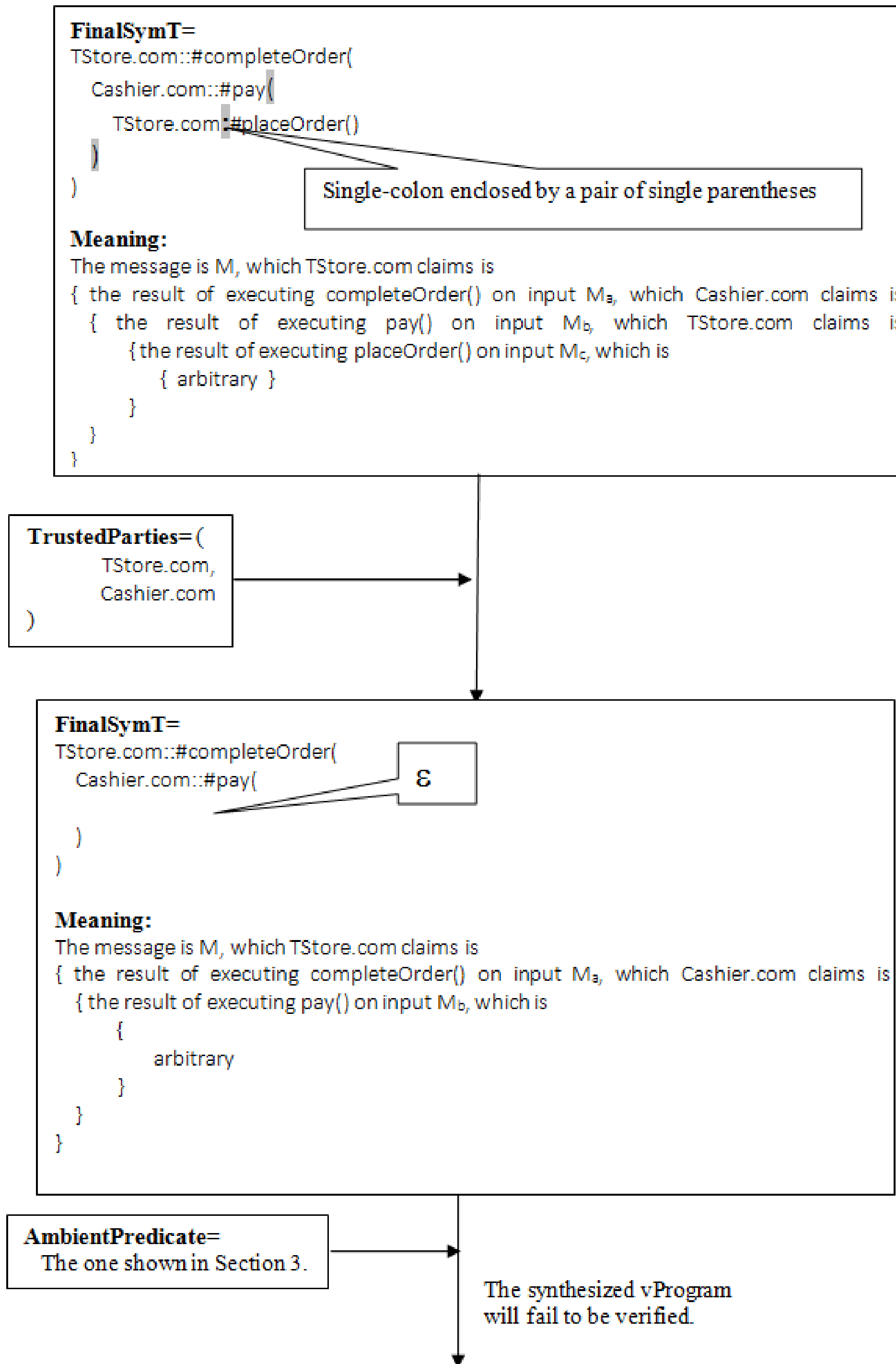


Figure 4.8: An example of the synthesizer discarding an untrusted call. (Note that the single colon with the placeOrder call)

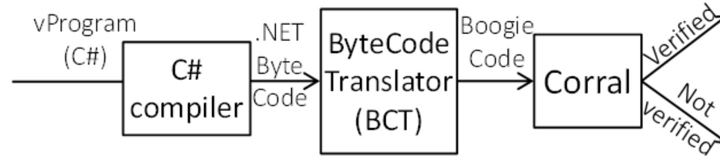


Figure 4.9: Our tool chain.

is verified without a bound.

4.3.4 Cache

Program verification is expensive (e.g., 10 - 30 seconds to verify a transaction in our cases). It is impractical to do it for every transaction. Caching is therefore essential in CST: when the verification is done, the certifier caches the result (i.e., true/false) with the triple `FinalSymT`, `AmbientPredicate` and `TrustedParties`. Any future call to the certifier by any user will return the result directly if it matches a previously cached triple.

4.4 Applying CST in the Real World

We have applied CST to enhance various systems that serve practical purposes. Unlike proof-of-concept prototypes, these systems contain realistic source code and data structures written by actual developers. We view it as an accomplishment that all our enhanced systems are ready for commercial deployments. For example, people can install our CST-enhanced NopCommerce to run their stores: a customer can choose items, check out orders, and specify shipping and payment methods, etc; payments are made on the real Amazon and PayPal servers. People can also use our Live Connect SDK to enable single-sign-on on their websites. Functionality-wise and performance-wise, our systems are indistinguishable from the original ones, except that the implementations are provably secure. These systems are all publicly accessible. Their URLs, source code, instructions

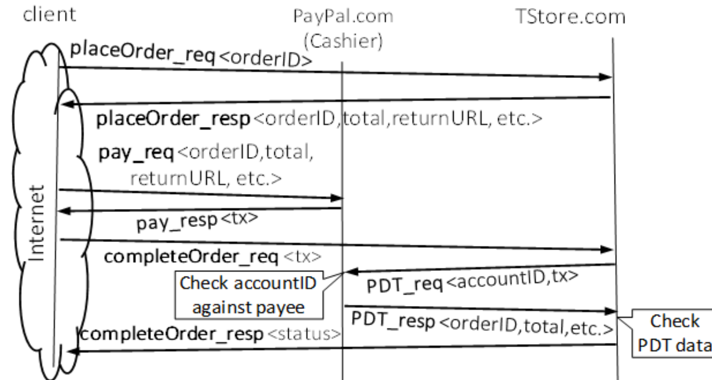


Figure 4.10: PayPal Standard.

