Improving security and privacy of integrated applications using behavior-based approaches

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Engineering)

by

Yuchen Zhou

December 2014

Abstract

Modern applications integrate third-party services for easier development, additional functionality such as analytics services, and extra revenue including advertising networks. This integration, however, presents risks to application security and user privacy. This research addresses integrated applications that incorporate two types of third-party services: (1) services from trusted providers that provide security-critical functionalities to the application such as Single Sign-On (SSO) and file sharing services, and (2) services from untrusted providers that provide other functionalities such as analytics and advertisements. Unlike traditional library inclusions, integrated applications present new challenges due to the opaqueness of third-party back end service and platform runtime.

For the first type of integration, we assume a benign service provider and our goal is to eliminate misunderstandings between the service provider and the application developer which may lead to security vulnerabilities in the implementation. We advocate for a systematic approach to discover implicit assumptions and SDK bugs that uses an iterative process to refine system models and uncover needed assumptions. Our preliminary results for SSO systems have shown significant opportunity and impact — of the 55 popular applications we've tested, more than half had serious security vulnerabilities due to missing at least one security-critical assumption uncovered by our approach.

To better understand the prevalence of previous discovered vulnerabilities in a larger scale, we present the design and implementation of an automated vulnerability scanner, SSOScan, that can be deployed in an application marketplace or as a stand-alone service. This testing framework can drive the application automatically and check if a given application is vulnerable by carrying out simulated attacks and monitoring application traffic and behavior.

Our evaluation results on the top 20,000 websites show that SSOScan is able to automatically check 80% of the applications for four different types of vulnerabilities, and that approximately 20% of sites which integrate the SSO service have at least one vulnerability.

For the second type of integration, the embedding application does not rely on a third-party service for security-critical functionality, but wants to prevent harm to the application and its users from embedded

services that may be malicious. Integrated services often execute as the same principal as the host application code and therefore have full access to application and user data. To mitigate the potential risks of integrating untrusted code, our approach aims to prevent third-party services from exfiltrating sensitive data or maliciously tampering with the host content. To this end, we developed two modified browsers that mediates third-party services' access to host web content based on whitelist and blacklist policies. We also developed automatic policy generators for them, and our experience and evaluation show that the whitelist approach could generate more robust and clear policies than the blacklist approach.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

Yuchen Zhou

This dissertation has been read and approved by the Examining Committee:

David Evans, Advisor

Joanne Bechta Dugan, Committee Chair

Shuo Chen

Kevin Sullivan

Westley Weimer

Ron Williams

Accepted for the School of Engineering and Applied Science:

John Stiles, Dean, School of Engineering and Applied Science

December 2014

To everyone who's helped me succeed

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern applications increasingly rely on code and services from multiple parties. The need for extra functionality and better modularity drives software developers to employ third-party solutions. These solutions differ from the traditional libraries (such as libc) since execution of integrated third-party code relies on communications with their private back end server. The recent emergence of social network giants such as Facebook and Twitter, as well as multiple advertising and analytics services have made almost every mobile or web application a complicated mash-up. A recent study done by Nikiforakis et al. [1] reveals that some websites today embed (and therefore trust) scripts coming from 295 different domains! The integration of Single Sign-On (SSO) services have been trending upwards and web statistics show that 55% of the 10,000 most popular sites use Google Analytics, 29% use Doubleclick Advertising.

1.1 Threat to Mashup Applications

The widespread integration of third-party services raises a critical problem — how to ensure desired security and privacy properties for programs integrating third-party services. Throughout this dissertation we define a *third-party service* as a module or code snippet that is provided by a party other than the application developer and which communicates with its own back end server to which the application developer has limited access. We are primarily concerned with two types of integrated applications:

1) Applications that rely on code and services provided by trusted third parties for critical security functionalities. Some examples include authentication, authorization, file sharing and payment services. In this case, the third party is trusted to be benign and its goal is to work together with its integrator to meet the security and privacy goals of the application. The attacker comes from anywhere outside of both parties,

it may collect and use any information available to the public. The goal for the attacker is to exploit logic flaws and violate the intended security properties of the integrated application.

Note that analyzing and checking the security and privacy properties of such integrated applications is more difficult than a traditional app integrating a local library such as a Java class or a C library. The key difference is that third-party services implement a significant part of functionality on their back end server, which is not controllable or even observable by the integrator. When including a local C library, the integrator has full control — the power to view or even modify the library.

2) Applications that embed untrusted third-party components for non-security-related purposes. In web application contexts, such third-party code often takes the form of JavaScript wrappers and helpers to ease web development, or provide add-on functionality such as analytics and advertising services. The embedded scripts often run as the same principal as their host application and have full access to all client resources. These services may be malicious themselves or compromised by another party to distribute malicious content.

We describe how we improve the security and privacy goals for the two scenarios in the next two subsections.

1.1.1 Integrating security services

In this scenario, it is important that the developer of the integrated apps understands the assumptions upon which secure use of the service relies. Wang et al. [2, 3] showed that even trivial and subtle misunderstandings may bring disastrous vulnerabilities in the integrated application.

Therefore, finding these gaps in understanding is a key task to eliminate vulnerabilities. We describe our approach to systematically find security-critical assumptions required by the service provider. We model the relevant behavior of the system and ignore unimportant system details such as UI interaction and underlying network implementation. After security assertions are inserted into the model, we run a model checker that outputs paths that may violate the assertions. We then analyze those paths to derive bugs and missing assumptions. After the bug is fixed or assumptions are added as facts to the model, we run the model checker again to continue this iterative approach. We are done after all security critical components of the system have been modeled.

To better understand the real-world implications, we then present the design and implementation of a tool named SSOScan, to automatically check if an application is vulnerable by testing if its behavior is consistent with the vulnerability behavior pattern we discovered. As an example, appearance of a specific type of OAuth token in the SSO traffic may be one of the vulnerability patterns. SSOScan can be used by an application marketplace such as Facebook's App Center or Google's Play Store. Contrary to general fuzzing or testing

techniques our approach takes advantage of previously known vulnerability information to help automate and guide the tests.

1.1.2 Embedding untrusted services

In this scenario, we target a different goal — understanding, monitoring, and limiting third-party scripts' access to the host application resource, and preventing exfiltration of sensitive data to untrusted hosts.

Our approach to this scenario offers fine-grained and flexible protection but relies on per-page access control policies. We developed a modified browser that supports fine-grained DOM resource access mediation and JavaScript context isolation, which state-of-the-art's same-origin policy protection does not offer. We implement a modified browser that understands and enforces fine-grained access control policies at the level of DOM nodes. It can be used to prevent accesses to part of the hosting page from third-party scripts while also allowing them to access the rest of the page to maintain functionality. The security policies may be provided by site administrators. To help reduce such efforts, we developed a proxy-based automatic policy generator which infers sensitive information based on comparing responses across sessions with different credentials.

The previously mentioned policy generator is a solid first step, but the automatic sensitive information identification still comes with a high false positive rate, and the policy generated are often not easily comprehended by the website administrators. To improve the policy generator's accuracy as well as help site owners truly understand the behavior of third-party scripts, we developed *ScriptInspector* and its associated *Visualizer* and *PolicyGenerator* extension. Compared to our previous work, the biggest improvement here is that the resource accessed by the third-party scripts can be visualized on the page and presented to the site administrator in a user-friendly fashion. The policy candidates created by *PolicyGenerator* are also much easier to be understood and maintained by human and therefore greatly improves the usability of the entire tool chain. Finally, using a training phase to build policies in an offline fashion eliminates the need to send duplicate traffic to servers, which is required by our previous approach.

1.2 Thesis

We argue that *the application developers can have much higher confidence in the security and privacy of integrated applications by using behavior-based analysis and dynamic approaches.*

We show that important security assumptions are much more unlikely to be missed when the service provider explicates their SDKs and documentations before release. Furthermore, to help ensure the security-critical instructions are correctly followed by the application developer, our evaluation results show that a

scanner tool like SSOScan can automatically and efficiently vet the applications for common implementation mistakes when integrating Single Sign-On services.

For untrusted third-party services, we show that the tool chain we developed can effectively help site administrators understand, monitor, and restrict accesses to sensitive resources based on the page access control policy. To improve usability and ease deployment resistance, we also show that the policies can be generated automatically and approved by site administrators for future intrusion detection.

1.3 Dissertation Contributions

This dissertation makes both analytical and experimental contributions addressing security and privacy issues for the two application integration scenarios. In particular, I make the following contributions to enhance the security of applications embedding security-critical services:

- A systematic and iterative approach to uncover assumptions needed for secure implementation of integrated application. This approach is an improvement over several previous works which used ad hoc analysis on similar systems and gives both parties higher confidence that the system is secure with respect to the model we build.

- Formal models of several integrated systems and runtimes. We probe the behavior of the system and investigate the SDK code and documentation to extract security-critical parts and summarize them into compact models. Such models are based on non-trivial study of system code, behavior and understanding, and are of great value to future studies.

- Evaluation of the explication approach on several important SDKs. We uncover 2 bugs and 9 implicit assumptions in such systems following the explication process. We analyze real-world applications that missed such assumption(s) and synthesize vulnerability traffic or behavior patterns that can be used in automated checking.

- SSOScan, a tool that an application marketplace such as Facebook app center, Google Play or Microsoft Live Connect developer center could use to automatically check submitted apps. SSOScan is able to automatically determine whether the app's behavior aligns with any known vulnerable pattern and provide guidance for developers to fix. By taking advantage of prior knowledge of vulnerability patterns, SSOScan is able to check 80% of applications using Facebook SSO automatically using a very short time per test case.

- We run SSOScan on the top 20,000 US websites and present the key results from the experiment. In addition to reporting the percentage of vulnerable applications, we also explain how vulnerability rates vary due to different ways of integrating Facebook SSO. For the 20% applications that SSOScan fails to scan automatically, we manually analyze them and report on the reasons for failures. Our study reveals the complexity of automatically interacting with web sites that follow a myriad of designs, while suggesting techniques that could improve future automated testing tools.

The contributions I make to protect host applications and user privacy from embedded untrusted third-party services include:

- A modified browser that understands and enforces fine-grained DOM access control and JavaScript context isolation policies. Compared to previous works, our modified Chromium browser may enforce policies down to DOM node level for individual third-party scripts and does not limit the use of JavaScript to a safer subset. Our browser also isolates the execution context of one script from another as specified by the policy.

- A proxy-based automatic policy generator that helps web application administrators generate access control policies for the modified Chromium browser. The policy generator sends two requests, one with credentials and another without, compares two responses and marks the differences as private nodes. We evaluate its identification accuracy and combine it with the modified browser to evaluate compatibility issues.

- ScriptInspector, another modified browser that intercepts and records critical resource accesses such as DOM APIs, outgoing network requests and browser configurations. When combined with the Visualizer extension, they can present a user-friendly visual representation of what resources have been accessed by a third-party script. The site administrators may then inspect the resources, weigh the advantages and risks of embedding that script, and make an informed decision of whether to include it in the website.

- PolicyGenerator for ScriptInspector, an extension that takes access records from ScriptInspector as input, uses a set of heuristics to synthesize policy candidates for site administrators to inspect, edit, and approve. The generated policy candidates can be easily understood and interpreted by a human, and the DOM resources involved can be visualized as well.

- Using ScriptInspector's visualization functionality, we made some interesting discoveries on third-party services leaking sensitive user information. We also evaluated the feasibility of deploying ScriptInspector's tool chain as an intrusion detection system, and showed that it is possible to generate

concise and effective policies for various scripts embedded in many sites that yield low false alarm rates in the detection stage.

Chapter 2

Background

This chapter provides general necessary background on mashup applications, Single Sign-On services and its Facebook flavor. Related works that are of interest with respect to certain techniques or specific applications are given later in the respective sections.

2.1 Mashup Applications

As we have stated in the introduction, modern applications are often composed of first party code and many third-party modules, which makes these applications a complex mashup [4]. Mashups are common in web and mobile applications, as shown here in Figure 2.1 as examples.

2.1.1 Embedding mechanisms

The left of Figure 2.1 is a screenshot taken from Yelp.com. This page contains a number of third-party components. The blue rounded rectangles highlight the advertisements injected onto this page. In a web application context, the display of such advertisement is often bootstrapped by a JavaScript snippet directly embedded in the host page. In addition to the elements that can be visually seen in the figure, there are numerous analytics and tracking scripts from multiple parties installed on this page, as shown in Figure 2.2. The green rectangle highlights the Google Maps module embedded by Yelp. Often times widgets like Google Maps, Facebook Like button or friend status are embedded in an *iframe* in the host page.

The right of Figure 2.1 shows a screenshot taken from a popular Android application GasBuddy. The application is used to find cheap gas stations near the user, and it embeds an advertisement from Google Admob, shown in the blue rounded rectangle. To embed a mobile ad, the application developer often downloads an SDK from the advertisement provider and include it into the packaged application. The

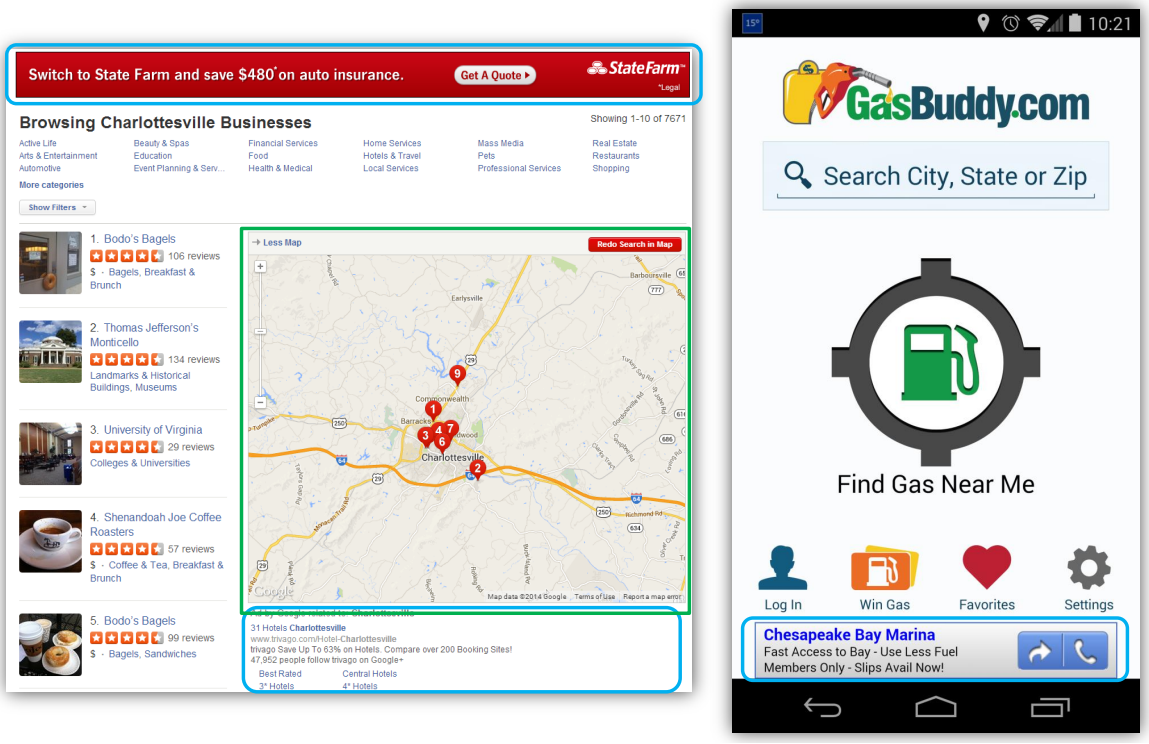


Figure 2.1: Modern mashup application examples

developer will then need to specify where he or she wants the ad to be displayed, and any additional information that may help to increase the conversion rate. Similar to the web platform, mobile applications also see a fast growing in the popularity of integrating analytics and tracking code.

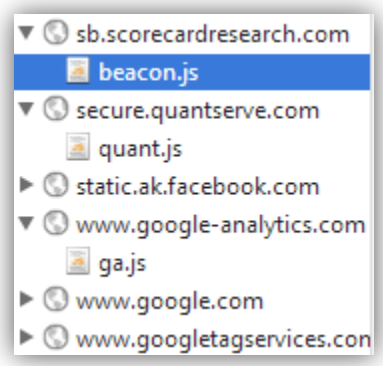


Figure 2.2: Analytics and Tracking Scripts on Yelp.com

2.1.2 Third-Party Script Privileges

Regardless of whether the third-party has malicious intent itself or if it is compromised by an adversary, such third-party modules pose security and privacy risks for the host and its users. The direct inclusion of

scripts basically grants the third-party *full* access to the host resource. The privilege ranges from reading sensitive page content, modifying page layout to accessing user credentials such as password typed into a login form, or **document**.cookie which empowers an adversary to impersonate the current user. On the contrary, accesses from a widget that is embedded in an iframe whose origin is different from the host are subject to the same-origin policy [5], and therefore are completed blocked from all host resources.

We give an example in snippet 2.1 to show what a directly embedded malicious script can do:

Listing 2.1: Example content of a malicious script

```
1 var img = document.createElement('img');
2 var c = 'cookie=' + document.cookie;
3 var p = 'pwd=' + document.getElementById('pwd').value;
4 img.src = 'http://www.evil.com/doBadThings?' + c + '&' + p;
```

After creating an image element in line 1, the script uses standard DOM APIs to obtain all first-party cookies from the host (line 2), as well as the password if the user has typed in (line 3). It then sends this information off to evil.com by setting the source attribute of the newly created image element (line 4). Although an asynchronous XMLHTTPRequest (i.e. AJAX request) is restricted by the same-origin policy and therefore cannot be used to exfiltrate this information, setting source to an image is not and such information can be transferred as GET parameters to the URL. To obtain user passwords with greater success, the above code can be set to execute as an onbeforeunload event handler for the document object, which is triggered upon page navigation — likely when the user has input his or her password and clicked the 'login' button.

The panacea appears to be simple: the host should embed all third-party code in iframes with different origins. However, online advertising these days always employs history and contextual targeting [6]. This means the advertisements are fetched catering to the user's browsing history and the page content. Accesses to these resources however, will be blocked by the *all-or-nothing* access model enforced by the same-origin policy in all major browsers today if they were embedded in an iframe with different origin. Although sometimes user tracking can be done by embedding an (almost) invisible image (a.k.a. tracking pixel), any advanced tracking and analytics functionality can only be realized by a directly embedded script.

The high risks exposed by embedding third-party script is directly caused by the inflexibility of same-origin policy, which essentially lead to the host abandoning the built-in protection for functionality and advertisement revenue. Motivated by this, we present our solution towards a much more flexible protection mechanism in Chapter 5. To this end, we are focused on protecting web applications; similar approach to cover mobile applications is a promising future research direction but may pose additional challenges and is out of the scope of this dissertation.

2.2 Single Sign-On Service

Integrating security-critical third-party services, even when the service provider is fully trusted, may bring vulnerabilities to the application. In this section, we describe one particular types of such services — Single Sign-On in more details, which is necessary for readers to understand Chapter 3 and 4.