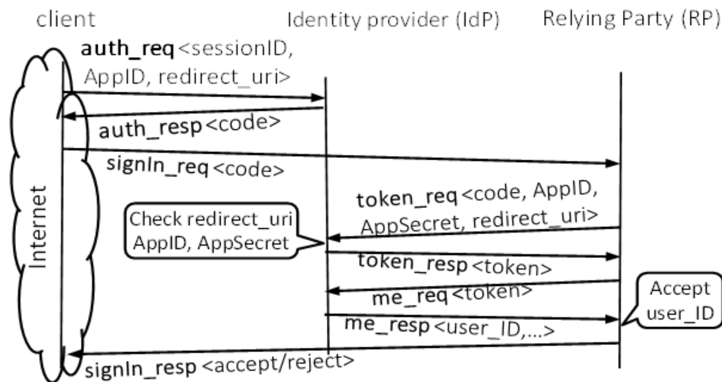


Figure 4.11: SSO based on OAuth-2.0.

and videos are given in `https://sites.google.com/site/symbolictransaction/` [60].

4.4.1 Categories of enhanced systems

We have worked on three categories of systems so far:

Payment/checkout. NopCommerce [61] is a widely used open-source e-commerce application. It was one of the focused systems in previous security studies [7, 62]. NopCommerce accepts many third-party payments. We decided to enhance its payment modules for Amazon Simple Pay and PayPal Standard. The former is essentially what we described in Section 4.2.1, and latter is shown in Figure 4.10. They are significantly different in that Amazon Simple Pay is based on signed redirection messages, whereas

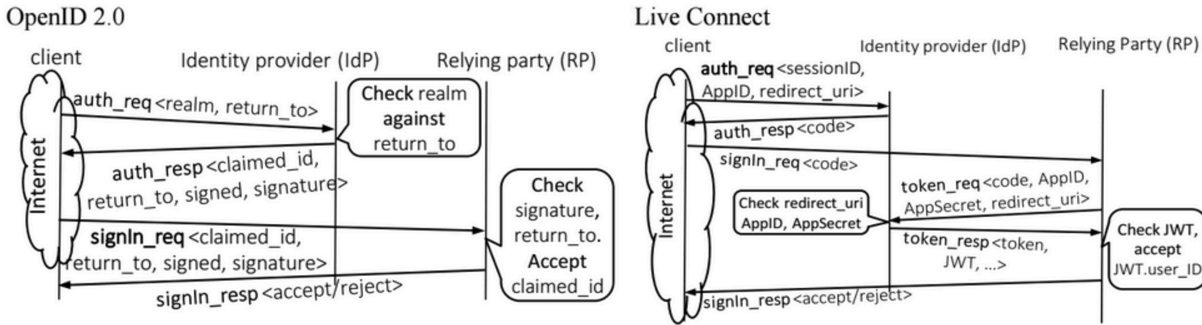


Figure 4.12: SSO based on OAuth-2.0.

the PayPal Standard mechanism relies on a direct server-to-server call, namely the PDT (Payment Data Transfer) query, for securely communicating the payment details.

Single-sign-on (SSO). We worked on the implementations of three different SSO protocols: 1) the OpenID-2.0-based SSO [63] in the DotNetOpenAuth framework [64]; 2) the OAuth-2.0-based SSO [65] in Microsoft Visual Studio ASP.NET MVC 4 web application template that uses Facebooks OAuth service [66]; 3) Live Connect SDK [67], which heavily influenced the OpenID Connect specification [68]. (Note that the terminology may cause a little confusion. OpenID Connect is a protocol, drafted by the OpenID Foundation, to use OAuth 2.0 for SSO. It was published very recently. Live Connect SDK predates the OpenID Connect specification, so the SDK refers to its SSO mechanism as OAuth 2.0, rather than OpenID Connect.) The message diagram of the OAuth-2.0-based SSO is shown in Figure 4.11. The ones for OpenID 2.0 and Live Connect are given in Figure 4.12.

Gambling. People are familiar with the above two categories, because standards organizations and major companies have provided protocol specifications or API documentations. We decided to use CST to build a gambling system. The goal is two-fold: 1) we do not have any existing gambling protocol to conform to, so building this system is an end-to-end exercise of the protocol-independent thinking process; 2) previous scenarios

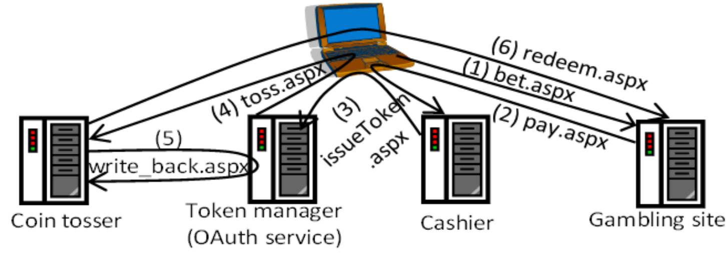


Figure 4.13: The gambling system.

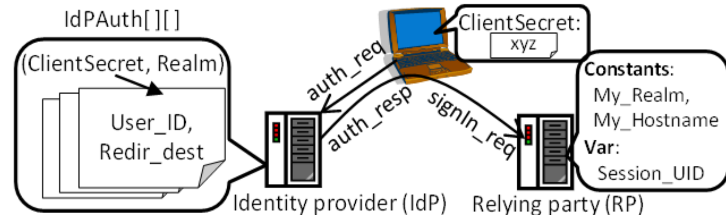


Figure 4.14: A protocol-independent definition of single-sign-on.

only involve two trusted services. We want to challenge the CST approach by involving more parties. The gambling system we built consists of four independent services for betting, payment, authorization and coin-tossing (see Figure 4.13).

4.4.2 Ambient Predicates

Despite the significant differences among these systems and their adopted protocols, we specify the same ambient predicate for each category.

Payment/checkout. Our enhanced implementations for Amazon Simple Pay and PayPal Standard ensure the same ambient predicate as we presented in the example in Section 4.2. It is to ensure that when the merchant is about to check out an order, there exists a payment record in the cashier that matches this order.

SSO. Figure 4.14 gives our protocol-independent definition of the SSO security goal. In every SSO system, there are a client, an identity provider (IdP) and a relying party (RP). The client holds a piece of `BrowserSecret`, which is shared with the identity

provider, but not the relying party. The relying party has at least two constants: `My_Realm` is its identifier known to the identity provider; `My_Hostname` is its network-addressable name. Variable `Session_UID` is the session variable to be set upon a successful sign-on.

An SSO transaction starts with a request from the client to the identity provider, namely `auth_req`, containing the `BrowserSecret` and the `Realm` of the relying party that the client wants to sign in. The identity provider then retrieves an object called `ID_Claim` using the pair (`BrowserSecret,Realm`). `ID_Claim` contains at least two fields: `User_ID` is the identifier of the user that this claim is about; `Redir_dest` indicates the destination of the redirection message (i.e., `auth_resp` followed by `signIn_req` in the figure). The retrieval is based on a two-key dictionary called `IdpAuth`, which is defined as follows in C#:

```
Dictionary<string, Dictionary<string, ID_Claim>> IdpAuth;
```

Note that how `IdpAuth` entries are established is not what SSO concerns about. The identity provider can identify the client as "Alice" for any reason (e.g., through password or SSL client certificate), thus create an entry. An SSO protocol is to prove to the relying party the existence of the entry, i.e., the fact that the identity provider somehow believes the client is Alice. A transaction must satisfy the ambient predicate:

```
1 IdPAuth[auth_req.BrowserSecret][My_Realm].Redir_dest == My_Hostname &&
2 IdPAuth[auth_req.BrowserSecret][My_Realm].User_ID == Session_UID
```

The first clause asserts that the identity provider passes the `ID_Claim` to this relying party, not to any other website (which could then use the `ID_Claim` to sign into this relying party illegally). The second clause asserts that the user ID to be associated with the session is the one in the aforementioned `ID_Claim`.

Gambling. The ambient predicate for the gambling system is given below. Lines (3) - (6) ensures that a proper payment has been made for the bet (identified by final_req.betID);

Lines (8) - (12) ensures that the bet is valid and matches the tossing result of the coin-tosser. The validity of the predicate depends on the computations on all four services.

```
1  GamblingSite.bets[final_req.betID].status=="Pending" &&
2  ∃ i. (
3      Cashier.payments[i].total ==
4          GamblingSite.bets[final_req.betID].amount  &&
5      Cashier.payments[i].orderID == final_req.betID &&
6      Cashier.payments[i].payee==GamblingSite.MySellerID) &&
7  ∃ x. (
8      TokenMgr.records[x].payee==GamblingSite.MySellerID &&
9      TokenMgr.records[x].betID == final_req.betID  &&
10     TokenMgr.records[x].EffectiveResult != ``untossed'' &&
11     GamblingSite.bets[final_req.betID].guess ==
12         TokenMgr.records[x].EffectiveResult)
```

As mentioned earlier, a motivation for building this gambling system is to challenge the CST approach with substantial complexity. In this case, the final SymT of a normal transaction contains 4 parties and 7 hash values:

```
GamblingSite.com:#redeem(CoinTosser.com::#post_toss((TokenMgr.com:#
write_back((CoinTosser.com:#toss(TokenMgr.com::#issueToken(amazon.com::#pay
(GamblingSite.com::#bet()))))))))
```

The synthesized vProgram has more than 300 lines of C# code, which the certifier is able to verify.

4.4.3 Programming

Every verification technology applied to real-world systems needs the effort of abstraction. The abstraction is often done through *factoring* and *stubbing*. This subsection explains what they mean in the CST programming. For concreteness, the description incorporates our experience of enhancing the Live Connect SDK, although the general ideas apply to our experiences of enhancing other systems.

Factoring. CST requires the core computations to be factored out in order to be logically verified. Typical non-core computations include methods for parsing, composing and HTTP-encoding/decoding for messages. These non-core computations contain

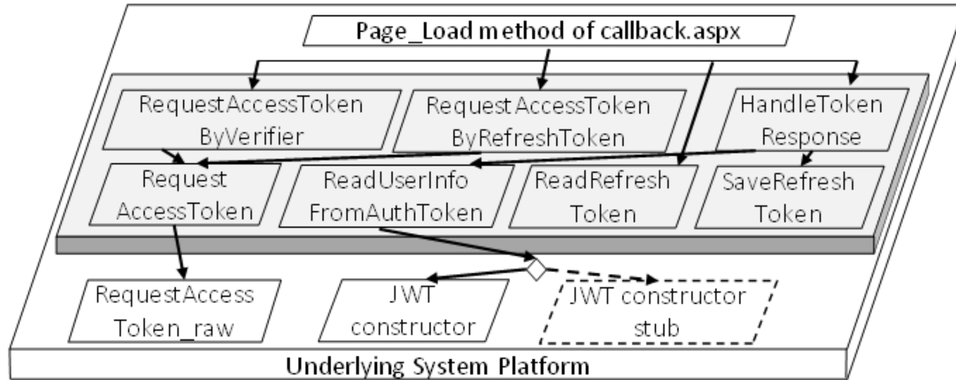


Figure 4.15: Factoring and stubbing in callback.aspx.

complicated string (byte-array) operations. Currently Corral and Boogie have only limited capability for reasoning about string operations: assignment, equality comparison and string-indexed dictionary are supported, but concatenation, tokenization, character operations, etc. are not.

Figure 4.15 shows the call-graph of `callback.aspx` in the Live Connect SDK, which handles a redirection from the LiveID server. In this 3-dimensional drawing, the methods are placed on an unshaded level and a shaded level. The shaded level consists of the core methods that we factor out. The unshaded level serves as the interface between the core logic and the underlying platform. For example, `Page_Load` parses HTTP arguments. `RequestAccessToken_raw` is a method we create so that the string operation for constructing an HTTP request can be separate from the core method `RequestAccessToken`. The constructor of class `JWT` (i.e., `JsonWebToken`) performs Base64 decoding and signature validation, which are byte-array operations.

It is not a requirement that all complicated string (byte-array) operations are moved out from the core methods. For example, a core method can still construct a string for the logging/debugging purpose, as long as it does not affect the validity of the ambient predicate. Our experience on existing implementations of real-world service frameworks

is that they are already architected similarly to Figure 4.15 so that lower-level methods parse HTTP requests into well-structured objects and assemble HTTP responses using these objects, while upper-level modules implement core computations on these objects. The core computations usually deal with basic types (e.g., integers and Booleans), structs and arrays of basic types, as well as string assignments and equality comparisons. Corral/Boogie can effectively reason about all these programming constructs.