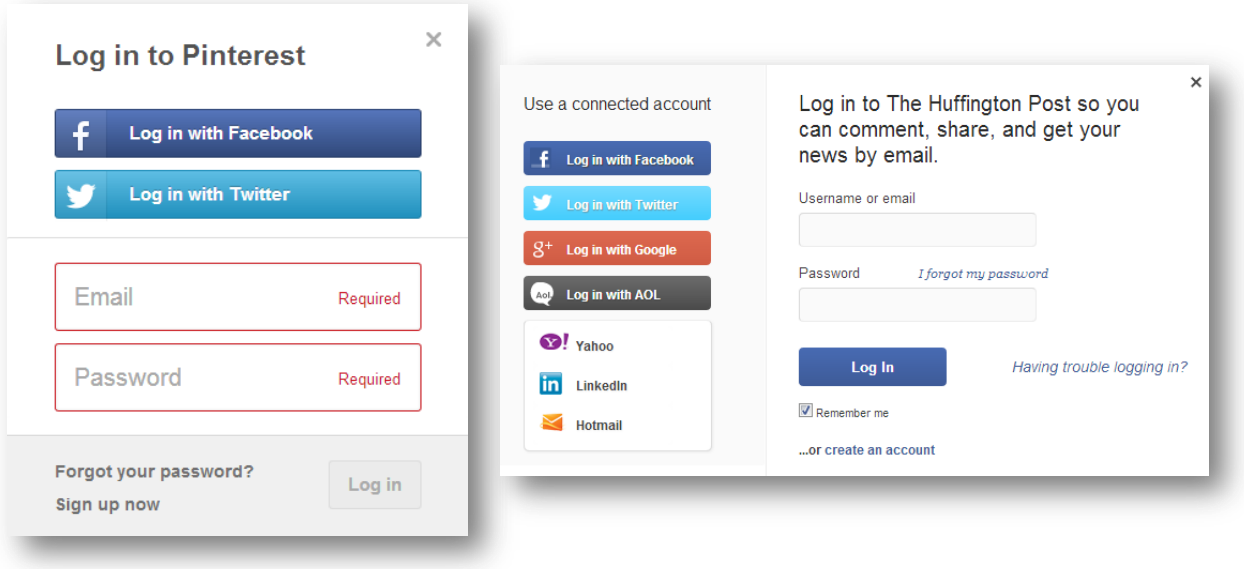


Figure 2.3: Single Sign-On enabled applications

Single Sign-On (SSO) service enables an relying party (RP) application developer to quickly implement its authentication functionality by redirecting users to log in through their account at the identity provider (IdP). The application users also benefit from this since they do not need to remember and type in user names/passwords each time they log in to the application. The relying party has the added benefit of possibly obtaining more information from the user's Facebook profile than they normally can through traditional registration, and may request privilege to post on users' walls to promote their business. Figure 2.3 shows two screenshots from pinterest.com and huffingtonpost.com, and as long as the user is currently logged in to Facebook and has previously granted permissions to those two applications, he or she can simply click on the button to login.

Up to date, the most widely deployed identity provider by far is Facebook, but several other popular options such as Twitter, Google Plus and Microsoft Live. There is no official statistics from these major identity providers about their adoption rate, but the results we obtained by running SSOScan (described in Chapter 4) show that almost 10% of the top-20,000 ranked sites implement Facebook SSO.

2.2.1 Integration approach

The most recommended approach to integrate SSO service is to use an SDK or widget provided by the IdP. For example, Facebook issues JavaScript SDK for front-end integration of web applications, and many SDK choices for various back-end framework such as PHP and Python. The developers will need to refer to documentations about the pre-defined functions in the SDK and call them when the user elects to login through SSO and when the SSO credentials are presented to the back-end server. It is worth mentioning that although most heavy-lifting (e.g. signature verification, request parameter crafting) can be done by the SDK functions, the developer would still need to make calls to those functions to ensure a secure implementation.

If experienced developers wish to further control and customize the process, IdPs like Facebook also provide documentations to their RESTful APIs. Developers can craft HTTP requests to such APIs and parse the responses themselves without the help of SDKs, however, such process often requires a deeper understanding of the SSO protocol.

Another growing business in this area is to delegate the integration to a "fourth party service", for example Janrain or Gigya. Such service often helps the RP to integrate more than one IdPs together in a "toolbox", and sometimes provide additional analytics statistics about their visitors.

Although direct client support for SSO service is much less popular than the above mentioned approaches, it has been evolving fast as of late. Devices running the Android operating system offer built-in SSO support for Google Plus accounts. Mozilla is also promoting its SSO service — Mozilla Persona, which is built into the Mozilla Firefox Browser. The advantages of using a natively-supported SSO service include SDK-free integration and push-updates to the SSO code base should vulnerabilities be discovered.

Finally, before starting to implement SSO, the application developer has to register the application at identity provider's service management center. For Facebook, this is located at `http://developers.facebook.com/`. This step is important, since Facebook needs to set up a secret known only to the developer and Facebook, and issue a valid application ID to the developer. Such information are essential for correctly requesting and parsing in the SSO process.

2.2.2 The SSO protocols

All major identity providers follow one of the Single Sign-On protocol standards, the most popular of which are OAuth [7] (version 2.0 and 1.0a) and OpenID [8]. For this dissertation we are going to focus on the OAuth 2.0 protocol, which is used by Facebook SSO and Microsoft Live SSO service. Individual IdPs may extend the protocol slightly and use different terms for credential types defined in the standard, and we highlight the

workflow of Facebook's implementation of OAuth 2.0 which serves readers the best for understanding the rest of the dissertation.

2.2.3 Facebook OAuth 2.0 workflow

A typical Facebook OAuth flow involves three parties, as shown in Figure 2.4. Alice (the user) first visits a web application (1) and elects to use SSO to login. She is then redirected (2) to the Facebook's SSO entry point with a set of pre-configured parameters. After Facebook confirms her login credentials (3), it will issue to Alice's browser the requested OAuth credentials (4), which will then be forwarded to the application back-end server (5). The application server confirms the identity by either checking the credential signature or sending a round-trip traffic to Facebook depending on the credential type (6), and if everything checks out, authenticates the client as Alice (7).

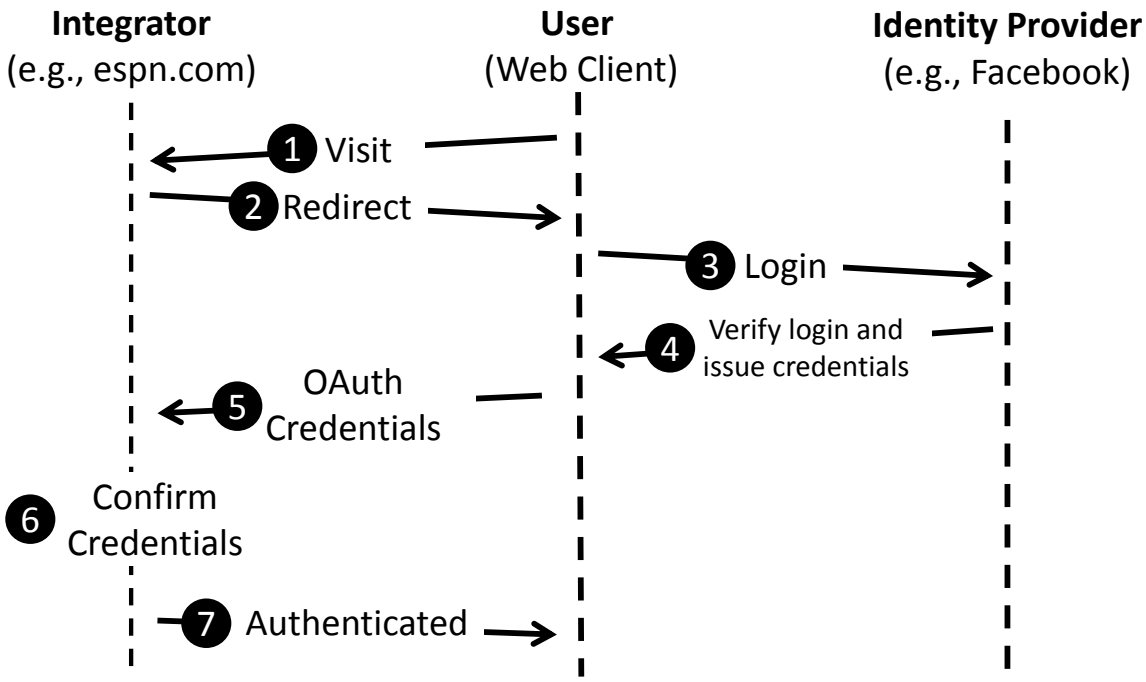


Figure 2.4: Typical OAuth Workflow

In these steps, the relying party developer needs to pay special attention to step (2), (5) and (6) to ensure a vulnerability-free implementation, while the Facebook's back-end server takes care of (3) and (4).

2.2.4 Facebook Connect credential types

The OAuth 2.0 standard specifies that the IdP should return different types of credentials (shown in step 4 of Figure 2.4) based on different *grant_type* values submitted in the HTTP request. Here we list the ones

Facebook adopts, including an extended, Facebook-specific type.

- **access_token.** *Access_token*s is the basic OAuth credential type designed for authorization purpose. It is a bearer token, and represents user's granted permission to the token holder. A Facebook *access_token* is only issued when a request to the OAuth entry point API explicitly asked for it. An API call made to request the token must contain the application ID (set up earlier) and requested permissions. After the user agrees, an access_token bound to those permissions is issued to the developer, which can be used as credential to call further Facebook *graph* APIs and request additional user information. *Access_token* is often valid from several hours to several days but can be extended to a much longer period by calling a token refresh API.

- **code.** A Facebook *code* credential is issued by default to calls which do not specify a *grant_type*, and can be used for both authentication and authorization purposes. By itself, *code* does not contain any meaningful information to the developer, is only valid for a short period of time, and has to be used to exchange a valid *access_token*. To do this, the developer needs to append its *client_secret* (set up earlier) with the *code* in the API call to retrieve the token. The *access_token* obtained this way can be used to request additional user information as described above.

- **signed_request.** *Signed_request*s are an extended type of OAuth credential that is specific to Facebook. It enables applications to perform authentication *without* initiating communication with Facebook, unlike the *code* flow. This is done by signing a data blob containing user information (his or her Facebook user ID, email address, and possibly a valid *code* to obtain *access_token* upon request) using the *client_secret* known only to the application owner and Facebook. To complete the authentication process, the developer needs to verify the signature of *signed_request* before recognizing the user.

We leave out the *refresh_token* type defined in OAuth 2.0 standard as it is irrelevant to this dissertation.

Chapter 3

The Explication Process[1]

In this Chapter and Chapter 4, we consider the scenario where the third-party service provider is benign and works with the host, but the integrated service implements security-critical functionality of the system, such as authentication and authorization.

Implementing secure authentication is an important requirement for any modern application that interacts with its users. Even when the application's native authentication (traditional user name and password log in) is secure, integrating third-party SSO services may significantly complicates the situation. Recent years, many researchers have done work to find vulnerabilities in such integrations, however, no previous study has rigorously examined the security properties these SDKs provide to real-world applications. Typically, SDK providers simply release SDK code, publish related documentation and examples, and leave the rest to the app developers. An important question remains: if developers use the SDKs in reasonable ways, will the resulting applications be secure? We show in this chapter that the answer today is *No*. The majority of apps built using the SDKs we studied have serious security flaws. In addition to direct vulnerabilities in the SDK, many flaws exists because achieving desired security properties by using an SDK depends on many implicit assumptions that are not readily apparent to app developers. These assumptions are not documented anywhere in the SDK or its documentation. In several cases, even the SDK providers are unaware of these assumptions.

This motivates us to develop a more systematic approach — *the explication process*, through which we hope to offer more confidence to the service provider and the application developer that no missing assumptions or vulnerabilities are left behind. The goal of this work is to systematically identify the assumptions needed to securely integrate the SDK. We emphasize that it is not meaningful to verify an SDK by itself. Instead,

[1]The contents of this chapter is based on the paper: Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization [9].

our goal is to explicate the assumptions upon which secure use of the SDK depends. We do this by devising precise definitions of desired security properties, constructing an explicit model of the SDK and the complex services with which it interacts, and systematically exploring the space of applications that can be built using the SDK. Our approach involves a combination of manual effort and automated formal verification. Any counterexample found by the verification tool indicates either (1) that our system models are not accurate, in which case we revisit the real systems to correct the model; or (2) that our models are correct, but additional assumptions need to be captured in the model and followed by application developers. The explication process is an iteration of the above steps so that we document, examine and refine our understanding of the underlying systems for an SDK. At the end, we get a set of formally captured assumptions and a semantic model that allow us to make meaningful assurances about the SDK: an application constructed using the SDK following the documented assumptions satisfies desired security properties.

We argue that the explication process should be part of the engineering effort of developing an SDK. Newly discovered security-critical assumptions through this process can either be dismissed by modifying the SDK (the preferable solution), or be documented precisely.

For the rest of this chapter, we first discuss prior works in more details in Chapter 3.1. We then describe an illustrative missing implicit assumption example in Chapter 3.2, how the iterative explication process works in Chapter 3.3, how various parts of the system are modeled in Chapter 3.4, and finally the implicit assumptions we uncovered using this approach on Facebook and Microsoft's SSO services in Chapter 3.5 and their exploit opportunities in Chapter 3.6.

3.1 Prior Works

The idea of formally verifying properties of software systems goes back to Alan Turing, although it only recently became possible to automatically verify interesting properties of complex, large scale systems. The explication process makes use of considerable advances in model checking that have enabled model checkers to work effectively on models as complex as the ones we use here. This approach is most closely related to other work on inferring and verifying properties of interfaces such as APIs and SDKs, which are briefly reviewed next.

3.1.1 API and SDK Misuses

It is no longer a mystery that APIs and SDKs can be misunderstood and the results often include security problems. On various UNIX systems, *setuid* and other related system calls are non-trivial for programmers to understand. Chen et al. "demystified" (that is, explicated) these functions by comparing them on different

UNIX versions and formally modeling these system calls as transitions in finite state automata [10]. Wang et al. [2, 3] showed logic bugs in how websites integrate third-party cashier and SSO services. Many of the bugs found appear to result from website developers' confusions about API usage. Georgiev et al. [11] showed that SSL certificate validations in many non-browser applications are broken, which make the applications vulnerable to network man-in-the-middle attacks. Our work started from a different perspective — our primary goal is not to show that SDKs can be misused, but to argue that these misuses are so reasonable that it is SDK providers' lapse not to explicate the SDKs to make their assumptions clear. We expect that our approach could be adapted to other contexts such as third-party payment and SSL certificate validation.

3.1.2 Program and Interface Verification

Various techniques have been proposed to help automatically verify certain program properties regarding library interfaces. Spinellis and Louridas [12] built a static analysis framework for verifying Java API calls. Library developers are required to write imperative checking code for each API to assist the verification process. Henzinger et al. [13, 14] worked on a set of languages and tools to help model the interfaces and find assumptions that need to be met for two APIs to be *compatible*. The compatibility check is performed by computing compatible states, which amounts to solving a game between the product automaton (which tries to enter illegal states) and its environment (which tries to prevent this). JIST [15] infers necessary call sequences given a class and an exception property for Java, so that the exception is never raised. Our proposed work has similar goals but is more focused on the study of details of complex real-world integrated systems and developing a systematic approach to analyze them. We do not assume source code access from the system. For example, our work involves probing and summarizing semantics for various parts of the SSO system including the device runtime, identity provider back end server, and application back end server.

Several works have automatically inferred program invariants or specifications. Daikon [16–18] automatically learns program invariants of variables (strings, certain data structures, etc.) by instrumenting the program, running test cases, collecting and analyzing program traces. Felmetsger et al. [19] applies Daikon to web applications to detect logic vulnerabilities. Yang and Evans [20, 21] proposed approaches to discover temporal invariants by analyzing execution traces. Weimer et al. [22, 23] proposed automated and generic ways to mine temporal specifications based on the assumption that certain parts of the program are more error-prone than others, such as code with poor readability or traces that raise exceptions. Sekar [24] does approximate string comparison to infer input/output relationship between web application requests and responses to help eliminate code injection attacks. Such inference techniques generally require a large number of execution traces, as well as that the majority of them are correct. This assumption is not necessarily true

in our problem.

3.1.3 Protocol Analysis

Bansal et al. [25] modeled OAuth 2.0 protocol and verified it using ProVerif [26]. They also built a library for future researchers to model web APIs into ProVerif language in a easier fashion. Pai et al. [27] used Alloy framework [28] to verify OAuth 2.0 and discovered a previously known vulnerability. Sun et al. [29] discussed a number of website problems affecting OAuth's effectiveness, such as not using HTTPS, having cross-site scripting and cross-site request forgery bugs. The fact that both SSO services we studied are based on OAuth is mainly because of its widespread adoption, but the security issues we found concern the SDK and service implementations, rather than flaws inherent in the OAuth protocol.

In addition to OAuth protocol analysis, Fett et al.* [30] used mathematical model (in form of Pi calculus) to verify Mozilla Persona Single Sign-On, which follows BrowserID protocol. They found a number of critical security flaws and reported to Mozilla team. Dietz and Wallach [31] investigated the weaknesses of TLS used in token transportation in the BrowserID protocol and proposed an extension of TLS to protect against potential token leakage vulnerability. Again, the reason that we focus our work on OAuth protocol is because of its popularity and authorization capabilities (OpenID and BrowserID protocol does not support authorization), which may cause additional vulnerabilities.

Similar to our goal of addressing the root cause of the vulnerabilities, Cao et al.* [32] propose to completely redesign the client-IdP-RP communication. They argue that the root cause of many SSO vulnerabilities are a result of the leaked and misused browser-relayed messages. While we agree with this statement, requiring establishment of secure communication channels between all three parties would significantly change all current implementations of SSO integration.

3.2 An Illustrative Example

To motivate our work, we describe a simple example in the context of the Microsoft Live Connect (Microsoft's Single Sign-On service). It illustrates what can go wrong when SDKs are provided without thoroughly specifying their underlying security assumptions.

*Works done after our paper was published.

3.2.1 Intended Use

To start things from scratch, we assume the developer has no previous experience with such implementation. The most reasonable first thought is to visit the developer guide[1] for Microsoft Live [33]. This page provides code snippets in JavaScript, C#, Objective-C and Java showing how to use Live Connect SDK to sign users into a client app. Ignoring the irrelevant details, we generalize the code snippets to the authentication logic shown in Figure 3.1.

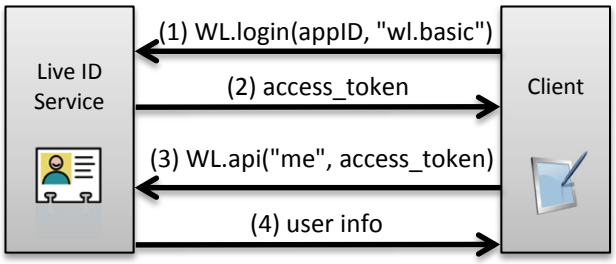


Figure 3.1: Documented Microsoft Live Connect authentication logic

In the figure, WL stands for Windows Live. A developer first needs to call WL.login defined in the JavaScript SDK. The call takes an argument value "wl.basic", indicating that the app will need to read the user's basic information after the function returns an *access_token* in step (2). The *access_token* is the same as described earlier in Chapter 2.2.2. Once the app obtains the *access_token*, it calls another function, shown in step (3), to get the user's basic information. The call is essentially equivalent to issuing an HTTP request to the following URL:

```
https://apis.live.net/v5.0/me?access_token=ACCESS_TOKEN
```

The Live ID service responds with the user's name and Microsoft Live ID in the message in step (4). This completes the process, authenticating the user with the provided information.

3.2.2 Unintended (Hazardous) Use

The developer guide as depicted in Figure 3.1 is valid for a client-only application, but it is not clear that the same logic must not be used with an application that also incorporates an online service. Without stating this explicitly, developers may be inclined to use the SDK insecurely as shown in Figure 3.2. The interactions with the Live ID service are identical in the two figures. The only difference is that in the second scenario, the *access_token* is sent back to the back-end server of the application in step (2a) and it is the app server that makes the REST API call to retrieve user information and authenticate.