Stubbing. The core methods call many other methods, which will not be included in the vProgram for verification. In other words, these methods are treated as unimplemented, from the certifier's standpoint. For Corral/Boogie, the default semantics of an unimplemented method is that it returns a non-deterministic value, but does not modify any program state (i.e., the body is a no-op). This works in most cases, because most of these methods are not essential to the verification. However, there are a few situations in which the semantics of these methods matter, so developers need to define their semantics as stubs. In the Live Connect SDK, we provided a stub method as shown in Figure 4.15. The source code is below.

```
static JsonWebToken JWT_Constructor_stub(OAuthToken token){
    JsonWebToken jwt;
    havoc(jwt);  //to assign jwt a non-deterministic value
    Contract.Assume(jwt == token.jwt && jwt != null);
    return jwt;
}
```

The reason to provide `JWT_Constructor_stub` is to replace the Base64 decoding and signature validation operations in the JWT constructor with the logic most essential to the verification. Specifically, the logic is that a new JWT object equals to the jwt member of the input argument `token`, i.e., `jwt!=null && jwt== token.jwt`.

Another situation for providing a stub is to model a database operation. In real-world systems, persistent data (e.g., the payment records) are often stored in and queried from

a database by INSERT and SELECT. Corral/Boogie does not have built-in support for these operations. Developers need to wrap these operations in C# methods, and define stubs that are logically equivalent to database operations but use C# data structures like array, set, list, etc.

Mapping from variable names in ambient predicate to those in implementation. Every verification technique needs to map variable names in specification to that in source code, and so does CST. The ambient predicates defined above use generic names for message fields and variables. Unfortunately, they are named differently across protocol documentation and implementations. In our programming, we had to adapt the ambient predicates to the terminologies of these implementations. For example, the `BrowserSecret` in our definition is called `MSPAuth` in Live Connect; the `Realm` in our definition is called `AppID` in Facebook OAuth and `openid.realm` in OpenID 2.0.

4.4.4 Deployment

The deployment path we envision is that, first, major service providers, e.g., Facebook and Amazon, attach the SymT field in their messages; then, relying websites gradually opt-in to take advantage of CST. Note that CST has the advantage of incremental deployment: without any modification, a CST-unaware relying website (i.e., a relying party or merchant website) will just work normally with a CST-enhanced service provider, except that it is not provably secure.

Even before the service providers actually deploy the enhancement, CST can be used to secure real-world transactions. This is exactly what we did for all the aforementioned open-source packages (except for DotNetOpenAuth), which do not contain the service provider code. For each of these services, we built a “wrapper service”, which serves as a relay in order to attach the SymT field. An example is shown in Figure 4.16, the

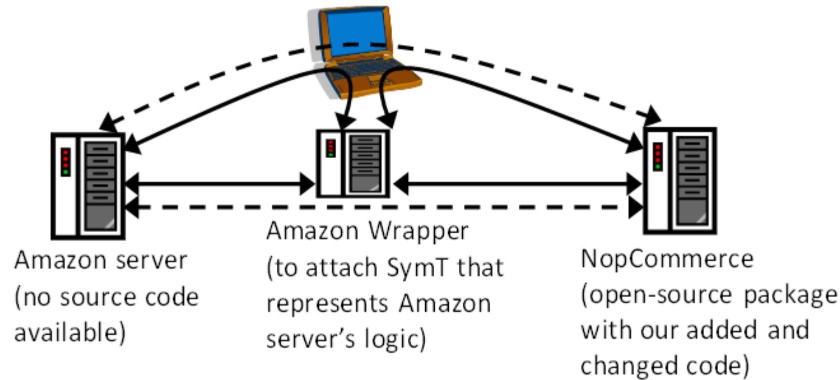


Figure 4.16: An example wrapper service.

dashed-line redirection is implemented as two solid-line ones with the wrapper service as the trampoline. A server-to-server call is similarly implemented. Hence, all transactions certified on our relying websites are processed by the actual service providers for real. Of course, a caveat is that the attached source code hash is only our best effort approximation for the logic behind these web methods.

4.5 Evaluation

We evaluated the CST approach on security, protocol-independence, performance, and programming effort.

4.5.1 Security

We studied 14 real-world vulnerabilities (listed in Table 4.1), which we believe are a representative sample set in this problem space this set includes all the cases reported in the literature [7, 8, 43, 41] that allows an attacker to either check out an order without a proper payment or sign into a victim users account through SSO, excluding the cases due to generic web programming flaws like cross-site scripting, cross-site request forgery (CSRF) and session fixation. We show next that 12 out of the 14 cases would be addressed

No.	Attack	Target System	CST effectiveness
#1	Section III.A.1 of [7]	NopCommerce with PayPal Standard	Yes
#2	Section III.A.2 of [7]	NopCommerce with Amazon Simple Pay	Yes
#3	Section III.B.1 of [7]	Interspire with PayPal Express	Yes
#4	Section III.B.2 of [7]	Interspire with PayPal Standard	Yes
#5	Section III.B.3 of [7]	Interspire with Google Checkout	Yes
#6	Section III.C of [7]	Websites using Amazon Payments	No
#7	Section 4.1 of [8]	Websites using Google ID	Yes
#8	Section 4.2 of [8]	Websites using Facebook Connect	No
#9	Section 4.3 of [8]	Websites using JanRain sign-on	Yes
#10	Section 4.5 of [8]	Websites using Google ID	Yes
#11	Section 2 of [43]	Websites using OAuth implicit flow for SSO	Yes
#12	Section IV.A.1 of [41]	osCommerce, CS-Cart and AbanteCart using PayPal Standard	Yes
#13	Section IV.A.2 of [41]	OpenCart and TomatoCart using PayPal Express	Yes
#14	Section IV.A.3 of [41]	TomatoCart using PayPal Express	Yes

Table 4.1: Real-world cases studied in our security analysis.

by CST. The remaining two are out-of-scope issues.

Cases for which CST is effective. Attacks for cases #1, #2 and #11 can be launched against the systems that we built using CST NopCommerce with PayPal Standard, NopCommerce with Amazon Simple Pay and OAuth-2.0-based SSO. We confirmed that the attacks result in vPrograms not satisfying the ambient predicates.

Case #7 is about a relying party that uses the email address (`email`) field, rather than the `claimed_id` field, as the user's identifier. The reported vulnerability is because the `signIn_req.email` field can be excluded from the signature coverage by the malicious user, so that it bears an arbitrary value supplied by the client. We intentionally introduced this vulnerability to our OpenID 2.0 implementation on DotNetOpenAuth. When the attack is launched, the resulting vProgram fails to verify clause (2) in the ambient predicate, because `Session_UID` is taken from the `signIn_req.email` field, which is non-deterministic in the vProgram.

Cases #3-#5 are about Interspire, which is another merchant software providing similar functionalities as NopCommerce. We have not applied CST on Interspire. However, based on the nature of the attacks, it is clear that they fall nicely into the scope of CST:

- Case #3 is an attack in which the attacker starts two independent transactions one is expensive, the other is cheap. The attacker only performs the PayPal payment step in the cheap transaction, but not in the expensive transaction. At a particular stage, the merchant takes a signed `orderID` as the input argument. It is at this stage where the attacker supplies the signed `orderID` of the expensive transaction into the HTTP session of the cheap transaction, so the expensive order is checked out although only the cheap order is paid. This attack will be defeated by CST, because the `orderID` is always attached with the SymT, and signed together. When the attacker swaps the `orderID` of the expensive transaction into the cheap transaction,

the SymT of the transaction has to be swapped in as well. The SymT clearly indicates that no payment step has been performed, so the ambient predicate will fail to verify.

- Case #4 is a vulnerability because the merchant may take the `orderID` from the client's cookie that is not signed. According to our definition, any unsigned value supplied by the client is non-deterministic. Having a non-deterministic `orderID`, the ambient predicate fails to be verified.

- Case #5 is because the payment total is calculated based on the shopping cart at the checkout time, but the order being checked out is generated based on the shopping cart after the payment is made. The ambient predicate will not verify in this case, because the shopping cart is a runtime object, querying its property at two time points are semantically two method calls, corresponding to two different symbolic values. The equality would not be established in the verification.

Case #9 is about JanRain SSO service. The attack is to set the redirection destination (i.e., `Redir_dest`) to the attackers website when the (victim) user tries to sign into a (victim) website. The JanRain server correctly checks the redirection destination, but the most important step in the attack is that the client can swap in an unchecked URL as the redirection destination after the checking. If CST was applied, the unchecked URL would be an arbitrary value (i.e., the attached SymT would not indicate any logic constraint imposed on this URL), so the clause in the ambient predicate about the redirection destination would fail.

Similar to case #7, case #10 is about a relying party that intends to use the email field as user ID. However, the developer mistakenly uses an arbitrary non-email field as the email field, due to a misunderstanding of the OpenID 2.0 protocol. CST would prevent

the flaw because the `IdPAuth` dictionary on the identity provider would not even contain this arbitrary field.

Cases #12-#14 include every exploitable flaw reported in [41]. In case #12, the attacker replaces the payee account ID with his own PayPal account ID, and checks out an order from the victim store by paying himself. Cases #13 and #14 are similar to case #3, in which the attacker places two orders in two sessions, and supplies a message obtained from the session of the cheaper order into the session of the more expensive order. We have explained that these are precisely the type of logic flaws that CST would prevent.

Cases that are not addressed by CST. CST relies on every party to correctly verify signatures. Case #6 is a vulnerability in signature verification. It is out of scope of CST. The root cause of case #8 is a client-side cross domain issue. Specifically, it is due to a special Adobe Flash communication mode that does not conform to the same-origin policy. This causes secret data from the IdP to be obtained by a malicious webpage on the victim users browser. CST does not address security flaws in the underlying platform.

4.5.2 Protocol independence

The fact that we check the same ambient predicates for systems adopting considerably different protocols shows the protocol independence of their security goals. To make the point even stronger, we built implementations that *blatantly violate protocols but are nevertheless secure*. Three of them are shown in Figure 4.17.

Implementation (A) does not conform to the OAuth 2.0 protocol (shown earlier in Figure 4.11) for two reasons:

1. The protocol requires `token_req` to contain the field `AppSecret`, which is a secret the identity provider assigned to every relying party at the registration time. We realized

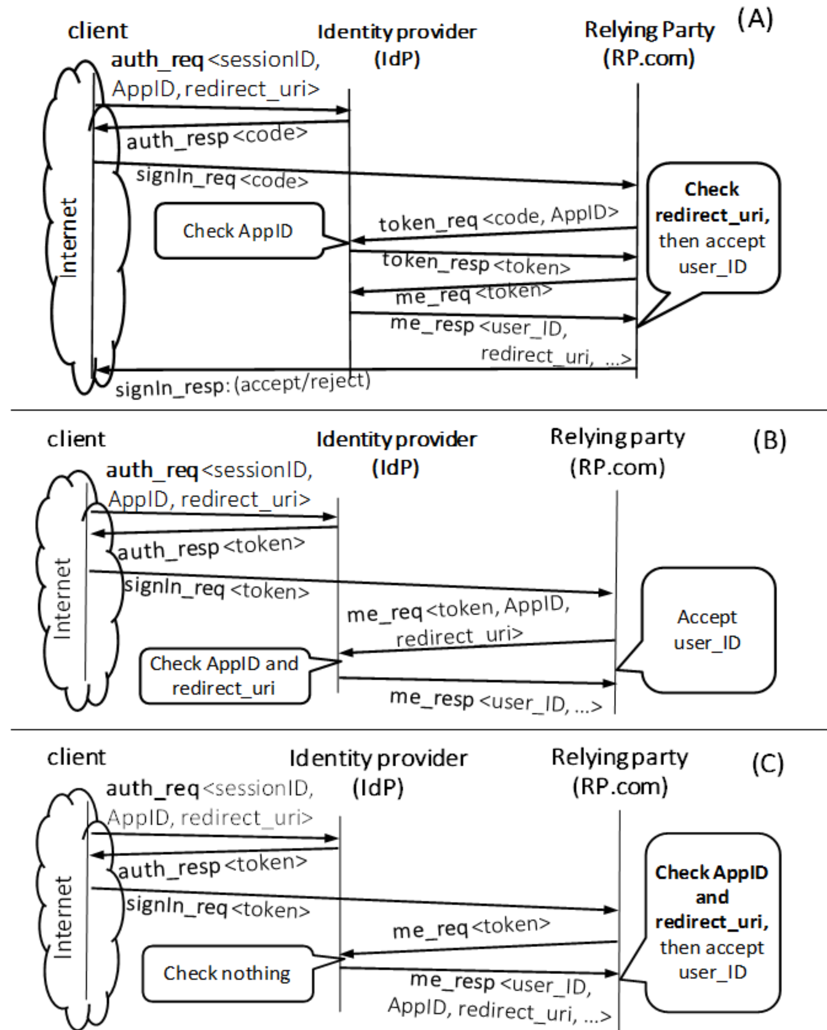


Figure 4.17: Protocol-violating yet secure implementations.

that the purpose of `AppSecret` is to prevent another website from impersonating the relying party to access the identity provider, which seemed unrelated to SSO. We removed `AppSecret`, and the ambient predicate still held, suggesting that `AppSecret` is indeed unnecessary in SSO. Note that we do not claim that `AppSecret` is useless in general in the OAuth protocol (in fact, we now understand precisely where it is useful).