[1] Information accurate as of April 2014.

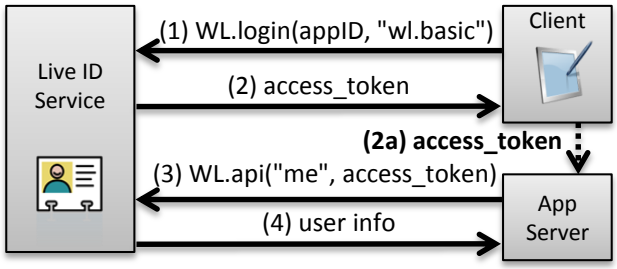


Figure 3.2: Vulnerable Microsoft Live Connect authentication logic

This can lead to a serious vulnerability that allows any application on the device to sign into the app server as the user. A rogue app may send a legitimate request to the Live ID service for an *access_token* to view public information of the victim, and the user granted the requested information without suspicion. The problem is this token, obtained by the rogue app and intended for authorizing access to the user's resource, can be abused and sent to the vulnerable application's back-end server from the rogue app developers to perform an impersonation attack. Since the vulnerable app's server never checks which application the *access_token* is issued for, it would happily accept the token from rogue app developers and authenticate them as the victim. A quick search on the app market reveals that this mistake is fairly common in real world applications. In addition, although we first observed it when analyzing the Live Connect documentation, we later found that many apps using the Facebook SDK suffer from the same issue.

3.2.3 Resolution and Insights

From one perspective, this is simply a matter of developers writing buggy apps, and the blame for the security vulnerability rests with the app developers. We argue, though, that the purpose of the SDK is to enable average developers to produce applications that achieve desired security properties by default, and the prevalence of buggy apps created using this SDK indicates a failure of the larger engineering process. The developer exercised reasonable prudence by using the *access_token* to query the ID service for user information and followed exactly the process described in the SDK's documentation (Figure 3.1). The problem is a lack of deeper understanding of the difference between authentication and authorization, and the role of the *access_token* (i.e., why it is safe to use the *access_token* as shown in Figure 3.1 but not in Figure 3.2). Correct usage depends on subtle understanding of what kind of evidence each message represents and whether or not the whole message sequence establishes an effective proof for a security decision. It is unrealistic to expect an average developer to understand these subtleties, especially without a clear guidance from the SDK.

We contacted the developers of some of the vulnerable apps. A few apps have been fixed in response to our reports. We also notified the OAuth Working Group in June 2012 about these vulnerable apps[1]. Dick Hardt, one of the editors of OAuth 2.0 specification (RFC 6749) [7], emailed us requesting a paragraph to be included in the specification to address this issue. We proposed the initial text and discussed with working group members. This has resulted in Section 10.16 "Misuse of *access_token* to Impersonate Resource Owner in Implicit Flow" being added to the specification.

The key point we want to get across in this example is that security of applications constructed with an SDK depends on an understanding of the external service offered by the SDK provider, as well as subtleties in the use of tokens and assumptions about evidence used in authentication and authorization decisions. We believe the prevalence of vulnerable apps constructed using current SDKs yield compelling evidences that a better engineering process is needed, rather than just passing the blame to overburdened developers. This motivates us to advocate for a process that explicates SDKs by systematically identifying the underlying assumptions upon which secure usage depends.

3.3 The Explication Process

In order to explicate the SDKs, we need to clearly define the desired security properties. This Chapter introduces our target scenario and threat model, and then defines the desired security properties and gives an overview about the explication process for uncovering implicit SDK assumptions.

3.3.1 Scenario

A typical question about security is whether some property holds for a system, even in the presence of an adversary interacting with the system in an unconstrained manner. Such problems are often tackled by modeling a concrete target system and an *abstract* adversary (i.e., a test harness in software testing terminology) that explores all interaction sequences with the concrete system. In our scenario, however, the target system is not even concrete. We wish to reason about all applications that can be built with the SDK following documented guidelines. Hence, we need to consider both the application's client and server components as abstract modules.

Without mingling in the threat model, Figure 3.3 illustrates the three parties involved in an SSO process. There are three main components: application client on the left, application server on the right, and the

[1]Subsequently, we learned that John Bradley, a working group member, had posted a blog post in January 2012 about a similar issue [34]. The post considers the problem a vulnerability of the protocol, while we view it as a consequence of an unclear assumption about SDK usage because there are correct ways to use OAuth for service authentication

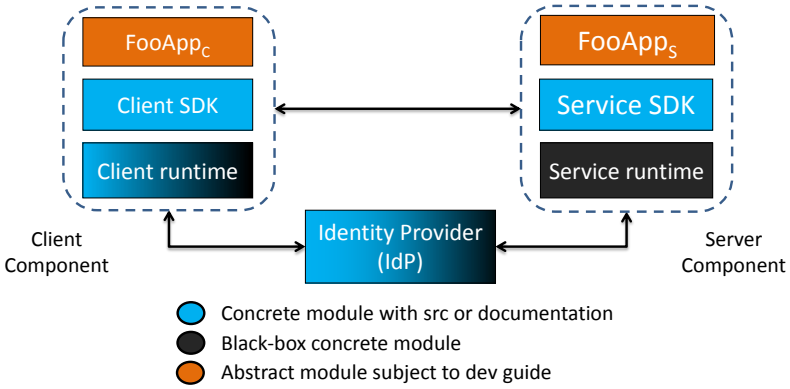


Figure 3.3: Parties and Components in the SSO Process

identity provider on the bottom. Both the client and server applications are abstract modules, and they can be further divided into three layers.

The bottommost layer represents all the client runtime, e.g. the browser or application VM. Special attention has to be exercised when learning and modeling this layer. These are complex modules that an outsider researcher typically does not understand in detail in the beginning of the study. Developing a semantic model for these components involves substantial system investigation effort (as described in Section 3.4) because the seemingly clear SDK behavior running on top (even with source code available) actually depends a lot on this mysterious (and often incompletely documented) underlying runtime. We consider the formal semantic models resulting from this study as one of the main contributions of this work.

Assuming the developer always integrates the IdP-provided SDK for authentication and authorization, the middle layer consists of such SDKs, which we often have source code access to. These are relatively easy to learn and model compared to the rest of the layers.

On top of these two layers lies the application code — FooApp_C and FooApp_S for client and server component respectively. We assume that FooApp_C and FooApp_S do not directly interact with the service runtime except via the middle layer SDK. Note that if the IdP receives a call from either of these components, it has no reliable way to tell if the caller is a client device or an application server. FooApp_C and FooApp_S belong to the *abstract but restricted* module type: At modeling time, they do not have concrete implementations, and our goal is to reason about all possible apps built on top of the other two layers; nevertheless, the app modules do have constraints on their behaviors: we assume their developers are reasonable and therefore the modules must not violate the rules documented in the existing SDK developer guides.

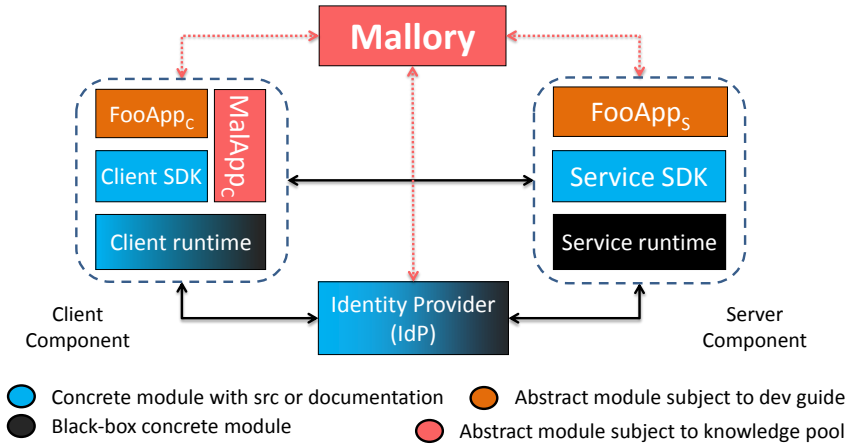


Figure 3.4: Threat Model in the SSO Process

3.3.2 Threat Model

We assume a malicious application, MalApp$_C$ shown in Figure 3.4, may be installed and ran on the victim's device. This can be done by either offering an incentive or masquerading as another popular benign application. For the malicious application, its behavior is not constrained by the client SDK, therefore it is closer to be a fully abstract module, only to be limited by the functionalities provided by the client device runtime (e.g., it cannot access cookies of other domains or access restricted files). In addition, the attacker also controls an unconstrained external machine, which we call "Mallory". Readers can think of Mallory as a combination of a client and server that can freely initiate communication with all the rest parties.

3.3.3 Security Properties

A precise understanding of the security properties that an integrated service should provide is essential for explicating the SDK: they determine which assertions should be added to the model. We have identified several general principles of security property with respect to third-party services: 1) *verification* — a secret or signature needs to be verified before it is trusted; 2) *secrecy* — user credentials or application secrets must not be leaked through any API calls; 3) *consistency* — bindings of user data must be consistent across transactions; and 4) *complete mediation* — every access to a protected resource must be check for appropriate authorization.

Applying these principles to Facebook's SSO service, we define three important security properties: 1) *authentication* — the application must verify the identity and signature of submitted credentials upon consumption to ensure protection against impersonation attacks as described in the illustrative example; 2) *authorization* — OAuth tokens bearing user permission must stay within the session, ensuring no unsolicited

permission leakage; and 3) *association* — the user identity of the application must match the identity of her associated OAuth tokens, ensuring consistencies between OAuth credentials and session identities.

3.3.4 The Iterative Explication Process

Explicating SDKs is a systematic investigation effort to explicitly document our knowledge about these modules and examine the knowledge against defined security goals. Figure 3.5 shows that this is an iterative process, in which we repeatedly refine our model and formally check if it is sufficient to establish the security properties or if additional assumptions are needed. A failed check (i.e., a counterexample in the model) indicates either that our understanding of the actual systems needs to be revisited or that additional assumptions are indeed needed to ensure the desired security properties.

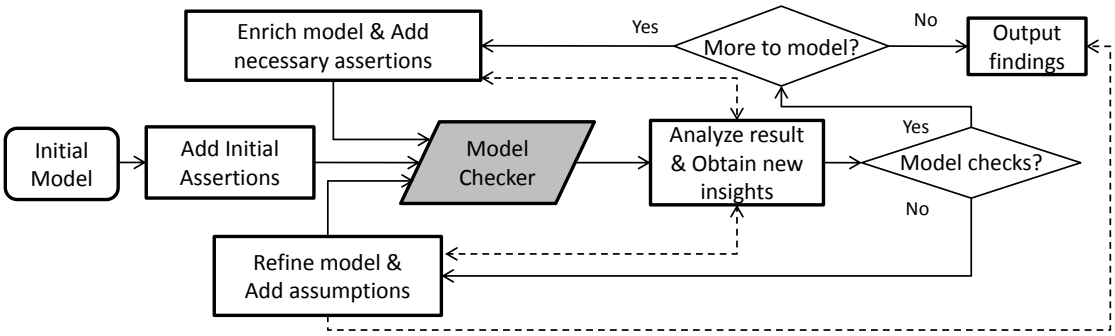


Figure 3.5: Iterative process to uncover security-critical assumptions

The outcome of the process is the assumptions we explicitly added to the model, as well as all other adjustments we made to the system behavior model in order to ensure all security properties. These adjustments reflect bugs/flaws in real world systems and can be fixed based on our modifications to the model. In Chapter 3.5, we show that many of the uncovered assumptions can indeed be violated in realistic situations.

3.4 Semantic Modeling

In this chapter, we give an overview of the semantic modeling effort for the three SDKs we studied — Facebook Connect PHP SDK, Microsoft Live Connect PHP SDK and Microsoft modern app client side SDK. The resulting models are available at https://github.com/sdk-security/. They reflect a total of 12 person-months' effort in creating and refining the system models, however, part of this effort is spent to develop the idea of explication. We believe this cost can get much lower as the researcher gets more

experience about the explication process, or if the explication can utilize any inside information of either the IdP or the application platform provider.

3.4.1 Modeling language

To specify the semantics of the modules, we need to convert behaviors of the real-world system to a language that has a suitable formal analysis technology for verification. In the first period of our investigation, we used Corral [35], a property checking tool that can perform bounded verification on a C program with embedded assertions. Corral explores all possible execution paths within a bound to check if the assertions can be violated. Later, we re-implemented all the models in Boogie [36], a language for describing proof obligations that can then be tested using an SMT solver, which allowed us to fully prove the desired properties. This provides a higher assurance than the bounded verification done by Corral, but the basic ideas and approach are the same for both checking strategies.

Based on our experience with both formal analysis tools, we recommend that future explication process starts with a bounded model checker that works directly on C-like (or other well-understood languages) syntax, does not require loop invariants to be explicitly specified, and outputs human-friendly counterexample traces. This significantly reduces the learning curve, which gives researchers more time to focus on behavior probing and converting added assumptions to real-world vulnerabilities. However, after more experience are gained and when the model is relatively stable, switching to a theorem prover that can perform unbounded verification ensures the security properties cannot be violated with a much higher confidence.

For brevity, this section describes the Boogie version to explain our modeling. The key Boogie language features necessary to understand this paper are:

- The * symbol represents a non-deterministic Boolean value.
- HAVOC v is a statement that assigns a non-deter-ministic value to variable v.
- ASSERT(p) specifies an assertion that whenever the program gets to this line, p holds.
- ASSUME(p) instructs Boogie to assume that p holds whenever the program gets to this line.
- INVARIANT(p) specifies a loop invariant. Boogie checks if p is satisfied at the entry of the loop, and inductively prove p's validity after each iteration.

If Boogie fails to prove an assertion or an invariant, it reports a counter-example. This leads us to refine the model, and adding assumptions when necessary.

3.4.2 Modeling Concrete Modules

Concrete modules do not have any non-determinism. The key aspects of building semantic models for the concrete modules are summarized below.

Data types. The basic data types in the models are integers and several types for enumerables. We also define struct and array over the basic types. In the actual systems, the authentication logic is constructed using string operations such as concatenation, tokenization, equality comparison, and name/value pair parsing. We found that most string values can be modeled into integers without losing their relevant security logic representation, except those of domain names and user names, which we canonicalize as Alice (the victim), Mallory (the adversary), foo.com (the web application developer), and etc. It is important that we limit the data types used in the model to maintain compatibility with the theorem prover and improve performance of the model checker.

SDKs. The SDKs we studied are of moderate size (all under 2000 lines) and their source code are all published in the public. They were implemented in HTML, JavaScript and PHP, so we first translate the SDKs function-by-function into the modeling language. We do this translation manually, and although this process can likely be automated by program language tools, they lack of the functionality to abstract data structures and omit security-irrelevant details such as GUI manipulation and networking code.

Listing 3.1: Example Facebook PHP SDK code

```
1  protected function getUserFromAvailableData() {
2          if ($signed_request) {
3                  ...
4                  $this->setPersistentData('user_id',
5                          $signed_request['user_id']);
6                  return 0;
7          }
8          $user = $this->getPersistentData('user_id', $default = 0);
9          $persist_token = $this->getPersistentData('access_token');
10         $access_token = $this->getAccessToken();
11         if ($access_token &&
12                 !($user && $persist_token == $access_token)) {
13                 $user = $this->getUserFromAccessToken();
14                 if ($user)
15                 $this->setPersistentData('user_id', $user);
```

```
16                  else $this->clearAllPersistentData();
17      }
18      return $user;
19  }
20
21  public function getLogoutUrl() {
22          return $this->getUrl(
23                  'www', 'logout.php',
24                  array_merge(array(
25                  'next' => $this->getCurrentUrl(),
26                  'access_token' => $this->getAccessToken(),),...));
27  }
```

Listing 3.2: Example Facebook PHP SDK model

```
1  procedure  {:inline 1} getUserFromAvailableData()
2  returns (user:User) {
3    if (IdP_Signed_Request_user_ID[sr] != _nobody) {
4      ...
5      user := IdP_Signed_Request_user_ID[sr];
6      call setPersistentData_user_id(user);
7      return;
8    }
9    call user := getPersistentData_user_id();
10   call persisted_at := getPersistentData_access_token();
11   call access_token := getAccessToken();
12   if (access_token >= 0 &&
13     !(user != _nobody &&
14     persisted_access_token == access_token)) {
15       call user := getUserFromAT(access_token);
16       if (user != _nobody) {
17         call setPersistentData_user_id(user);
18       } else {
19         call clearAllPersistentData();
20       }
21   }
```

```
22    return;
23  }
24
25  procedure {:inline 1} getLogoutUrl()
26  returns (API_id: API_ID, next_domain: Web_Domain,
27  next_API: API_ID, access_token: int)    {
28    API_id := API_id_FBConnectServer_login_php;
29    call access_token := getAccessToken();
30    call next_domain, next_API := getCurrentUrl();
31  }
```

For example, Listing 3.1, 3.2 show two functions in the Facebook PHP SDK and their corresponding Boogie models. For function getUserFromAvailableData, the changes are essentially line-by-line translations with objects converted to simple data types in a straightforward fashion. For getLogoutUrl, the PHP code performs a string operation and returns a string. Our Boogie translation in this case is not obviously line-by-line. For example, our procedure returns a four-element vector instead of a string. The PHP function calls getUrl and array_merge, which concatenate substrings, therefore, are implicitly modeled by the four-element return vector.

Underlying system layer. Unlike the SDK, which is simple enough to model completely, the identity provider, client runtime, and server runtime are very complex and often do not even have source code available. Completely modeling every detail of these systems is infeasible, but our analysis depends on developing suitable models of them. To this end, we try to reconstruct the model by observing and probing the behavior of black-box components.

Particularly, the identity provider responds differently to different input arguments and various app settings in its web portal. Each identity provider has a web page for app developers to tweak a number of settings, such as app ID, app secret, service website domain, and return URL. Many of these settings are critical for the identity provider's behavior. Further, different inputs to the provided APIs cause different responses. Because we do not have the source code for the identity providers, we probe these behaviors by constructing different requests and app settings, and monitoring the responses using a proxy server. For example, the Microsoft Live Connect API dialog_permissions_request(), RST2_srf() and oauth20_authorize_srf() models involved 11, 8 and 6 if-statements respectively, to describe different behaviors we observed in the probing process. Similarly, for client runtime we set up different parameters to call platform APIs defined in Windows 8 modern apps SDK and observe their return values.

3.4.3 Modeling Abstract Modules

The abstract modules interacts with the concrete modules in a non-deterministic manner. It includes both the benign application and adversary controlled resources.

Non-deterministic Calls. The key attributes of a model for an abstract module is to be able to exhibit non-deterministic behaviors. To this end, we put the HAVOC statement, which generates a non-deterministic value inside a **switch** statement, with possible actions inside each **case** statement. In the execution of the formal analysis tools, the symbolic values will be automatically assigned to violate any assertion if possible. This part is not new to the model checking community, however, the situation is more complicated here as the benign application will need to be 'reasonable' and follow the documented guidelines. For example, we assume the developers never send out its app secret intentionally to the adversary. For the guidelines, we exercise caution to read and enforce the guidelines on the switch statements. Note that under-restriction is always better than over-restriction, as the latter will cause the symbolic execution engine to miss possible implicit assumptions.