2. The protocol requires the identity provider to check `redirect_uri`. In implementation (A), the identity provider does not perform the check. Instead, it returns `redirect_uri` in me_resp, so that the relying party can check it. The ambient predicate still hold in this case.

Implementation (B) even more blatantly violates the protocol, because it gets rid of code, but uses token for the client to authenticate into RP.com. According to a previous study, using token to authenticate is a pervasive and serious vulnerability [43]. Recently, the OAuth 2.0 specification has been augmented to explicitly forbid this kind of token usage (see section 10.16 of RFC 6749). However, we realized that this usage is vulnerable only because, if the relying party accepts a `token` rather than a code for authentication, the steps `token_req` and `token_resp` will be skipped. The relying party directly calls `me_req:(token)`, so the checking of `redirect_uri` and `AppID` required by the ambient predicate is missing. If `me_req` took additional arguments `redirect_uri` and `AppID`, and the identity provider performed the checking, as in implementation (B), then security would still be achieved.

Suppose the identity provider insists not to check anything, is the implementation doomed flawed? Not necessarily. The checking can be performed by the relying party, like in implementation (C).

Summary. We can see that all these implementations are just different ways of sharing the responsibility of performing all necessary checks. The OAuth 2.0 protocol describes one particular way, but not the only way. Of course, we understand that protocol conformance not only affects security, but also modularity, deployability, interoperability, etc. We do not suggest implementers disregard protocols, but only argue that security can (and should) be ensured independently, because understanding “who should do what, and why” about each protocol specification can be very subtle.

4.5.3 Performance

A significant strength of CST is its near-zero runtime overhead. Table 4.2 provides the measurement results, obtained from a server with a 2.10 GHz CPU and a 3.5 GB RAM, running Windows Server 2008. The numbers fall into two categories: per transaction cost and one-time cost. The time spent on synthesizing and verifying a vProgram belongs to the one-time cost, because the caching amortizes the cost over all transactions on all users. In fact, developers themselves are most likely the users who actually pay for the cost during testing.

Per-transaction cost. For a non-certifier party, the only runtime overhead is to produce the SymT. The source code hash is a pre-computed constant for a given version, so the only overhead is a string concatenation. Also note that *CST incurs no additional signing operation*, i.e., any unsigned message in the original implementation will remain unsigned in the CST-enhanced implementation. For the certifier, the only per-transaction overhead is the cache lookup for the SymT. Obviously, the runtime overheads for both a non-certifier party and the certifier should be extremely small. We nevertheless did the actual measurements to confirm that, for every system we implemented, the per-transaction runtime overhead is too small to report.

The SymT field incurs traffic overhead for protocol messages. We measured the average traffic overhead per SymT field (shown as Bytes/SymT). Our implementations use SHA-1 (160 bits), RSA (384 bits) and UTF-8 for hashing, encryption and encoding.

One-time cost. The synthesis cost is measured for two situations when the de-hash table is stored locally or on another server. The first one mainly indicates the computational time of the synthesis algorithm, which is within 5 milliseconds in each of our case. The second situation may be more beneficial in practice because it offloads the de-hash table to another server. Although the synthesis time is longer, since it is a

	Per-transaction cost		**One-time cost**		
	Runtime overhead	Average traffic overhead	Program synthesis using a local de-hash server	Program synthesis using a remote de-hash server	compilation, byte-code translation and verification
Live Connect SDK	≅0ms	106 B/SymT	3ms	568ms	18758ms
OpenID 2.0 on DotNetOpen-Auth	≅0ms	119 B/SymT	5ms	409ms	15380ms
Facebook SSO using ASP.NET MVC 4	≅0ms	120 B/SymT	5ms	408ms	12090ms
NopCommerce with Amazon Simple Pay	≅0ms	78 B/SymT	2ms	450ms	15444ms
NopCommerce with PayPal Standard	≅0ms	105 B/SymT	8ms	190ms	10990ms
Coin tossing gambling	≅0ms	205 B/SymT	3ms	945ms	32477ms

Table 4.2: Performance overhead per transaction and one-time costs.

one-time cost, it should not be a performance concern in practice.

The last column in Table 4.2 corresponds to the real heavy-lifting step in CST. It consists of C# compilation into .NET byte code, byte-code translation into Boogie code and verification of Boogie code. The time reflects the significant logic complexity for verifying a transaction consisting of realistic methods. In contrast, today, this significant logic reasoning is never conducted, and correctness is taken on faith.

4.5.4 Programming effort

Table 4.3 shows the lines of code (LoC) we added or changed in each open-source project, excluding comment and white lines. The certifier is the same across all projects. It consists of 347 LoC. The LoC numbers in the unshaded cells are a good measurement of the effort for factoring and stubbing. The amount of code is fairly small, under 200 LoC for each party, indicating that the original developers had architected the code well so that it was amenable for the CST enhancement. The shaded cells correspond to our wrapper code for the real API providers, and factoring and stubbing do not apply for them.

4.5.5 Related Work

There is a rich body of literature about verifying security protocols themselves, which we do not discuss here due to the space constraint. Research is also conducted to address issues in protocol implementations. Existing approaches can be categorized as either *top-down* or *bottom-up*. The top-down approaches focus on generating or verifying implementations based on formal specifications of protocols. For example, Bhargavan et al. [69] verified a number of reference implementations of the InfoCard protocol. In their work, the protocol and the security specifications are written in high-level languages F#

	Shared methods	**The relying website**	**The API-provider**
Live Connect SDK	0	48	100 (wrapper)
OpenID 2.0 on DotNetOpenAuth	104	59	182
Facebook SSO using ASP.NET MVC 4	0	119	411 (wrapper)
NopCommerce with Amazon Simple Pay	0	71	375 (wrapper)
NopCommerce with PayPal Standard	0	71	239 (wrapper)

Table 4.3: Lines of code that we added or changed in the open-source packages (comment and white lines excluded)

and WSDL. Bhargavan and Corin et al. [70, 71] developed a compiler that can synthesize a protocol implementation from a high-level F# specification of multiparty transactions. The bottom-up approaches try to extract protocols from actual systems. Aizatulin et al. [72] proposed to use symbolic execution to convert a protocol implementation in C into its high-level model in the applied pi calculus. Bai et al. developed a technique to extract SSO protocols from HTTP messages of network traces [42]. The uniqueness of CST is that it performs static verification at runtime, which converts the harder obligation of verifying a system into that of verifying intended transactions.