Knowledge Pool. For the adversary, its behavior is not restricted by the developer guide; however, every call it makes will be require to supply meaningful parameters. We introduce the concept of *knowledge pool*, which models the adversary's current knowledge as the test harness runs. Different types of knowledge, such as *access_token*, *Code*, and PHP session IDs, are explicitly differentiated. An AddKnowledge function for each security-critical credential type is added as Mallory's capabilities. Initially, the knowledge pool is populated with publicly available information, such as the benign application's app ID, as well as Mallory's own credentials to IdP and Foo application. As the adversary makes calls, it adds all information returned by that call to its knowledge pool. Subsequent calls non-deterministically select a set of parameters out of all applicable type in the knowledge pool. Finally, we consider attacks that involve providing arguments of the incorrect type out of scope, e.g., giving a session ID to a function expecting an access_token.

Test Harness. Finally, to put everything together and complete the test harness, we use an outside-most loop and another non-deterministic switch statement that has a limited loop count depth for Corral, and an endless loop for the Boogie theorem prover. The loop count represents the maximum steps allowed for Corral to explore counterexamples. For theorem prover this is unnecessary since it verifies the model based on the loop's operational semantics and manually supplied loop invariants. Inside the switch statement, each case can be an API call from Foo application, a Mallory's move from the user's device or their back-end server. The entire process is shown in Figure 3.6.

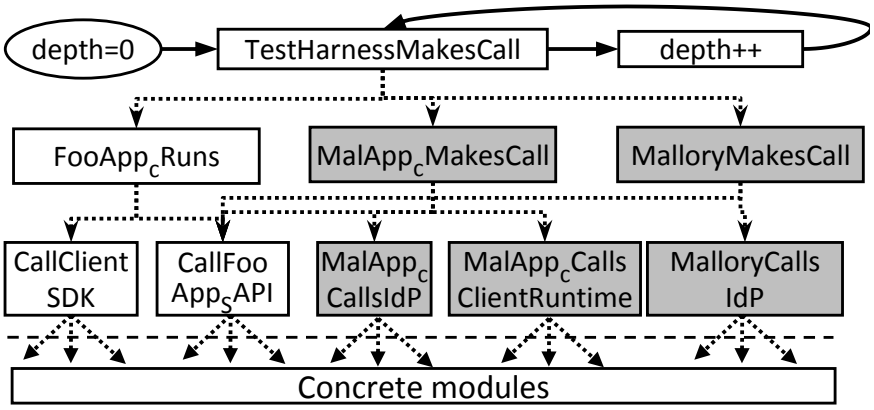


Figure 3.6: Test harness workflow

3.4.4 Constructing Assertions

This Chapter describes how we use ASSERT statements to document and test the desired security properties, covering each of the security violations described in Chapter 3.3.3.

Authentication violation. An authentication violation occurs when an attacker acquires some knowledge that could be used to convince FooApp$_S$ that the knowledge holder is Alice. A simple example is the case we described in Chapter 3.2, in which the knowledge obtained is an *access_token*. In addition, we also consider IdP-signed data such as Facebook's *signed_messages* or Live ID's authentication tokens.

As an example of the assertions in the model, when a Facebook *signed_messages* k is added to the knowledge pool, we assert that

k.user_ID= _alice && k.app_ID != _foo_app_ID && TokenRecordsOnIdP[k.token].user_ID != _alice!

where TokenRecordsOnIdP represents IdP's database storing the records of *access_token*s.

Authorization violation. To detect authorization violations, we add ASSERT statements inside the Mallory knowledge pool. For example, the assertion in function AddKnowledge_Code is:

ASSERT(!(c.user_ID == _alice && c.app_ID == _foo_app_ID))

This checks that the *Code* added to the knowledge pool is not associated with Alice on FooApp, and similar assertions are added to other credential types. The app secret is different from the above knowledge types, because it is tied to the app not the user. Therefore, an assertion must be added that under no circumstances can an app secret be leaked to any party other than Foo and the IdP.

Association violation. At the return point of every web API on FooApp$_S$, we need to ensure the correct association of the user ID, the permission (represented by an *access_token* or *Code*), and the session ID. This

ensures that the three key variables of the session all involve the same user.

3.5 Summary of explication discoveries

We applied the explication process to the Facebook PHP SDK, Microsoft Live Connect SDK and Windows 8 Modern app SDK. The Facebook PHP SDK is the only server-side SDK officially supported on Facebook developers website and is currently among the most widely used third-party SSO SDKs. Facebook also has client SDKs for Android and iOS apps, which have many concepts similar to the PHP SDK, but we have not studied them in detail. The Live Connect SDK is provided by Microsoft for developing metro apps that use Live ID as the identity provider. The Windows 8 Modern app SDK is another necessary lower layer SDK to support metro apps to use OAuth-based authentication (IdP not limited to Microsoft).

3.5.1 Assumptions Uncovered

The models resulting from our study formally capture what we learned about the SDKs and the systems. Our assumptions are specified in two ways: (1) all the ASSUME statements that we added; (2) when we need to assume particular program behaviors, such as a function call must always precede another, we model the behaviors accordingly, and add comment lines to state that the modeled behaviors are assumptions, rather than concrete facts. All the assumptions are added in order to satisfy the assertions that described in Chapter 3.4.4.

Verification. After all the assumptions were added, the models were automatically verified by Corral with the bound set to 5, meaning that the depth counter of the main loop never exceeds 5 in the test harness (Figure 3.6). Such a depth gives a reasonable confidence that the security properties are achieved by the models and the added assumptions: the properties could only be violated by attacks consisting of six or more steps. Running on a Windows server with two 2.67GHz processors and 32GB RAM, it took 11.0 hours to check the Facebook PHP SDK, 26.3 hours to check Live Connect SDK (Both verification require Windows 8 Modern app SDK as the underlying SDK).

Unbounded verification. The verification being bounded is a limitation of the models built for Corral, so we subsequently re-implemented all three models in Boogie language. As stated before, verification of Boogie models is not automatic. It requires human effort to specify preconditions and post-conditions for procedures, as well as loop invariants. The Boogie verifier checks that (1) every precondition is satisfied at the call site; (2) if all preconditions of the procedure are satisfied, then all the postconditions will be satisfied when the procedure returns; (3) every loop invariant holds before entering the loop, and if after an iteration it still

SDK Name	Assumption	Impact when violated	Exploit opportunity	Vendor response
A1 (FB)	In FooAppCMakesACall, assume FooAppC.cookie.sessionID == _aliceSession.	Mallory's session will be associated with with Alice's user ID.	When the SDK is used in subdomaining situations, e.g., cloud domains	Countermeasure on service platform
A2 (FB)	For any PHP page, after calling getUser, getAccessToken must be called subsequently.	Alice's user ID will be associated with Mallory's returns the *access_token*.	When $FooApp_S$ contains a PHP page that directly userID	SDK code fix
A3 (FB)	Before getLogoutUrl returns, assume logoutURL.params.access_token != getApplicationAccessToken()	App access token is added to the knowledge pool (owned by the adversary).	When a PHP page does not have the second code snippet shown in the dev guide	SDK code fix
A4 (LC)	In saveRefreshToken on $FooApp_S$, assume user_id != refresh_token.user_id	Alice's refresh token will be associated with Mallory's session.	When the term *user id* in the dev guide is interpreted as the user's Live ID	Dev guide revision
A5 (W)	In callAuthenticateAsyncFromMal, assume (app_id == _MalAppID user == _Mallory)	Alice's access token or Code for FooApp is obtained by $MalApp_C$.	When a client allows automatic login or one-click login	See Section 5.2.3
A6 (W)	Assume FooAppC always logs in as Alice, i.e., the first argument of dialog_oauth is _Alice.	Alice's session will be associated with Mallory's user ID and access token.	When request forgery protection for app logon is ineffective	Notifying developers

Table 3.1: Critical implicit assumptions uncovered using explication

SDK	Assumption	Impact	Proposed Fix
B1 (FB)	Result of getAccessToken returned to client is not equal to getApplicationAccessToken()	App *access_token* is added to the to the knowledge pool.	Develop checker to examine the traffic from $FooApp_S$
B2 (FB)	In dialog_oauth, assume FooApp.site_domain != Mallory_domain	Alice's *access_token* or *Code* for FooApp is obtained by Mallory.	Automatically scan if the *Site Domain* is properly set
B3 (FB)	Before $FooApp_C$ sends a request, assume request.signed_request.userId == _Alice	Alice's session is associated with Mallory's ID and *access_token*.	Enhance dev guide to require runtime check on $FooApp_C$
B4 (LC)	In HandleTokenResponse, assume auth_token.app_ID == _foo_app_ID	Alice's *authentication_token* for MalApp will be used by Mallory to log into FooAppS as Alice	Develop checker to scan if auth_token signature is verified.
B5 (LC)	In constructRPCookiefromMallory, assume RP_Cookie.access_token.user_ID == RP_Cookie.authentication_token.user_ID	Alice's ID associate with Mallory's access token, or vice versa	Enhance dev guide to require runtime check on $FooApp_S$

Table 3.2: Other implicit assumptions uncovered using explication

holds. By induction, the verified properties hold for an infinite number of iterations. Rewriting the three models in Boogie took 14 person-days of effort, including a significant portion on specifying appropriate loop invariants. The Boogie modeling did not find any serious case missed in the Corral modeling, but provides a higher level of confidence for the verification.

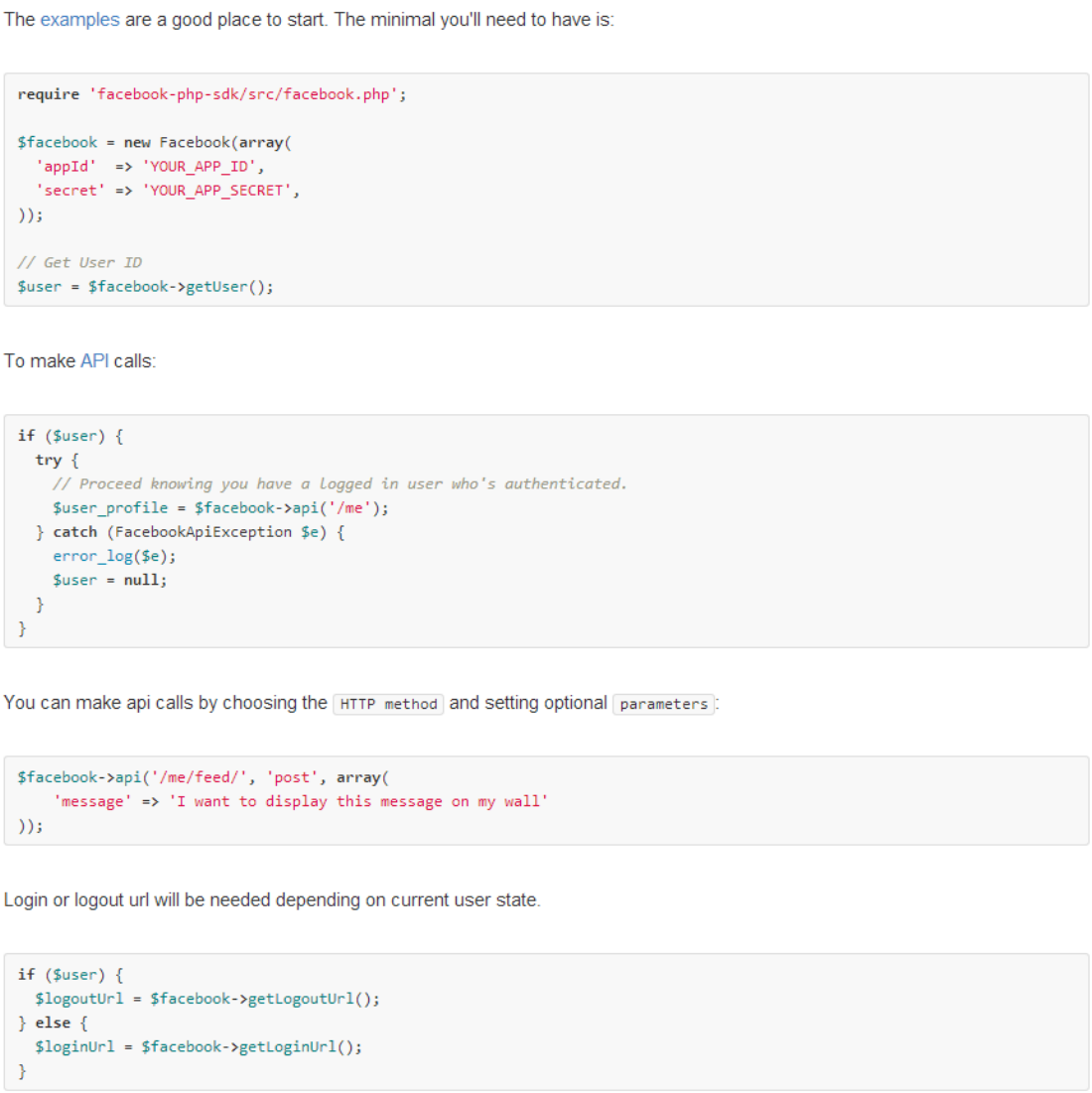
Mapping added assumptions to the real world. We manually examined each assumption added to assess whether it could be violated in realistic exploits. This effort requires thinking about how apps may be deployed and executed in real-world situations. Table 3.1 summarizes the assumptions uncovered by our study that appear to be most critical. These assumptions can be violated in the real world, and the violations result in security compromises. Based on our experience in communicating with SDK providers, finding realistic violating conditions is a crucial step to convincing them to treat the cases with high priority. This step requires extensive knowledge about systems, and does not appear to be easily automated. We describe these assumptions in more detail in Section 3.6. Table 3.2 lists some assumptions uncovered that, if violated, would also lead to security compromises. But, unlike the assumptions in Table 3.1, we have not found compelling realistic exploits that violate these assumptions.

3.6 Exploit opportunities

This Chapter explains how a real world attacker can exploit an application that is missing the critical assumptions in Table 3.1. These facts show how the SDK's security assurance depends on actual system behaviors and app implementations, illustrating the importance of explicating the underlying assumptions upon which secure use of the SDK relies.

3.6.1 Facebook SDK

Assumptions A1, A2, A3, and A6 listed in Table 3.1 concern the Facebook PHP SDK.


The examples are a good place to start. The minimal you'll need to have is:

```
require 'facebook-php-sdk/src/facebook.php';

$facebook = new Facebook(array(
  'appId'  => 'YOUR_APP_ID',
  'secret' => 'YOUR_APP_SECRET',
));

// Get User ID
$user = $facebook->getUser();
```

To make API calls:

```
if ($user) {
  try {
    // Proceed knowing you have a logged in user who's authenticated.
    $user_profile = $facebook->api('/me');
  } catch (FacebookApiException $e) {
    error_log($e);
    $user = null;
  }
}
```

You can make api calls by choosing the `HTTP method` and setting optional `parameters`:

```
$facebook->api('/me/feed/', 'post', array(
    'message' => 'I want to display this message on my wall'
));
```

Login or logout url will be needed depending on current user state.

```
if ($user) {
  $logoutUrl = $facebook->getLogoutUrl();
} else {
  $loginUrl = $facebook->getLoginUrl();
}
```

Figure 3.7: Facebook PHP SDK usage instructions

Assumption A1. This assumption states that the cookie associated with Alice's client must match Alice's session ID. Figure 3.7 is a screenshot of the usage instructions given in the README file[1] in the Facebook PHP SDK. It seems straightforward to understand: the first code snippet calls getUser to get the authenticated user ID; the second snippet demonstrates how to make an API call to retrieve further information; and the third snippet toggles between login and logout state, so that an authenticated session will get a logoutURL and an anonymous session will get a loginURL in the response.

The SDK's implementation for the getUser method is very simple. It calls the getUserFromAvailableData function shown in Listing 3.1. There are two statements calling setPersistantData, which is to set a PHP session variable denoted as _SESSION['user_id']. This action belongs to binding operations because it associates the user's identity with the session, which may affect the predicate that we define against association violations — specifically, if Alice's user ID is assigned to the _SESSION['user_id'] of Mallory's session, it would allow Mallory to act on FooApp_S as Alice. Since the session ID is a cookie in the HTTP request, the assertion depends on how the client runtime handles cookies.

Normally, because of the same-origin-policy of the client, cookies attached to one domain are not attached to another. However, the policy becomes interesting when we consider a cloud-hosting scenario. In fact, Facebook's developer portal makes it very easy to deploy the application server on Heroku, a cloud platform-as-a-service. Each service app runs in a subdomain under herokuapp.com (e.g., FooApp_S's subdomain runs as foo.herokuapp.com). Of course, Mallory can similarly run a service on behalf of mallory.herokuapp.com.

The standard cookie policy for subdomains allows code on mallory.herokuapp.com to set a cookie for the parent domain herokuapp.com. When the client makes a request to foo.herokuapp.com, the cookie will also be attached to the request. Therefore, if Alice's client visits the site mallory.herokuapp.com, Mallory will be able to make the client's cookie hold Mallory's session ID. Thus, FooApp_S incorrectly binds Alice's user ID to Mallory's session.

In response to our report, Facebook developed a countermeasure for the SDK. Instead of accepting an old session ID value in cookie, it always generates a new session ID (unknown to Mallory) every time a client is authenticated. Facebook offered us a bounty three times the normal Bug Bounty amount for reporting this issue, as well as the same award each for Assumptions A2 and A3 discussed next.

Assumption A2. This assumption is a case in which Corral actually discovered a valid path for violating an assertion completely unexpected to us. The path indicated that if a PHP page on FooApp_S only calls getUser (e.g., only has the first code snippet from Figure 3.7), Mallory is able to bind her user ID to Alice's

[1] Taken from https://github.com/facebook/facebook-php-sdk/blob/master/readme.md

session. The consequence is especially damaging if the session's *access_token* is still Alice's. Corral precisely suggested the possibility (see Table 3.1): if there is a *signed_request* containing Mallory's user ID, then the first setPersistentData call will be made, followed by a return. The method sets _SESSION['user_id'] to Mallory's ID without calling getAccessToken, which would otherwise keep the *access_token* consistent with the user ID. Therefore, the association between the user ID and the *access_token* could be incorrect. The session will operate as Mallory's account using Alice's *access_token*. After investigating our report about this, Facebook decided to add checking code before processing the signed request to the SDK to avoid the need for this assumption.

Assumption A3. This assumption requires that any PHP page that includes the first and fourth snippet in Figure 3.7 must also include the second snippet in between. This assumption is discovered when we model getAccessToken, as shown in Listing 3.3. We realized that in Facebook's authentication mechanism there are two subcategories of access token: 'user' *access_token*, as described earlier in this dissertation, and 'application' access token, which is provided to a web service for a number of special purposes, such as publishing instances of 'secure Open Graph actions'. In fact, the app secret can be derived solely from the application access token, so it is a serious authorization violation if Mallory can obtain it.

Listing 3.3: Facebook PHP SDK getAccessToken function

```
1 public function getAccessToken() {
2         ...
3         $this->accessToken= $this->getApplicationAccessToken();
4         $user_access_token = $this->getUserAccessToken();
5         if ($user_access_token) {
6                 $this->accessToken=$user_access_token;
7         }
8         return $this->accessToken;
9 }
```

Method getLogoutUrl in the fourth snippet in Figure 3.7 constructs a URL to send back to the client. The URL contains the result of getAccessToken. To obtain the application access token, Mallory only needs to send a request that hits a failure condition of getUserAccessToken, which prevents `$this->accessToken` from being overwritten in the bold line in Listing 3.3. We confirmed that this can be done by supplying an invalid *Code*in the request.

Interestingly, getAccessToken is also called by getUser in the first snippet in Figure 3.7. If a PHP page includes the second snippet after the first, the supplied access token will be used to call a REST API. When

it is an application access token, the API will raise an exception, which foils the exploit. That is why the second snippet is required before the fourth.

In response to our report on this issue, Facebook modified the SDK so that getLogoutUrl now calls getUserAccessToken instead of getAccessToken, thus avoiding the need for developers to satisfy this assumption.