Proof carrying code (PCC) [73] is a technology for a code consumer (e.g., an OS kernel) to examine whether the code from an untrusted producer (e.g., a kernel extension from a third-party company) is accompanied by a logic proof of desired safety properties. CST and PCC target different problems. CST does not have the "proof carrying" aspect of PCC, but interestingly has a "code carrying" aspect that enables the verification.

Secure multiparty computation (SMC) [74] is a methodology for multiple parties to perform a joint computation over a set of private variables without revealing these variables to every party. Although SMC can be applied to multi-party protocols, it offers a different set of security guarantees hence cannot replace CST. CST operates under the assumption that protocol implementations can contain implementation errors. On the other hand, an erroneously implemented SMC-compliant protocol has no security guarantees.

Our work has connections with logic-based access control. Research on access control logic focus on expressiveness, decidability and theorem-proving efficiency of different logic frameworks. Lampson et al. defined a decidable logic based on the "*speaks for*" relation [75]. Appel and Felten found that many access control scenarios need higher-order logic, which is more expressive, but usually undecidable. They proposed proof-carrying authentication (PCA) [76], motivated by the idea of PCC, to shift the proof obligation to requestors. Code-carrying authorization (CCA) [77] is a follow-up of PCA. CCA allows requestors to provide fragments of the reference monitors code (in form of the spi calculus), rather than proofs as in PCA. Our work is different from prior work on access control logic in two ways: 1) The certifier in a CST system is not a reference monitor; rather the computation being certified by the certifier is akin to a reference monitor; 2) The notion of proof in a CST system is partitioned into reasoning about trust (in the synthesizer) and logical correctness (in the program verifier), enabling the use of off-the-shelf program verifiers. On the other hand, proof systems for access-control are monolithic and based on custom axioms and inference rules about trust and authority, which makes it difficult to use off-the-shelf verifiers.

Connections can also be drawn between CST and secure multiparty computation [78] and verifiable computation [79] in applied cryptography. However, the goal are very

different from CST. Secure multiparty computation is to enable parties to jointly compute a function over secret data held by individual parties. Verifiable computation enables a weaker device to securely outsource computations to untrusted servers.

Chapter 5

Conclusion

Web developers often make mistakes when coding their applications. We show that not only are these implementation errors are prevalent and difficult to prevent, but they also do considerable harm to the security of underlying web applications. In this thesis, we designed and implemented two practical frameworks that harden the security of web applications under the assumption that they contain implementation errors. These frameworks are App Isolation, and CST.

For App Isolation, we have shown that a single browser can achieve the security benefits of using multiple browsers, by implementing entry-point restriction and state isolation to isolate sensitive apps. These mechanisms might not be appropriate for every web site, but they can be effective for many high-value web sites, such as online banks. Using this approach, these high value web sites can help protect themselves and their users from a broad spectrum of attacks with minimal effort.

For CST, we have shown that it can be practically deployed to guard a wide-range of web protocols against logic flaws. CST represents a paradigm shift for developers. Programming is less about conforming to a protocol, but more about explicating the computations in order to establish an end-to-end global safety property. From the security standpoint, protocols become advisory rather than mandatory. What is truly mandatory

is the ambient predicates independent of these protocols.

The Web is advancing at a rapid pace while new breeds of attacks are being discovered constantly. Therefore, requiring developers to perfectly secure code is proving to be unrealistic and ineffective. Both App Isolation and CST signal for a type of security practice that lessens the burden of security from developers and shift them towards security experts (from browser vendors and protocol working groups).

Bibliography

[1] CGI Security, "The Cross-Site Scripting (XSS) FAQ." http://www.cgisecurity.com/xss-faq.html.

[2] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 1193–1204, ACM, 2013.

[3] C. Jackson, "Improving browser security policies." PhD thesis, Stanford University, 2009.

[4] C. Reis, B. Bershad, S. D. Gribble, and H. M. Levy, "Using processes to improve the reliability of browser-based applications," in *In Under submission*.

[5] A. Barth, C. Jackson, and C. Reis, "The Security Architecture of the Chromium Browser." 2008 Technical Report.

[6] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 892–903, ACM, 2014.

[7] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online–security analysis of cashier-as-a-service based web stores," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 465–480, IEEE, 2011.

[8] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 365–379, IEEE, 2012.

[9] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 413–428, USENIX, 2012.

[10] R. Cook, "The Next Big Browser Exploit," *CSO Magazine*, p. 15, Feb 2008.

[11] E. Iverson, "Two Web Browsers can be More Secure than One."

[12] Mozilla Foundation Security Advisory 2009-29, "Arbitrary code execution using event listeners."

[13] Mozilla, "Test Pilot." `https://testpilot.mozillalabs.com/`.

[14] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a Formal Foundation of Web Security," *Computer Security Foundations Symposium, Proceedings of, IEEE*, vol. 0, pp. 290–304, 2010.

[15] Mozilla, "Prism." `http://prism.mozillalabs.com/`.

[16] T. Ditchendorf, "Fluid." `http://fluidapp.com/`.

[17] M. Silbey and P. Brundrett, "Understanding and working in Protected Mode Internet Explorer," 2006. `http://msdn.microsoft.com/en-us/library/bb250462.aspx`.

[18] C. Grier, S. Tang, and S. T. King, "Secure Web Browsing with the OP Web Browser," in *IEEE Symposium on Security and Privacy*, pp. 402–416, 2008.

[19] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The Multi-Principal OS Construction of the Gazelle Web Browser," in *USENIX Security Symposium*, pp. 417–432, 2009.

[20] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson, "Protecting Browsers from Cross-Origin CSS Attacks," in *ACM Conference on Computer and Communications Security)*, 2010.

[21] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A Safety-Oriented Platform for Web Applications," in *IEEE Symposium on Security and Privacy*, pp. 350–364, 2006.