Assumption A6. This assumption requires that the user on FooApp_C should not be Mallory. Otherwise, Mallory would be able to associate its *access_token* and user id with Alice's session. This is in fact requiring the application server to thwart any adversary performing session fixation attack by correctly implementing the cross-site request forgery defense. Moreover, this association violation can be particularly damaging when the service app has its own credential system, and supports linking a Facebook ID to Alices password-protected account. Once the linking can be done in the session, Mallory will be able to sign into Alice's account using Mallory's Facebook ID. We confirmed that among the 21 applications displayed on Facebook showcase page, 14 service apps which violate the assumption, 6 of them support linking, and thus allowing Mallory to login as Alice in the future. We reported this issue to Facebook, who undertook efforts of notifying app and website developers.

3.6.2 Microsoft Live Connect

Assumption A4. The sample code given by Microsoft Live Connect documentation is essentially a program skeleton, with comment blocks for app developers to implement. The core of the problem lies in the following function, whose implementation is empty except for a comment:

Listing 3.4: Microsoft Live Connect PHP SDK saveRefreshToken function

```
1 function saveRefreshToken($refreshToken) {
2         // save the refresh token associated with the
3         // user id on the site.
4 }
```

This is precisely what we call a binding operation. The refresh token is the input parameter, but it is not clear where the user id comes from. Within the scope of this function, the straightforward way to obtain this ID is from a cookie called AUTHCOOKIE, which contains the user's Live ID. However, if the application is implemented this way, the SDK's logic is not sufficient to ensure that Alice's refresh token is always associated with her user ID. In fact, we show that Alice's refresh token will be associated with Mallory's user ID if the attacker lures Alice (who has already logged into her Live Connect account) to visit a carefully crafted page. This will further lead to Alice's long-lived *access_token* and *authentication_token* be exposed to Mallory.

We sent this proof-of-concept exploit to Microsoft. The team replied us and the comment in saveRefreshToken function has been revised to "save the refresh token and associate it with the user identified by your site credential system." This change was also made in the ASP.NET version of the sample code.

3.6.3 Windows 8 Modern app SDK

Assumption A5. Windows 8 Modern app SDK is used by Windows 8 app developers who wishes to integrate OAuth-based identity providers. In the SDK, the function of interest here is authenticateAsync. Figure 3.8 illustrates the data passing through this function when the app requests an *access_token*. The key observation is that the client does not conform to the same-origin policy, because the 302 response is in the context of https://facebook.com, while on Windows 8, an app runs in its own specialized domain, ms-appx://packageID. Without the same-origin policy, we were unable to see why Alice's *access_token* for FooApp is guaranteed to be passed to FooApp_C, not MalApp_C. To test this, we implemented a successful proof-of-concept attack to obtain Alice's *access_token*.

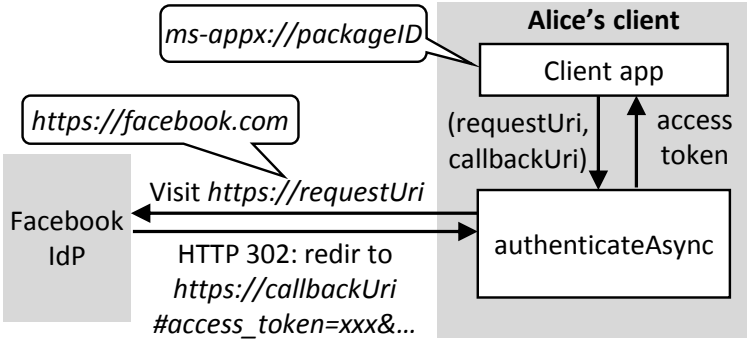


Figure 3.8: The data flow of authenticateAsync function in Modern app SDK

We reported this finding to Microsoft and Facebook. Microsoft considered it "a shortcoming of the OAuth protocol and not specific to our implementation.". Facebook pointed out that when authenticateAsync is called, an embedded browser window (usually called a WebView) is always prompted for Facebook password. This lowered the severity of the attack. We consider this a shaky security basis: if authenticateAsync allows a user to login automatically or with one click without using a password in the future, the basis will become invalid.

We investigated how SDKs on other platforms handle the data passing, and found a similar issue with the Facebook SDK for Android. However, on Android, there is a mechanism to skip the password prompt to get the *access_token* automatically. In response to our report, Facebook is developing a fix for its Android SDK.

Assumption	Number of test apps	Vulnerable cases
IE	27	21
A1	7	6
A6	21	14

Table 3.3: Test results for assumptions IE, A1 and A6

3.6.4 Testing Real-world Applications

To see if the discovered implicit assumptions are indeed likely to be missed by real-world applications, we selected three assumptions that are amenable to checking — A1, A6 in Table 3.1 and the one in the illustrative example in Chapter 3.2 (denoted as assumption IE). We subsequently describe our test application selection, testing approach and results.

Test application pool. we downloaded high-ranked applications from Windows 8 Store for assumption IE and did Google queries for relevant keywords for assumption A1 and A6. We manually confirm that the test applications supports Facebook/Microsoft Live Connect login.

Testing approach. Checking applications requires the ability to manipulate traffic to and from the IdP and the testing application's server. We use the Fiddler proxy [37] which may act as a Man-In-The-Middle and support modification to decrypted SSL traffic. For assumption IE, we manually login to target application through SSO as Mallory, requests an *access_token* Facebook, however, before forwarding this to the application, we replace it with Alice's *access_token* intended for another malicious application and check if the impersonation attack has succeeded. This process is described in more details later in Chapter 4.1. For assumption A1, if the testing application server's hostname is foo.a.com, we assume the proxy controls another hostname mallory.a.com. The test follows the steps described earlier in this section. Eventually the proxy checks if the authentication is successful, but the associated session ID is identical to that of Mallorys session on foo.a.com. For assumption A6, the proxy observes the HTTP request that FooApp_C sends to Facebook. It attempts to locate a field named *state*, which is an parameter supported by Facebook to prevent request forgery for login. The proxy then replaces the OAuth credential and the state field with the ones that Mallory's session owns. After sending the request, the proxy checks whether Mallory can associate her Facebook ID with Alice's session, and reports a violation if it sees a successful server response.

Test results. Table 3.3 shows the test results. The affected application percentages are very high for all vulnerabilities. We think this might be due to several reasons. For assumption IE, as of the time we carried out the tests, Windows 8 Store was just opened to public and it is conceivable that applications are less mature. For assumption A1, all applications using Facebook PHP SDK that is hosted on a cloud

platform would be affected, unless the developers included additional logic to defeat the attack (intentionally or unintentionally). For A6, it may appear to an average developer that the parameter *state* has nothing to do with the SSO functionality and can be safely omitted, therefore causing many vulnerable cases.

Chapter 4

SSOScan: Automatic Vulnerability Scanner[1]

Chapter 3 demonstrates that service provider's documentation and SDKs are far from perfect and often missing implicit security-critical assumptions, and an exploratory study in Chapter 3.6.4 shows that many application developers are prone to miss such assumptions which may cause serious vulnerabilities.

To better understand and mitigate these risks, we present our design and implementation of SSOScan in this Chapter, an automated vulnerability checker for applications using SSO. SSOScan carries out simulated attacks automatically by observing the application and monitors the application behavior through this process to determine vulnerability status. Compared to the manual studies we presented in Chapter 3.6.4, SSOScan is capable of successfully checking 80% of high-profile web applications in a short period of time per test without requiring any human interaction.

SSOScan prototype focuses on Facebook as the identity provider, but our approach is generic and could be applied to check other identity providers as well. It takes a website URL as input, determines if that site uses Facebook SSO, and automatically signs into the site using Facebook test accounts and completes the registration process when necessary. Then, SSOScan simulates several attacks on the site while observing the responses and monitoring network traffic to automatically determine if the application is vulnerable to any of the tested vulnerabilities. In addition to Single Sign-On services, many of the automation heuristics and techniques could also be adapted to scan for vulnerabilities in integrating other security-critical services such as online payments and file sharing APIs.

[1]The contents of this chapter is based on the paper: SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities [38].

For the rest of this Chapter, we first show in Chapter 4.1 the human effort involved in manually checking an example application for the assumption IE vulnerability status mentioned in Chapter 3.2. This gives readers a general idea of what actions SSOScan needs to automate. Then, we discuss some closely related works in this field in Chapter 4.2, and the vulnerabilities SSOScan aims to detect in Chapter 4.3. In Chapter 4.4 we explain the design and implementation of SSOScan, and present the large-scale study results we obtained by using SSOScan to check the top 20,000-ranked websites according to Quantcast in Chapter 4.5. Our exploration experiments and results to improve SSOScan's speed and automation success rate are given in Chapter 4.6. Finally, we discuss the vendor responses we got and possible deployment scenarios for SSOScan in Chapter 4.7.

4.1 A Manual Scanning Example

For reader's convenience, we recap the vulnerability previously mentioned in Chapter 3.2. The vulnerability exists because the application uses *access_token* as the only credential to authenticate users through Facebook SSO, and *access_token* is not tied to a particular application. Denoting Mallory as the malicious party, Alice as the victim, and Foo as the vulnerable application, this vulnerability can be exploited using the following steps: (0.0) Alice and Mallory both own accounts at IdP. (0.1) Mallory lures Alice to use Mallory's rogue application, through which Mallory obtains Alice's *access_token* for the rogue application as a preparation stage for the attack. (0.2) We assume Alice uses Foo application previous to the attack. (1) Mallory performs necessary actions to log in to Foo using Mallory's own IdP account. (2) As the IdP returns the *access_token* to Mallory's browser, Mallory swaps it with the previously obtained *access_token* from Alice. (3) Mallory checks with the browser to see if Foo has indeed authenticated Mallory as Alice, i.e. if Foo is vulnerable to this impersonation attack.

Playing as both the victim and the perpetrator, these steps are essentially what we did to test the 27 applications in Chapter 3.6.4, and here we list the actual actions performed to test the example `http://www.espn.go.com` web application.

- **Test account registration.** In step (0.0), we register two test accounts at Facebook for both Alice and Mallory.
- ***Access_token* stealing.** In step (0.1), we obtain Alice's *access_token* for Mallory's application by visiting a crafted URL and getting the token in the Facebook response.
- **Clicking "login through SSO" button(s).** In step (0.2) and (1), we need to click the correct button(s) to initiate the SSO process. For ESPN.com, this is to click the 'sign in' button in the

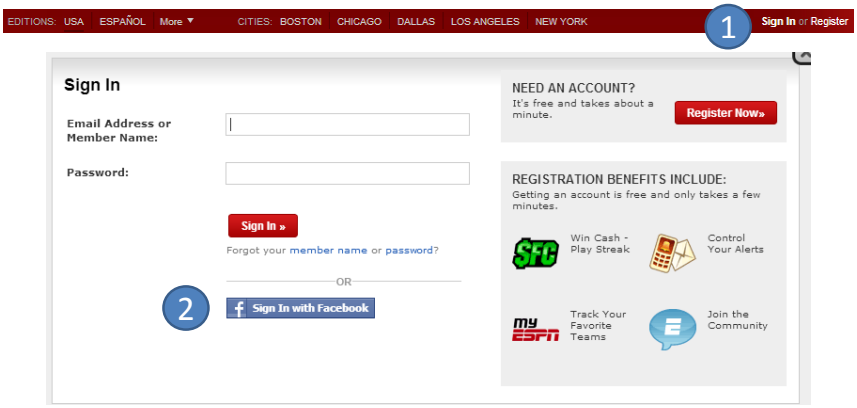


Figure 4.1: ESPN login buttons on homepage

upper-right corner, and then the 'login with Facebook' button in the pop-up layer , as shown in Figure 4.1.

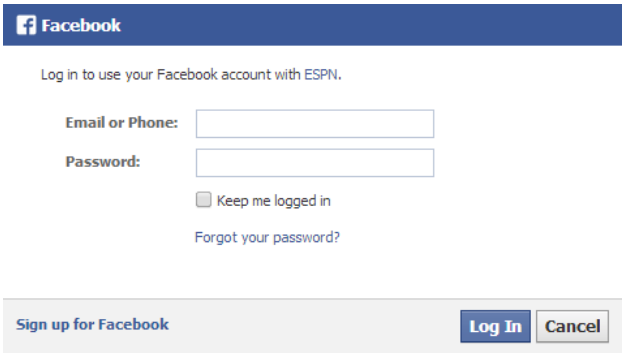


Figure 4.2: Typical Facebook login form

- **Automatically fill in Facebook Credentials.** In step (0.2) and (1), after the SSO process is triggered, we need to input test account information and log in to Facebook. This process does not vary much for every application, and a typical Facebook login form is shown in Figure 4.2.

- **Fill in registration forms if necessary.** In step (0.2) and (1), after the SSO process has completed, we need to fill in additional registration forms if the application so asks. Certain user information may have already been pre-populated from the test account Facebook profile, as is shown here in Figure 4.3. For ESPN.com, we simply need to click 'finish' button to complete the registration process.

- **Manipulate traffic.** In step (2), after IdP returns the token, we need to replace it with another token. This is done by manually identifying the response token in Fiddler proxy and overwrite it. We show an example response in Figure 4.4 with the string of interest highlighted.

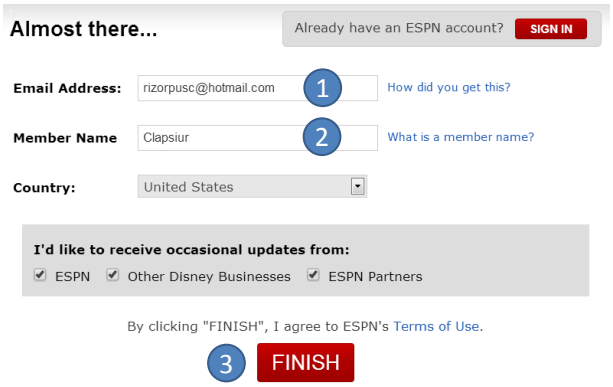


Figure 4.3: ESPN register form after SSO process

```
HTTP/1.1 200 OK
Pragma: no-cache
Strict-Transport-Security: max-age=345600
X-Frame-Options: DENY
...
Content-Length: 1599

<script type="text/javascript">var message =
"cb=f28c37ba0ed1c6c&domain=r.espn.go.com&origin=https\u00253A\u00252F\u00252Fr.e
spn.go.com\u00252Ff6bb1024f243d2&relation=opener&frame=f23834b4732503a&access_
token=CAABqGSEUW3UBAAvM27DAZC1b0xp7l1ph0jDcZAAd8uJSH35cRhAqImMd5cmMbc
LRobZCWarTcZA1Oiv0p3VJBkNoCGZC4dSND6k5QTZCLpkPV8xwNPmk64uZBnAG7VpTdPR9
KaaXZCv2f54H7h5ZAv3nmwtiX4d77AQThxZApSls3aZBKQkPtRFqZBuBKFlPZBRjR8oQZD&ex
pires_in=4355&signed_request=...&base_domain=go.com", ...;</script>
```

Figure 4.4: Using Fiddler proxy to tamper *access_token*

- **Detect session identity.** In the final step (3), after the token swap, we need to determine if the attack is successful. Identifying the currently logged-in user is relatively easy for a human, for example, the upper-right corner of the web page (Figure 4.5) shows the user name of current session, which acts as a good indicator.

For all the above steps, the first two is necessary to only perform once for all tests; However, all the rest of the steps are required per test and their actions vary across target applications. SSOScan attempts to automate these steps in a timely fashion, thus enabling large-scale scanning of vulnerabilities in high-profile websites.

Figure 4.5: Test account first name displayed after SSO process

4.2 Related Work

Our work builds on extensive previous work on automatically testing applications for vulnerabilities. We briefly describe relevant approaches next, as well as previous works that analyze vulnerabilities in SSO services.

Program analysis. Program analysis techniques such as static analysis [39] and dynamic analysis including symbolic execution [40,41] automatically identify vulnerabilities with fast testing speed and good code coverage. Runtime instrumentation techniques such as taint tracking [42] and inference [24] also help to safeguard sensitive source-sink pairs. Sun et al. [43] used a symbolic execution framework to detect logic vulnerabilities in e-commerce applications. However, these techniques require white-box access to the application (at least at the level of its binary), which is not available for remote web application testing. Automated web application testing tools that work on the server implementation [44–46] do not apply to large-scale vulnerability testing well. They either require access to application source code or other specific details such as UML or application states. For our purposes, the test target (application server implementation) is only available as a black box.

Oracle-based security testing. Penetration testing is widely used to check applications for vulnerabilities [47, 48]. The tester analyzes the system and performs simulated attacks on it, often requiring substantial manual effort. More automated testing requires an oracle to determine whether or not a test failed. Sprenkle et al. developed a difference metric by comparing two webpages based on DOM structure and n-grams [49] and improved results using machine learning techniques [50]. SSOScan also requires an oracle (Chapter 4.4.3) to determine session identity. For our purposes, a targeted oracle works better than their generic approach.

Automated GUI testing. SSOScan is also closely related to automated GUI testing. The GUI element triggering approach we take shares some similarities with recent works to simulate random user interactions on GUI element to explore application execution space on Android system [51], native Windows applications [52], and web applications [53, 54]. Their common goal is to explore app execution space efficiently to discover buggy, abnormal or malicious behavior. By contrast, our goal is to drive the application through a particular SSO process rather than explore its execution space. Further, we need the tests to proceed fast enough for large-scale evaluation. Since each simulated user interaction with the web application involves round-trip traffic and a non-trivial delay to get the response, our primary focus is to develop useful heuristics to quickly prune search space before triggering any user interactions.

SmartDroid [55] and AppIntent [56] both aim to recover sequences of UI events required to reach a particular program state or follow an execution path obtained from static analysis. These approaches target Android applications and rely on client-side information that is not available for our web application scanning

tool, where the necessary state only exists on the (inaccessible) server side.

Human cooperative testing. Off-the-shelf testing tools like Selenium [57] and TestingBot [58] can be used to discover bugs in web applications under developers' assistance. These tools replay user interactions based on testing scripts that are manually created by the application developer. BugBuster [59] offers some automatic web application exploration capabilities, but still does not understand the application context enough to perform any non-trivial actions such as those involving authentication and business logic.

To reduce developer effort, Pirolli et al. [60], Elbaum et al. [61], and the Zaddach tool [62] show promising results by collecting interactions from normal users and replaying them to learn application states and invariants for vulnerability scanning. These works do not require extra manual effort from developers to write testing script or specify user interactions. However, one potential problem these works fail to address is user's privacy concerns when submitting every interaction. This could be especially sensitive when the actions involve passwords or payments. SSOScan avoids this problem and is complementary to this line of work — in most cases, SSOScan attempts to scan applications in a fully automatic fashion and does not require traces from any party. When SSOScan fails, traces submitted from the users may guide SSOScan to complete the scan.

SSO security. We have already covered some works on SSO-related security analysis in Chapter 3.1, however, we want to point out several additional closely related works in SSO application scanning.

Integuard [63] and AuthScan [64] have similar goals with SSOScan. Integuard infers invariants across requests and responses and uses them to perform intrusion detection on future activities. AuthScan [64] is an automated tool to extract specifications from SSO implementations by using both static program analysis and dynamic behavior probing. Our goals differ in that we focus on detecting specific vulnerabilities rather than generic ones. This enables us to establish clear automation goals and build well-defined state machines for the scanner, and removes the uncertainties the previous works incur when inferring invariants or modeling unknown functions. The drawback is our approach relies on knowledge of particular vulnerabilities. For many integrated web services, including SSO, many vulnerabilities are known or can be obtained using systematic explication as described in Chapter 3.

Brahmastra* [65] is an automated testing tool for SSO vulnerabilities on Android platform. Rather than finding the GUI element of interest, it leverages program analysis and reverse engineering of application binary to find the function that triggers Facebook SSO process. It then calls that function and force the application to initiate the authentication process. Their approach takes advantage of the fact that entire

*Work published in the same conference as ours.

client code is visible for mobile applications, however, it achieves a relatively low success rate (39%) in testing applications.

4.3 Targeted Vulnerabilities

In addition to the *access_token* vulnerability discussed in Chapter 4.1, SSOScan aims to scan for three other vulnerabilities. The four vulnerabilities can be categorized into two general types.

Credential Misuse. If an application uses *access_token* to authenticate users, we classify the application as misusing credentials. However, sometimes even the developers chose the correct OAuth credentials to use, their application still ends up with a vulnerable implementation. One way this happens is when information decoded from a *signed_request* is used but the signature is never checked using the *app_secret*. The attack to exploit this vulnerability is similar to the *access_token* simulated attack, except that Mallory needs to replace the *signed_request* in addition to *access_token*.

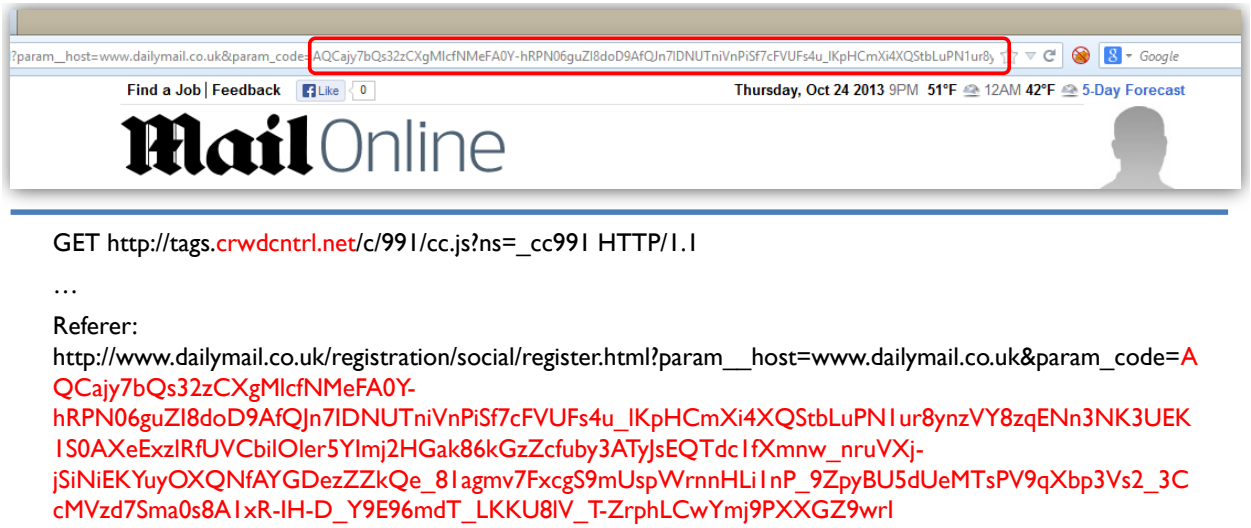


Figure 4.6: OAuth *Code* leaked through the Referer header

Credential Leakage. The OAuth credentials can be accidentally leaked to a potentially malicious third-party, if the application developer does not pay attention. When the Facebook OAuth landing page contains third-party content, requests to retrieve those contents will automatically include OAuth credentials in the Referer header, which leaks them to the third-party. In addition, credentials can be exfiltrated by third-party scripts if they are present in the page content. If a malicious party is able to obtain these credentials, it could carry out impersonation attacks or perform malicious actions using permissions the user granted the original application, such as posting on user's timeline or accessing sensitive information.

4.4 Design and Implementation

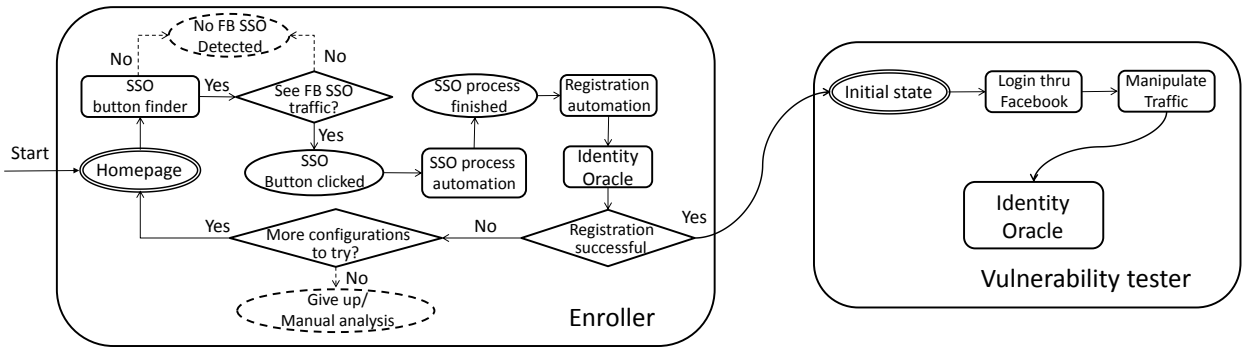


Figure 4.7: SSOScan components overview

SSOScan consists of two main parts: the Enroller and the Vulnerability Tester, as shown in Figure 4.7. Ovals represent testing states, curved rectangles represent different modules in our tool, and diamonds represent control flow decisions.

Left of Figure 4.7 describes the workflow of the Enroller. The Enroller automatically registers two test accounts at target web application using Facebook SSO login. Given a target web application A, our tool first removes all cookies from the browser and navigate to A. A short delay after the page has fired its onload event, the SSO button finder (Section 4.4.1) analyzes the DOM and outputs the most likely candidate elements for SSO button. The Enroller then simulates clicks on those elements, monitoring traffic to listen for the Facebook SSO traffic pattern.

When traffic to Facebook SSO entry point is captured, SSOScan automatically logs into Facebook and grants the requested permissions to the application. OAuth credentials returned from Facebook are stored for future use. An important and challenging step after the SSO process is to automatically complete the registration form when applicable. SSOScan combines heuristics with random inputs to fill in forms (Section 4.4.2) and locate and click the submit button. Should this process fail, the Enroller would try again using a different option (Section 4.6.1) and repeats the process until success or giving up after trying all available options.

After successfully enrolling both test accounts at target application, the Vulnerability Tester (Chapter 4.4.4) performs simulated attacks on the application and monitors the traffic and its behavior to determine vulnerability status. An identity oracle (Chapter 4.4.3) is necessary for both the Enroller and the Vulnerability Tester due to different purposes.

4.4.1 SSO Button Finder

Using Figure 4.1 as an example, the SSO button finder needs to locate the upper-right corner for 'sign in' and then the 'login with Facebook' button in the middle of the page. To automate this, SSOScan first extracts a list of qualifying elements from all nodes in an HTML page, and then extracts content strings from such elements. The Button Finder relies on the assumption that developers put one of a small pre-defined set of expected strings in the text content or attributes of the SSO button, and our evaluation results (Section 4.5) confirm that this assumption is nearly always valid. SSOScan computes a score for each element by matching its content with regular expressions such as `[Ll][Oo][Gg][liOo][Nn]` which indicates its resemblance to "login". SSOScan forms a candidate pool consisting of the top-scoring elements and triggers clicks on them. This process can be illustrated using Figure 4.8. We defer the details such as the heuristic choices SSOScan uses to filter elements and compute scores to Section 4.6.

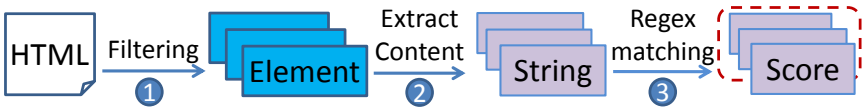


Figure 4.8: SSO Button Finder workflow

Once the login button has been found and the traffic to Facebook's OAuth endpoint has been captured, automating logging into Facebook is straightforward. Upon detection of Facebook login DOM structure, we automatically fill in the username/password field using test account information. SSOScan also grants all requested permissions to the application to facilitate registration.

4.4.2 Completing Registration

The required interactions to complete the registration process after single sign-on vary significantly across web applications. They range from simply clicking a submit button (e.g., Figure 4.3, in which all input fields are pre-populated using information taken from the SSO process), to very complicated registration processes that involve interactively filling in multiple forms and possibly CAPTCHA solving.

SSOScan attempts to complete all forms on the SSO landing page by leaving pre-populated fields untouched and processing the remaining inputs in the order of radios, selects, checkboxes and finally text inputs. We found this ordering to be very important to achieve higher automation success, as some forms may dynamically change what needs to be filled upon selecting different radio or select elements. Processing these elements first allows SSOScan to rescan for dynamically generated fields and process them accordingly.

For radio and select elements, SSOScan randomly chooses an option; for checkboxes, it simply checks all of them. For text inputs, SSOScan tries to infer their requirements using heuristics and provide satisfactory mock

values. Once all the inputs have been filled, the next step is to reuse the SSO Button Finder with different regular expressions and settings specifically designed to find submit buttons. After SSOScan attempts to click on a submit button candidate, it refers to the oracle 4.4.3 to determine if the entire registration process is successful.

4.4.3 Oracle

The Oracle analyzes the application and determines whether it is in an authenticated state, and if so, further identifies the session identity. This module is necessary for SSOScan to decide if a registration attempt is successful. It is also used by the Vulnerability Tester to determine if a simulated impersonation attack succeeds.

The key observation behind the Oracle is that web applications normally remove the original login button and display some identifying information about the user in an authenticated session. For example, after a successful registration many websites display a welcome message that includes the user's first name (Figure 4.5). Therefore, the Oracle works by searching the entire DOM and **document**.cookie for test account user information (e.g., names, email, or profile images) after the page has finished loading.

4.4.4 Vulnerability Tester

After the Enroller successfully registers two test accounts, control is passed to the Vulnerability Tester which checks the target application for the vulnerabilities described in Section 4.3. We use two different probing approaches to cover the five tested vulnerabilities: *simulated attacks* and *passive monitoring*.

Simulated Attacks. The two credential misuse vulnerabilities are tested using simulated impersonation attacks. We have already outlined the steps to manually carry out such attacks in Chapter 4.1. To automate it, SSOScan simply invokes existing components to complete logins. The only difference is that SSOScan will replace the OAuth credentials in Facebook's response with those previously obtained from Alice.

The attack is successful if Bob is able to login as Alice using the replaced credential. The Vulnerability Tester deems the site vulnerable if the Oracle determines that Alice is the logged in user after the simulated attack.

Passive Monitoring. The credential leakage vulnerabilities are detected using passive approaches. For brevity, we only explain how leaks through the Referer header are detected; the other leaks are detected similarly by observing network traffic and web page contents.

To check if an application leaks the user's OAuth credentials through the Referer header, SSOScan monitors all request data during the account registration process and compares each Referer header to OAuth

credentials recorded in earlier stages. If a match is found, SSOScan then checks if the requesting page contains any third-party content such as scripts, images, or other elements that may generate an HTTP request. SSOScan reports a potential leakage when credentials are found in the Referer header for a page that also contains third-party content.

Limitations. While SSOScan is able to automatically synthesize basic user interactions and analyze traffic patterns, it is not suitable for detecting all types of vulnerabilities. Specifically, it only works for vulnerabilities that can be checked by observing traffic or simulating predictable user events, and falls short if the vulnerability testing involves deep server-side application scanning or complicated interactions. For example, it is very hard to check for vulnerabilities caused by missing assumption A2 or A3 in Chapter 3.6 using an external tool with no awareness of the sites' implementation details or internal state. This type of vulnerabilities could instead be checked at the developer side using program analysis techniques.

4.5 Large-scale Study

We evaluated SSOScan by running it on the list of the most popular 20,000 websites based on US traffic downloaded from quantcast.com as of 7 July 2013. Of those 20,000 sites, 715 of the sites are shown as *hidden profile* (that is, no URL is given, thus excluded from our study).

We ran SSOScan on the remaining 19,285 sites in September 2013, and found that homepages of 1372 sites failed to load during two initial attempts (most likely due to either expired or incorrect domain name, server error, or downtime). We excluded these sites from our data set, leaving a final test dataset containing 17,913 sites.

Completing the tests took about three days, running 15 concurrent sessions spread across three machines. The average time to test a site is 3.5 minutes. We limited the maximum stalling time for each site on any one module to four minutes, and the overall testing time to 25 minutes per site. If this timeout is reached, SSOScan restarts and retries a second time before skipping it. We ran extra rounds on tests that failed or stalled during initial round until either the test is completed or the four rounds maximum limit has been reached. The extra rounds involved fewer sites (<10%) and took a week to complete running on one machine with 4 concurrent sessions.

4.5.1 Automated Test Results

Figure 4.9 presents results purely based on automatic tests run by SSOScan. SSOScan found a total of 1660 sites using Facebook SSO among the 17,913 sites (9.3% of the total). Figure 4.10 shows the number of

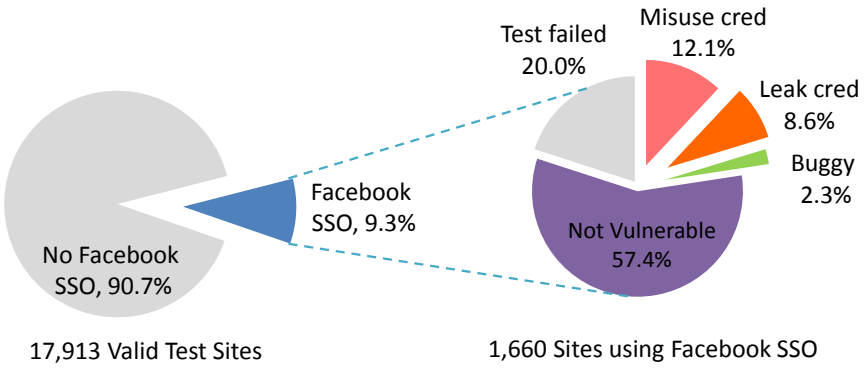


Figure 4.9: Large-scale study results overview

Facebook SSO supported sites, sites that misuse credentials, and sites that leak credentials distributed by site ranking. In Section 4.5.3, we report on our manual analysis on failed tests for sites ranked in the top 10,000.

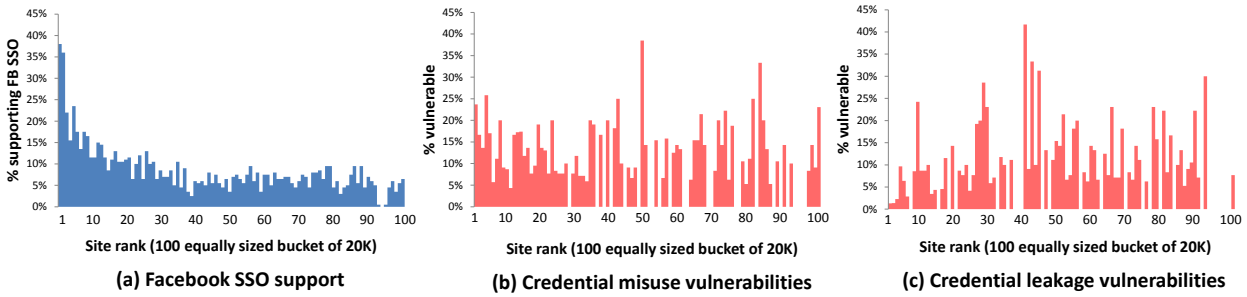


Figure 4.10: Facebook integration stats vs. sites ranking (each bin contains 179 sites, 1% of the total tested)

Facebook SSO integration. Figure 4.10 (a) shows that more popular sites are more likely to integrate Facebook SSO. Of the top 1000 sites, 270 (27%) of them include Facebook SSO, compared to only 52 out of the 1000 lowest-ranked sites in our dataset. This supports our believe that covering the top-ranked 20,000 websites is sufficient to get a clear picture of prevailing Facebook SSO usage since less popular sites are both less visited and less likely to use Facebook SSO.

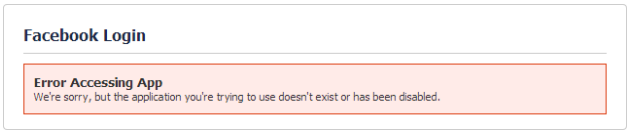


Figure 4.11: Facebook implementation error example

Faulty implementations. To implement Facebook SSO, an application must be configured correctly in the Facebook developer center. Using incorrect parameters to call the SSO entry point also result in errors that will prevent any user from authenticating to that application through SSO. Such cases, automatically

identified by SSOScan, were more common than we expected. The most popular errors include setting the application to 'sandbox' mode in the developer center for testing and development purposes (as shown in Figure 4.11), or providing a wrong application ID. SSOScan found 39 (2.3% out of 1660 sites that incorporate Facebook SSO buttons) sites that display visible Facebook SSO buttons but have implementations so buggy that no user could ever login using them. A possible explanation is that the buttons are there for SEO purposes and the developers never actually bothered to implement it, or the developers simply copied and pasted an SSO snippet customized for another application without ever testing it.

Vulnerability trends. We found 205 sites (12.1%) that misuse credentials (126 of which are misusing both access_token and signed_request) and 146 sites (8.6%) that leak Facebook SSO credentials (of which 72 sites are leaking through both referrer headers and URLs). A total of 345 sites (20.3%) suffered from at least one vulnerability, and 3 sites suffered from both credential misuse and leakage problems.

As shown in Figure 4.10 (b) and (c), the vulnerability rate does not appear to be strongly correlated with the site ranking. Of the 1000 highest-ranked sites, 60 of the 270 (22.2%) that support Facebook SSO are found to have at least one vulnerability. The vulnerability rate is 21.3% across all sites in the top 10,000 and 18.5% for sites ranked from 10,001 to 20,000. This indicates that development practices at larger companies do not appear to be more stringent (at least with respect to SSO) than they are at lower-profile sites.

Front-end Integration. There are three basic client-side methods to integrate Facebook SSO: a JavaScript SDK, a pre-configured widget, or a custom implementation. (We have no way to determine how the developers are integrating Facebook SSO at the back end.) We used SSOScan to aggregate front-end integration choices and compare them with vulnerability reports. Table 4.1 summarizes the results. Websites using client side SDKs and pre-configured widgets are more likely to misuse credentials (29.1% and 15.5% vs. 1.3% in non SDK/widget implementations). Our guess is that this is due to the way SDKs and widgets conveniently expose raw *access_token*, *signed_request*, or even user name Facebook ID values. This convenience may lead to the developers to neglect to check the signature and the intended audience of the credential. However, our results also show that websites using SDKs and widgets are better in hiding credentials (3.6% and 2.2% compared to 12.4% vulnerable rate in SDK/widget implementations). This is likely because such applications use the Facebook-provided landing page which has safe redirect URLs and no third-party content. Applications built this way are secure unless the developers explicitly add the credentials in the page content or URL.

Vulnerable application examples. We list two examples of vulnerabilities found by SSOScan here to illustrate the potential risks. Section 4.7 discusses our experiences reporting them to site owners.

ESPN.com. ESPN, as shown in the example in Chapter 4.1, is a sports and entertainment site that is the

16th-ranked US website. SSOScan found that espn.com is vulnerable to *signed_request* replacement attacks. In addition to serving news content, espn.com also hosts fantasy sports leagues, including some played with real money at stake. Damage caused by impersonation attacks can range from basic information stealing and comment posting to making changes to another player's fantasy teams.