[22] S. Crites, F. Hsu, and H. Chen, "OMash: enabling secure web mashups via object abstractions," in *ACM Conference on Computer and Communications Security*, pp. 99–108, 2008.

[23] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *International Conference on World Wide Web (WWW)*, 2010.

[24] T. Oda, G. Wurster, P. V. Oorschot, and A. Somayaji, "SOMA: Mutual Approval for Included Content in Web Pages," in *ACM Conference on Computer and Communications Security*, 2008.

[25] L. D. Baron, "Preventing attacks on a user's history through CSS `:visited` selectors," 2010.

[26] Z. Weinberg, E. Y. Chen, P. Jayaraman, and C. Jackson, "I Still Know What You Visited Last Summer: Leaking browsing history via user interaction and side channel attacks," in *IEEE Symposium on Security and Privacy*, 2011.

[27] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting browser state from web privacy attacks," in *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, (New York, NY, USA), pp. 737–744, ACM, 2006.

[28] E. Felten and M. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pp. 25–32, ACM, 2000.

[29] D. Morin, "Announcing Facebook Connect," 2008. `https://developers.facebook.com/blog/post/108/`.

[30] E. Hammer-Lahav, "Web Host Metadata," 2010. `http://tools.ietf.org/html/draft-hammer-hostmeta-13`.

[31] C. Jackson and A. Barth, "Beware of Finer-Grained Origins," in *Web 2.0 Security and Privacy*, 2008.

[32] Google, "Packaged Apps." `http://code.google.com/chrome/extensions/apps.html`.

[33] Mozilla, "Manifest File." `https://developer.mozilla.org/en/OpenWebApps/The_Manifest`.

[34] Google, "Verified Author." `http://www.google.com/support/chrome_webstore/bin/answer.py?hl=en&answer=173657`.

[35] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers," in *USENIX Security Symposium*, pp. 79–94, 2010.

[36] C. Reis and S. D. Gribble, "Isolating Web Programs in Modern Browser Architectures," in *ACM European Conference on Computer Systems (EuroSys)*, 2009.

[37] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[38] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[39] F. Kerschbaum, "Simple cross-site attack prevention," in *Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007. Third International Conference on*, pp. 464 –472, Sept. 2007.

[40] Mozilla, "CSP specification," 2011. `https://wiki.mozilla.org/Security/CSP/Specification#Report-Only_mode`.

[41] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications," in *21st Network and Distributed System Security Symposium*, 2014.

[42] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations.," in *NDSS*, 2013.

[43] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization.," in *USENIX Security*, pp. 399–314, 2013.

[44] C. S. Alliance, "The notorious nine cloud computing top threats in 2013."

[45] Internet Engineering Task Force (IETF), "The oauth 1.0 protocol." `http://tools.ietf.org/html/rfc5849`.

[46] Internet Engineering Task Force (IETF), "The oauth 2.0 authorization framework." `http://tools.ietf.org/html/rfc6749`.

[47] B. Fitzpatrick and D. Recordon, "Openid authentication 1.1." `http://openid.net/specs/openid-authentication-1_1.html`.

[48] Internet Engineering Task Force (IETF), "Oauth core 1.0 revision a." `http://oauth.net/core/1.0a/`.

[49] E. Hammer-Lahav, "Oauth security advisory: 2009.1." `http://oauth.net/advisories/2009-1/`.

[50] E. Hammer-Lahav, "Oauth 2.0 and the road to hell." `http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/`.

[51] Internet Engineering Task Force (IETF), "The oauth 2.0 authorization framework: Bearer token usage." `http://tools.ietf.org/html/rfc6750`.

[52] J. Bradley, "The problem with oauth for authentication.." `http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html`.

[53] Tencent Holdings Limited, "Tencent announces 2013 first quarter results." `http://www.prnewswire.com/news-releases/tencent-announces-2013-first-quarter-results-207507531.html`.

[54] Tencent Holdings Limited, "Tencent announces 2012 fourth quarter and annual results." `http://www.prnewswire.com/news-releases/tencent-announces-2012-fourth-quarter-and-annual-results-199130711.html`.

[55] Apple Inc., "Uiwebview class reference." `https://developer.apple.com/library/ios/documentation/uikit/reference/UIWebView_Class/Reference/Reference.html`.

[56] Google Inc., "Webview." `http://developer.android.com/reference/android/webkit/WebView.html`.

[57] M. Barnett and S. Qadeer, "Bct: A translator from msil to boogie," in *Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2012.

[58] "Boogie: An intermediate verification language."

[59] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Computer Aided Verification*, pp. 427–443, Springer, 2012.

[60] "A collection of online services enhanced by cst."

[61] "Nopcommerce."

[62] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations.," in *NDSS*, The Internet Society, 2013.

[63] O. Foundation, "Openid authentication 2.0 - final."

[64] "Dotnetopenauth."

[65] D. Hardt, "The oauth 2.0 authorization framework (rfc 6749)."

[66] T. FitzMacken, "Using oauth providers with mvc 4."

[67] M. Corporation, "Asp.net sample for live connect oauth sso."

[68] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "Openid connect core 1.0 incorporating errata set 1," 2014.

[69] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy, "Verified implementations of the information card federated identity-management protocol," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pp. 123–135, ACM, 2008.

[70] K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer, "Cryptographic protocol synthesis and verification for multiparty sessions," in *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pp. 124–140, IEEE, 2009.

[71] R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer, "Secure implementations for typed session abstractions," in *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pp. 170–186, IEEE, 2007.

[72] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from c protocol code by symbolic execution," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 331–340, ACM, 2011.

[73] G. C. Necula and P. Lee, *Safe kernel extensions without run-time checking.* Defense Technical Information Center, 1996.

[74] O. Goldreich, "Secure multi-party computation," *Manuscript. Preliminary version*, 1998.

[75] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 265–310, 1992.

[76] A. W. Appel and E. W. Felten, "Proof-carrying authentication," in *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pp. 52–62, ACM, 1999.

[77] S. Maffeis, M. Abadi, C. Fournet, and A. D. Gordon, "Code-carrying authorization," in *Computer Security-ESORICS 2008*, pp. 563–579, Springer, 2008.

[78] A. C. Yao, "Protocols for secure computations," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pp. 160–164, IEEE, 1982.

[79] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology–CRYPTO 2010*, pp. 465–482, Springer, 2010.