Match.com. Ranked 118th on the list, Match.com is a popular online dating website. SSOScan revealed that match.com is also vulnerable to *signed_request* replacement attacks. To use match.com services, users need to provide sensitive information including their birthday, location, photos, personal interests, and sexual orientation. Impersonators will not only have access to this information, but also learn whom the victim is dating and possibly the time and location of the dates.

Fodors.com. Fordos is a travel advice website that is the 217th-ranked US site. Its redirection landing page contains *access_token* information along with some other third-party scripts in its content. The scripts come from various sources including quantserve.com, fonts.com, yahooapis.com, and multiple domains owned by Google. The permission Fordos requests includes user's basic information, email address, and more importantly, permission to post to user's wall on the his or her behalf. This means if the *access_token* is leaked to a malicious party, it can post to a user's Facebook wall without consent in addition to accessing the user's basic information.

4.5.2 Detection Accuracy

To evaluate the detection accuracy of SSOScan, we sampled test cases from all results (including sites reported to have no Facebook SSO support, secure and vulnerable cases) and manually examined them. We consider two types of mistakes — misreporting whether the site integrates Facebook SSO, and misreporting whether a Facebook SSO-enabled website is vulnerable.

Facebook Login Detection. SSOScan searches SSO button based on heuristics and cannot guarantee success for all websites. Indeed, it is not possible for anyone to determine with complete confidence if a website uses Facebook SSO by just browsing the site. To roughly measure how many Facebook SSO-enabled

Method	Number	Misuse	Leakage
SDK	578	29.1%	3.6%
Widget	132	15.5%	2.2%
Custom Code	950	1.3%	12.4%
All	1660	12.1%	8.6%

Table 4.1: Rate of credential misuse and credential leakage for different Facebook SSO front-end implementations

websites were missed by SSOScan, we randomly sampled the 100 sites that were reported by SSOScan to have no Facebook SSO support and manually examined them. To make the samples representative of the whole set, we picked one site out of every 200 sites ordered by their rank. From manually investigating these 100 sites, we could only find one site that included Facebook SSO but was missed by SSOScan. As we introduce later in Section 4.7, we also deployed SSOScan as a web service that is made available to use in our research group. The web service has received a total of 69 valid submissions so far and we have also manually examined the vulnerability reports.[1] We found four cases (5.8%) where a submitted site included Facebook SSO but SSOScan was not able to trigger it.

The sites that SSOScan fails to find Facebook login present unusual interfaces which our heuristics are not able to navigate to. Specifically, `oovoo.com` and `bitdefender.com` do not show any login button on its homepage, but instead the user needs to click a 'my account' button to initiate the login process. The `sears.com` site displays a login button on its homepage, but the SSO process is not initiated until the user interacts with the pop-up window three times, which exceeds the maximum click depths (two) in this evaluation. We have also seen one case (`coursesmart.com`) in which the login process is rather typical, but SSOScan still missed the correct login button (that button is scored the 4th highest while SSOScan only attempts to click the top 3 candidates.). Most of these issues may be addressed with more relaxed restrictions and more regular expression matching as described in Section 4.6. Finally, our prototype implementation is limited to English-language websites due to its string matching algorithm, but could be extended to include keywords in other languages.

SSOScan may also incorrectly conclude that a website supports Facebook SSO when it does not. We have seen sites (e.g., `msn.com`) that only use Facebook SSO to download user activities and display them on the page, but do not integrate their identity system with Facebook SSO. Although SSOScan is designed to skip searching on typical Facebook-provided social plugins and widgets, non-standard integration of such functionalities may rarely lead to false positives.

Oracle Correctness. The Oracle determines if a user has logged in by searching the response for identifying information. Information for the test account is chosen carefully to be unlikely to appear otherwise but to be close enough to real names to pass sanity checks. For example, the randomly generated name "Syxvq Ldswpk" was rejected by a small number of websites, but "Jiskamesda Quanarista" always passed sanity checks and only appeared in an authenticated session in all of our tests.

The Oracle checks the whole response for identifying information instead of only the DOM content to handle sites which only embed such information in first-party cookies after logging in. In some rare cases,

[1] These have mostly been sites suggested by people we have demoed SSOScan scan to, since the service has not yet been publicized. Hence, it is a small and non-representative sample, so not clear what we can conclude from this at this point.

these cookies could be issued even before SSOScan finishes registration forms. This means that before the Enroller searches for registration forms to fill in, the Oracle deems registration as unnecessary because it concludes that the application is already in an authenticated state. Although SSOScan is able to proceed and determine vulnerability status, the application never enters an authenticated state and the report might be inaccurate.

Trusted Third-Party Domains. For credential leakage vulnerabilities, SSOScan reports an application as vulnerable if it identifies visible credentials co-existing with *any* content or script that comes from *any* origin other than the host or Facebook. This could overestimate the vulnerable sites because the host may own other domains and serve content over them, which should not be considered untrusted. For example, content delivery networks and sub-company scenarios (e.g. `cnn.com` embedding content from `turner.com`, but CNN is owned by Turner) are common among popular websites.

4.5.3 Automation Failures

For about 19% of the top 10,000 tested site that include functional Facebook SSO, SSOScan is not able to fully automate the checking process. Figure 4.12 shows the distribution of rank of failed test websites.

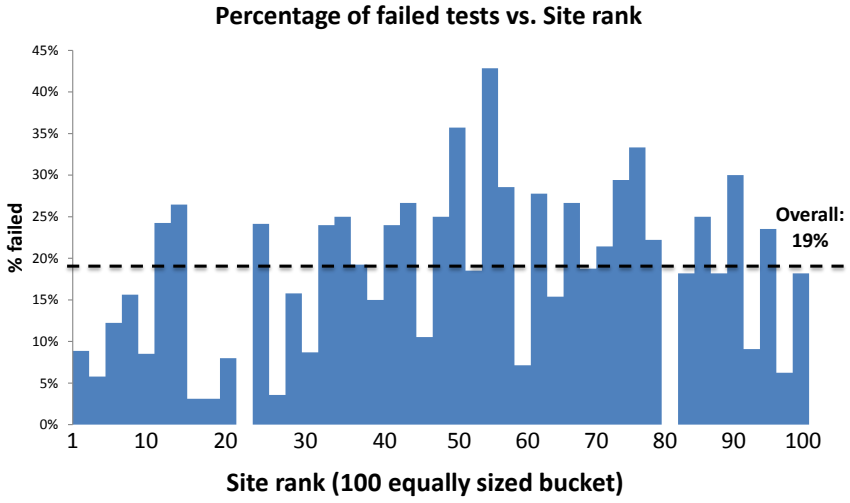


Figure 4.12: Failed tests rank distribution

To better understand the reasons why SSOScan fails, we manually studied all 228 failed cases reported by SSOScan for sites ranked in the top 10,000. We found that although 47 out of these 228 cases set their Facebook application configurations and SSO entry points properly, they never respond to credentials returned

Failure reason	Number	Percent
linking/subscription	51	28.1%
CAPTCHAs	34	18.8%
identity invisible to oracle	28	15.5%
atypical input elements	20	11.0%
atypical submit buttons	19	10.5%
email verification	10	5.5%
non-HTTPS submission forms	9	5.0%
other (e.g., timeouts)	10	5.5%
Total failures	181	100.0%

Table 4.2: Automation Failure Causes (top 10,000 sites)

by Facebook SSO, which means no users would be able to successfully log into these sites through Facebook SSO. Excluding these 47 left us a total of 181 failure cases.

Registration automation failure. By far, the most common reason for SSOScan to fail is due to complicated or highly-customized registration process. We found 43.7% of the sites that implement Facebook SSO still require users to perform additional actions to complete the registration (roughly evenly distributed by site popularity). SSOScan failed to complete registration on 143 (33.6%) of them. Table 4.2 shows the major reasons contributing to this failure ordered by their occurrences: 1) sites that require SSO users to link to an existing account or provide payment information to subscribe to the service; Currently SSOScan cannot handle the "linking" action: automatically registering a "traditional" account and perform the linking poses an out-of-scope challenge — doing so often requires solving CAPTCHAs[1]; 2) registration forms that include CAPTCHAs; 3) special input elements (e.g. `div`, `span` or `image` as opposed to `input`) cannot be found automatically, or special requirements for the input that cannot be fulfilled; 4) sites where the registration submission button cannot be located; 5) sites that requires users to confirm email addresses before continuing (usually this involves clicking a link in an email sent by the server to the user's email address); and 6) sites that insecurely send registration data using a non-HTTPS form which causes the testing browser to pop up a warning and stall.

Oracle confusion. SSOScan may also fail because the oracle reports failure (15.5%), which occurs when it detects the login button no longer exists after Facebook SSO but cannot identify the session identity. We manually analyzed such cases and found the biggest obstacle is that the application homepage does not include any identifying information at all. For example, instead of showing 'Welcome, {username}', it shows 'Welcome, customer', or simply 'Welcome', and the user name is only displayed when accessing the account

[1]On the contrary, most test applications (942 out of 973, see Section 4.5.3) do not ask users to solve CAPTCHAs when an account is created through SSO. This is a reasonable practice, since the user who is able to provide a valid Facebook account should have already passed Facebook's Turing test, and adding additional CAPTCHAs would be unnecessarily annoying to the users.

information page. In other cases, SSO authentication serves only a sub-service of the website such as its affiliated forum, but not the homepage which does not display any identifying information.

Others. During the testing, we have also seen a number of sites with extremely long loading time or inconsistent network latencies after Facebook SSO or upon navigating to certain pages. While the latency spikes can likely be resolved by re-running the tests, frequent long delays which accumulate to SSOScan's maximum timeout will always halt the automation process. For example, this happens when SSOScan accidentally triggers a browser confirmation dialog that requires user interaction, or asking users to stop a busy script execution.

4.6 Heuristics Evaluation

The ability of SSOScan to successfully complete the Facebook single sign-on and registration process depends on heuristics it uses to find buttons and fill in registration forms. Since each attempted button click involves a high-latency round-trip with the server, early pruning of search space and prioritization of elements is important for achieving successful completion within a reasonable amount of time.

In this section, we explain in detail about some of the heuristics SSOScan use to quicken the search and improve the results. We analyze the click data collected from the top 10,000 sites that use Facebook SSO (obtained from the large scale study) and show how tweaking the heuristics and significantly improve performance.

4.6.1 SSOScan Options

Each step in the automation process can be controlled by many options, including filters that can be enabled to eliminate candidate elements that are unlikely to be the correct target, weightings that adjust the contribution of different element properties to its score, and other behavior modifiers. The ones SSOScan used when running the Section 4.5 study are described below.

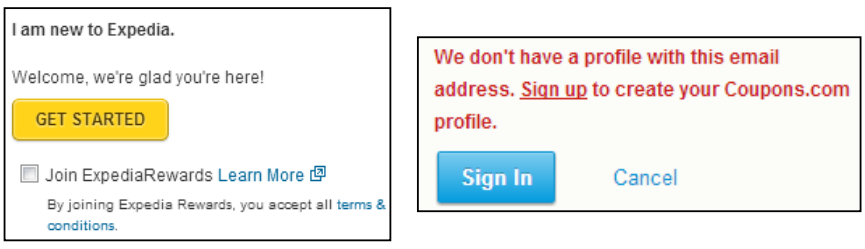


Figure 4.13: Example corner cases

Candidate rank. The button finder produces a candidate element list ranked by score. SSOScan will first attempt clicking on the highest-ranked element, but sometimes the correct element is ranked lower. This option controls how far down the list to attempt clicks until one is found that succeeds, or giving up. For Section 4.5's experiment, the lowest ranked element SSOScan clicks is the third.

Visibility filter. Most web sites only expect users to click on UI elements that are visible, so the button finder includes a filter that ignores all invisible elements (e.g., elements with zero height or width, shadowed in the background layer, or appear only when the user scrolls the initial screen position). We describe how we implement this in Chapter 4.6.2.

Position filter. We noticed that SSOScan sometimes gets distracted by a search box submit button when completing the registration form, even if it is able to correctly fill in the required information in all input elements. To eliminate these misclicks, the position filter ignores submit buttons which are displayed above any inputs, based on our observation that the target registration form is always the bottom-most form on the landing page.

Registration form filter. As mentioned earlier in Section 4.5.3, many websites provide two actions for the user after SSO is completed: 'create new account' or 'link an existing account'. The latter option requires the user to enter the user name and password of an existing account to finish the enrollment process. To avoid these, the registration form filter ignores a candidate submit button if its parent form contains only two visible text inputs, one has the meaning of 'name' or 'email' and the other is of type password, since such an element is most likely to be a submit button of a linking form.

Element content matching. SSOScan searches for elements whose labels are close to "login with Facebook" for SSO buttons by default. However, quite a few popular websites (e.g. `coupons.com`, right side of Figure 4.13) only allow users to "sign up with Facebook" first before logging in with Facebook. If the user has yet to do this, attempting to login with Facebook will produce an error. To handle this situation, SSOScan will search for elements with semantics similar to "sign up with Facebook" when it fails to register using the "login" buttons.

A filter may significantly reduce the number of misclicks. However, it may also occasionally exclude correct elements. For example, not every correct submit button is below all inputs (e.g., left of Figure 4.13, and `expedia.com`'s submit button would have been missed with the element position filter enabled).

Hence, SSOScan is designed to explore target sites using different option settings if enrollment does not succeed with the initial settings. It will continue to attempt to complete the enrollment process using different

settings until either the timeout threshold is reached or all configurations have been exhausted. SSOScan also avoids doing duplicate work by detecting if a click attempt has resulted in a previously visited or completely explored state, which is described in more details in Chapter 4.6.2.

4.6.2 Implementation

This chapter describes the techniques necessary to implement the previously mentioned filters.

Determining element visibility. As introduced in Section 4.6.1, SSOScan uses a filter that rejects elements which are not directly visible to the user. We implement the filter by using **document**.elementFromPoint API. For any given element E, we first get its current top-left corner coordinates, width and length. Then, **document**.elementFromPoint is called on ten points distributed equally along the diagonal inside this area. If more than five of the elements returned are either E itself or a child of E, E is considered to be on top. Depending on current strategy, SSOScan may call scrollIntoView to include searching elements displayed at lower sections of a document. Using **document**.elementFromPoint, however, has its limitations — it always returns the underlying canvas element when called at any point on that canvas, therefore onTopLayer always returns false for elements placed on canvases. We hope future browsers can fix this bug, but for now SSOScan relies on alternative option choices that does not ignore invisible buttons to handle such scenarios.

Avoiding futile and duplicate clicks. To further eliminate false positives in candidate lists as early as possible, SSOScan detects clicks which have no effect or have lead the web application into a previously visited state in the same attempt. Naïvely using URL or full **document**.body.innerHTML string to represent application state does not work well, because these information are not stable as variations exist across different requests and time (e.g. GET parameters and advertising content). Fortunately, getCandidates provides a representative feature naturally — the candidate list. To detect previous clicks that are futile, SSOScan records candidate information before a click happens (line 17), and if any candidate list computed for subsequent clicks is exactly the same as previously recorded, SSOScan considers previous click(s) as futile and fast forward to the next candidate. Similarly, SSOScan stores candidate information when the current state is fully explored, i.e. all candidates on the current page have been clicked or the number of attempts have reached a certain threshold, and if any future click lands on a page which matches the same set of candidates, SSOScan immediately moves on to the next candidate to avoid duplicate work.

To ensure SSOScan sees all candidate elements, we need to manually trigger event handlers upon selecting a particular option, and work around the same-origin policy to access elements in all iframes.

Triggering Event Handlers. Programatically changing values of option, checkbox and radio elements

using JavaScript does not trigger their onChange event handler. However, in practice we have found several websites rely on this event handler to deliver different inputs to the user. Therefore, we explicitly trigger the event handler after modifying element values.

Circumventing Same-Origin Policy. SSOScan needs to iterate through DOM elements while searching for candidates, but it cannot reach elements inside iframes from a different origin. This is because the content scripts run as the principal of the host, and thus their accesses to iframes are subject to the same-origin policy. However, we saw many cases where the target button is in an iframe which originates from the HTTPS domain of tested website. To handle this, we inject content scripts into all iframes that have an HTTPS origin. Excluding HTTP iframes will cause some content to be missed, however, including them generates problems as sites sometimes include iframes from a complete different website which may include SSO/submit buttons. Besides, login and registration requests should be served over secure connections to prevent eavesdropping. Some rare exceptions usually lead to more serious vulnerabilities such as password disclosure, but our prototype implementation does not check for this.

4.6.3 Experiment Setup

In theory, SSOScan could exhaustively trigger clicks on every element on the page (and on all response pages up to some maximum depth) which would result in nearly 100% success rate. This would be prohibitively slow in practice, though, so the number of attempted clicks must be limited for any realistic test. Given the time needed for each click attempt, it is important to configure our scoring heuristics well to maximize the probability of a successful enrollment in the minimum amount of time.

To gather statistics about the candidate elements, we modified SSOScan to try all possible strategies even if it has already *found* the correct login button and to record information about all attempted clicks, including for example their size, position, visibility to the user, content string feature and whether it is *successful*. We define a click as *successful* if it is included in any sequence of clicks from the start page to triggering the SSO process, regardless of whether it appeared in an attempt that failed to trigger the process. Note that because SSOScan skips previously explored states to avoid redundant effort, it automatically rejects click sequences which involves cyclic state transitions, such as clicking on a random link and then clicking on a logo which leads the application back to its initial state.

We set up SSOScan to expand the candidate pool size for each configuration from 3 to 8, add more matching regular expressions (e.g., to match the string “forum” which occasionally leads to a login page on sites where no login is visible on the start page), and use equal weight for each of them. We also removed all candidate filters described in Section 4.6.1. Our goal is to capture as many ways to trigger the SSO process

as possible by doing as close to an exhaustive search as is feasible. This increases the time required to scan a typical site to almost an hour (compared to a few minutes with the setup used in the full study), and does not include the registration or vulnerability status detection as it is only concerned with triggering the Facebook login process.

We ran the test on 973 sites from the top-10,000 ranked sites that were detected by SSOScan to support Facebook SSO in our main study (Section 4.5). This biases the study slightly, since it only includes sites where the configuration used in the initial study is able to find Facebook SSO. Ideally we would like to run all top-10,000 sites to avoid any bias introduced by the data set, but the significantly increased testing time prohibits us to do so, and the result of our subsequent study on a random sample of sites (Section 4.6.5) supports that only few sites containing Facebook SSO were missed by the main study.

4.6.4 Login Button Statistics

The experiment recorded 29,539 unique[1] click attempts, of which 5086 (17.2%) are successful (that is, they either directly trigger SSO, or lead to subsequent clicks that trigger SSO). This amounts to approximately 30 unique clicks attempted per site, but the number varies significantly based on the site design, from a few up to 109.

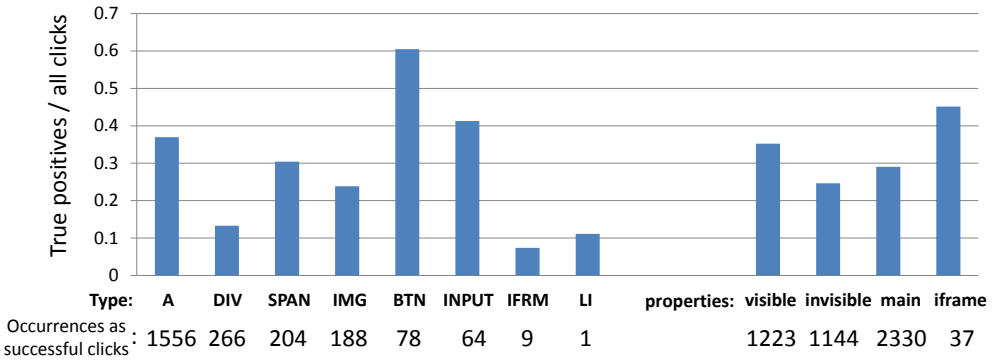


Figure 4.14: Login button type statistics

Element type and content. Figure 4.14 shows how different button types and properties impact success rates. We report the success rate as the number of times that element appeared as a successful click divided by the total number of clicks attempted on elements of that type. The number beneath the element feature gives the total number of times that type of element occurred as a successful click target across all the test sites. For example, the *BUTTON* element type has an excellent success rate — 60% of all *BUTTON*

[1]If two clicks happens on pages with the same URL, same element XPath and same element outerHTML, we consider them the same click.

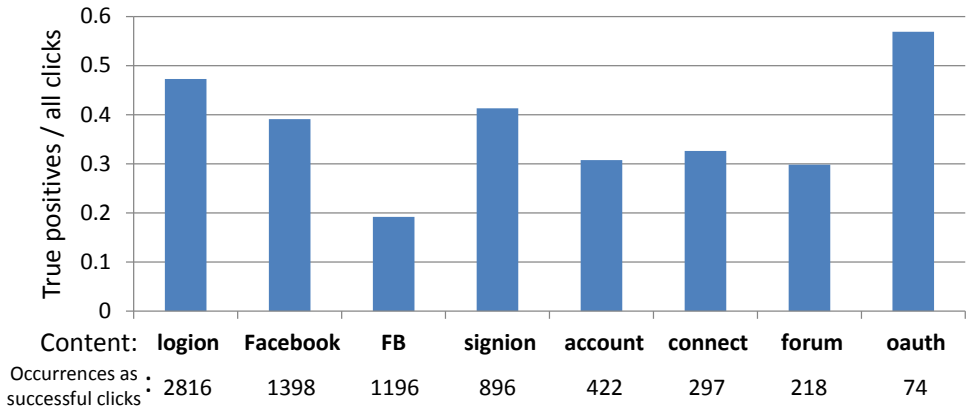


Figure 4.15: Login button content statistics

candidates are true positives for the Facebook SSO button. But since it only appears as a successful click on 78 out of 973 sites in our sample, it is rarely useful. By contrast, clicking on *DIV* elements are much less likely to trigger the Facebook login, but such elements are more common. The right side of Figure 4.14 shows that elements that are directly visible to the user has a higher success rate than invisible ones, and elements residing in iframes are twice as likely to be the correct target as their counterparts in the main page. These results suggest ways of weighting element types to improve the scoring function and increase the likelihood of finding successful clicks early.

Figure 4.15 shows how the success rate varies with attribute content (matched by the given regular expression). The keyword "oauth" rarely exists in any content, but when it appears it is very likely to identify the target element. The result also shows that "FB" is not a good indicator to predict the target, and we think this is probably because it is very short and may be used for similarly named JavaScript variables or random abbreviations.

Both figures include data for the *first* click only (but do measure first click success based on subsequent clicks). Data for the second clicks are noticeably different from the first, and overall success rates are lower on second clicks. The most interesting fact we found is that "connect" (39%) and "Facebook" (36%) become the most successful matches of all regular expressions, followed by "oauth" at (26%). No other regular expressions exceed 20% success for the second click.

Element size. Figure 4.16 gives the cumulative distribution function of the width and height of target elements. For example, the 80^{th} percentile width of the true positive elements is approximately 150px, compared to 300px for false positive elements. We did not find any significant difference between first and second clicks, so the figure combines data from all clicks. The key result is that wide elements are less likely to be true positives, possibly due to SSOScan incorrectly including many large underlay elements as candidates.

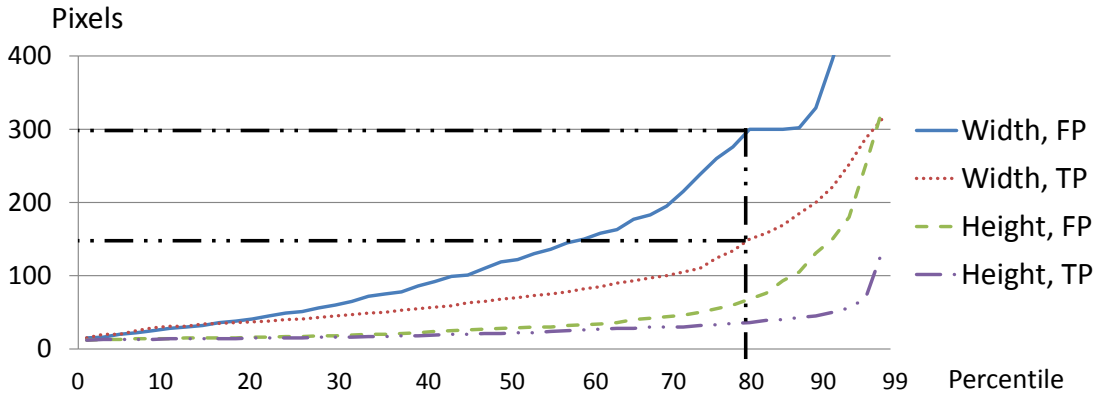


Figure 4.16: Impact of Login Button Size

The result is similar for element height (the lower two lines in the figure). This suggests that it would be useful to add a filter function that excludes candidates whose width is greater than 300px. We would expect it to eliminate 20% of the false positives while hardly missing any of the true positives. Alternatively, SSOScan could adjust the final score of a node according to its size based on these results.

Element position. Figure 4.17 shows the heatmap of the login button's position in a page. The intensity at a location indicates the number of elements found there satisfying the property. Only visible elements are shown, and each successful click only attributes to the intensity once. All four figures are normalized with respect to their maximum intensity (i.e., element density).

The figures show an interesting distinction from first click to second click: successful first clicks almost exclusively appear in the upper right corner of the page, while the second click appears generally in the upper-middle part of a page. The false positives are relatively more scattered everywhere on the page[1]. This result suggest we should assign a higher weight for elements for these locations, and focus on elements in the vicinity of the upper right corner for the first click. We could potentially even ignore the other criteria and only consider position to find login buttons on foreign-language sites.

Another potential improvement that we can learn from this result is the better sequence to retry different strategies. Assuming a strategy places restrictions on candidates that they must be visible and smaller than 300px. We learned from Figure 4.14 and Figure 4.16 that successful clicks on invisible elements are far more common than elements larger than 300px. Therefore, if this particular strategy fails to find the correct element, the immediate next strategy to try should expand search to invisible elements, but rather than larger ones.

[1]The figures also show a clear width boundary. In the experiments the browser resolution is 1920x1200, and it seems that most developers' designs follow a standard width of approximately 960px, which is why the density appears to be cut off.

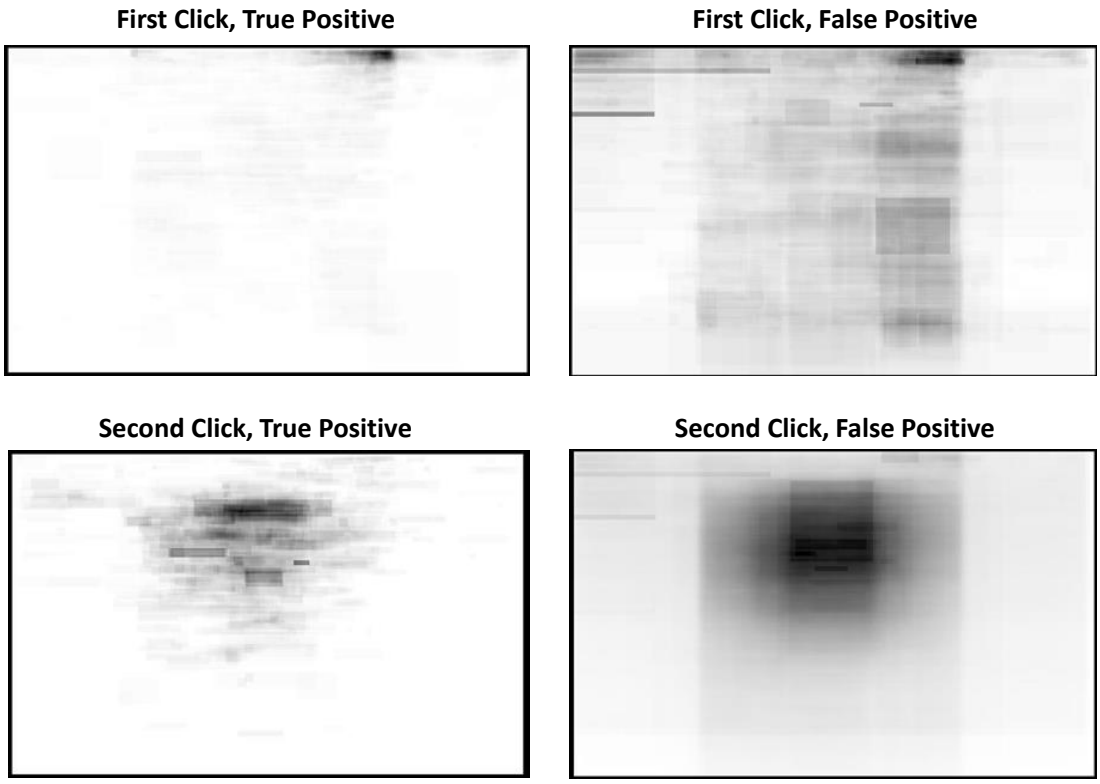


Figure 4.17: Login button location heatmap

4.6.5 Validation

After incorporating what we learned from these results (e.g., weight adjustment for different button sizes and types), we reran the SSOScan with the new heuristics on the sites ranked from 10,000 to 20,000 that SSOScan determined to support Facebook in the original study, which were not included in the heuristics evaluation. We compare the results with those obtained by using a "control" version of SSOScan, with equal weights on all features and no candidate filtering. All other settings such as candidate pool size are the same between two versions.

The results support the hypothesis that adjusting heuristics according to the results of the evaluation can improve the speed and robustness of detection of Facebook SSO integrations. The naïve control version missed 72 out of the 601 sites while the new heuristics missed only two. The average rank of correct candidate elements for the first and second click is 1.32 and 1.23 for the control experiment, which improves to 1.23 and 1.17 respectively with the new heuristics.

We also randomly picked 500 random sites from the sites that SSOScan have yet to find Facebook support in the experiment in Chapter 4.5. We tested the expanded heuristics on these sites, and further increased the maximum click depth to three to see if more SSO integrations could be found. Individual tests took an

average of 31 minutes to finish, but varies significantly from a few minutes up to an hour (threshold) based on site content.

Four additional sites were found that support Facebook SSO from this sample in total. Two are found due to the added regular expression [Ff][Oo][Rr][Uu][Mm], one of which required 3 clicks to trigger the SSO process. Another site is found due to the improved candidate ranking algorithm, and the fourth was found using the new candidate selection method that includes all elements in the right corner of the page, even if they do not match any regular expressions. This provides a reasonable degree of confidence that our original study found a large enough fraction of all the popular sites using Facebook SSO to be representative, although likely missed around 1% of Facebook SSO sites. We did not try click depths greater than 3 because of the exponential time growth required to complete each test, but we feel confident that the number of Facebook SSO interfaces that can only be discovered by attempting more than 3 clicks is very low.

4.7 Vendor Response and Deployment

In this chapter, we share our experiences reporting vulnerabilities, and suggesting ways SSOScan can be deployed to help secure applications integrating SSO services.

4.7.1 Developer Responses

We started contacting the site owners soon after obtaining our first list of vulnerable sites, sending out notifications to 20 vulnerable websites that we thought were interesting. We contacted them either through email or by submitting forms on their website. The responses were very disappointing, compared with previous experiences reporting SDK-level vulnerabilities to identity providers in Chapter 3. The vulnerabilities found by SSOScan, on the other hand, are primarily in consumer-oriented sites without dedicated security teams or clear ways to effectively report security issues. Of the 20 notifications, we only received 8 responses, most of which appear to be automated. After the initial response, three websites sent us follow-up status updates. `Espn.com` thanked us and told us the message has been passed onto appropriate personnel, but no follow up actions ensued. One of `answers.com`'s engineers asked us for more details, but failed to respond again after we replied with proposed fix. As of the time writing, both sites are still vulnerable. Four months after getting the automated reply from `ehow.com`, we received a response stating that they have removed Facebook SSO from their website due to "content deemed inappropriate", and we have confirmed that the Facebook SSO button has indeed been removed. Sadly, we think their staff likely did not (bother to) understand our explanation for the fix and simply removed the feature for their convenience.

The other instance where a reported vulnerability was fixed was for hipmunk.com. Hipmunk was found to be vulnerable to both the *access_token* and *signed_request* replacement attacks. We did not get any response from Hipmunk when the vulnerability was reported through the normal channels, but through a personal connection we were able to contact them directly. This led to a quick response and series of emails with one of Hipmunk's engineers. We explained how to check the signature of a *signed_request*, which should fix both vulnerabilities. However, when they got back to us believing that the fix was complete, we re-ran SSOScan and found that Hipmunk was still vulnerable to the *access_token* replacement attack. This meant Hipmunk checked the signature of *signed_request* after the fix, but never decoded the signed message body and compared its Facebook ID with the one returned by exchanging *access_token*. This surprised us, as we implicitly assumed the developers will consume the signed message body after verifying its signature, and thus only included 'verifying signature' in the proposed fix. After further explanation, the site was fixed and now passes all our tests.

4.7.2 Deployment

Our experiences reporting vulnerabilities found by SSOScan suggest that notifying vendors individually will have little impact, which is consistent with experiences reported by Wang et al. with online stores [3]. Hence, we consider two alternate ways of deploying SSOScan to improve the security of integrated applications.

App center integration. We believe SSOScan would be most effective when used by an application distribution center (e.g. Apple store, Google Play) or identity provider (e.g., Facebook) as part of the application validation process. The identity provider has a strong motivation to protect users who use its service for SSO, and could use SSOScan to identify sites that can compromise those users. It could then deliver warning messages to visitors of vulnerable applications during the log in through Facebook SSO process, or even go so far as to shut down SSO logins for that application. We also believe our results can provide guidance to vendors developing SSO services. The results in Chapter 4.5.1 indicate that sites are more likely to misuse credentials when using the Facebook JavaScript SDK. With Facebook's help, this problem could be mitigated by placing detailed instructions inside the SDK. The instructions could be presented as (non-executable) code in the SDK rather than as comments, so that the developers cannot get by without reading and removing them.

Checking-as-a-service. Without involving an centralized infrastructure, the best opportunity to deploy SSOScan is as a vulnerability scanning service that developers can use to check their implementations before their applications are launched (our prototype service at http://www.ssoscan.org/ can be used for this now). For a developer-directed test, it would be reasonable to ask the developer to either guide the tool

through the registration process or provide a special test account that bypasses this step in cases where it cannot be fully automated. Even if we assume no aid from the developers, they should at least be able to tolerate a longer testing time than is feasible in doing a large-scale scan.

Chapter 5

Restricting Untrusted Web Scripts[1]

In contrast to the last two chapters, we consider the scenario where the third-party service provider is not trusted to perform any security-critical actions on the application in this chapter. For example, the services of interest include web analytics and tracking scripts, online advertising and social widgets, as described in Chapter 2.1. Nikiforakis et al. showed that many popular sites include several third-party scripts per page and the trend is increasing [1]. Although such services are widely embedded in both web and mobile applications, we focus on web applications only in this dissertation.

Threat model. We extend the threat model described in Chapter 2.1 in more details here: the adversary controls one or more of the scripts embedded in the target page. To obtain private content, that adversary's script may use any means provided by JavaScript to get the text or attribute of a confidential node, or by probing values of variables in host scripts. We do not target (i.e. restrict) JavaScript frameworks such as jQuery that require rich, bi-directional interactions with the host's content. In these cases, we assume the developers fully trust the third party libraries. We also do not consider other attack vectors such as XSS attacks or web browser vulnerabilities. Many other projects have focused on mitigating these risks, and we concentrate on the scenario where the host page developer deliberately includes untrusted scripts.

Our goal under this threat model is different from the previous chapter: we aim to protect web content from embedded third party scripts based on fine-grained DOM content access control. Our solution also isolates third-party script execution contexts, preventing any undesired interactions by third party scripts and host scripts. These mechanisms significantly limit the damage a malicious third party script can do. We assume a one-way trust model since our goal is to protect user content from untrusted scripts rather than to protect embedded scripts from the host page or each other. In summary, our goal is to provide third party

[1]The contents of this chapter is based on the paper: Protecting Private Web Content From Embedded Scripts [66].

scripts with limited access to the DOM and no access to host scripts, while granting host scripts full access to third party scripts and the DOM. We realize these goals by building a browser that enforces fine-grained policies and providing an automatic policy generator tool to help site administrators generate these policies.

5.1 Prior Works

Many previous works focus on addressing the challenge of safely executing scripts from untrusted sources in a web page. We list previous isolation mechanisms in this Section 5.1.1, and then discuss policy generation works in Section 5.1.2.

5.1.1 Security Mechanism

The three main mechanisms to enforce security and privacy policies are to either rewrite the JavaScript code, extend existing browser features, or use cryptographic measures. We first list related works that modify browser to support added security policy, and then discuss approaches that rewrite JavaScript and use encryption.

Extending Browsers. Jim et al. proposed a per-script policy to defend against XSS attacks [67]. The basic idea is to create a whitelist of the hash of all scripts that are allowed to run on the page. MashupOS [68] intends to fix the integrator-provider security gap by introducing several new tags that can be used to restrict embedded scripts in different ways. However, it cannot support the policies needed to handle current advertising scripts since MashupOS requires the third-party content to be embedded in a particular way. Following this work, Crites et al. [69] proposed a policy that abandons the same-origin policy (SOP) by allowing the integrator to specify public/private web content. Completely abandoning SOP would require significant changes to websites. Jayaraman et al. [70] introduced OS protection ring idea to DOM access control. Each node is assigned a privilege level and only scripts within appropriate rings can access that DOM element. Compared to these works, our work supports the most expressive and flexible policies while emphasizing policy generation.

Specifically, the aforementioned works do not support fine-grained JavaScript execution context isolation. To this end, Barth et al.'s *isolated worlds* mechanism [71] supports this goal and is designed to isolate browsers from extension vulnerabilities. The main idea is to separate extensions from each other by forking the JavaScript execution context into several independent ones. We adopt this mechanism to isolate webpage scripts. Since this work is not targeting to protect user privacy, each *world* has the exact same and complete view of the page DOM. To achieve the similar goal of isolating JavaScript execution context to a fine-

grained extent, JCShadow [72] modified Mozilla Firefox's JavaScript engine TraceMonkey; Stefan et al. [73] implemented additional APIs for Firefox and Chromium to support mandatory access control for execution contexts of different scripts. However, fine-grained DOM access control was also a non-goal for these works.

An alternative to restricting the private information third-party scripts can access is to do computation on sensitive content inside the local browser completely. This approach is taken by Adnostic [74] and RePriv [75]. However, the downside is that such approaches would often require significant changes to the current web infrastructure.

Rewriting Client-Side Code. Rewriting JavaScript has the advantage over previously mentioned approach because it usually does not require browser modification and deployment. ADsafe [76] restricts the power of advertising scripts by using a static verifier to limit them to a safe subset JavaScript that excludes most global variables and dangerous statements such as **eval**. Caja [77] also limits JavaScript, but offers an automatic code transformation tool. JSand* [78] isolates SES-compatible JavaScript (Secure ECMAScript, a subset of JavaScript) execution by wrapping resource accesses using the new Harmony Proxy API. However, the rewriting procedure is often very complicated and cannot always preserve original script functionality and debug-ability. Additionally, a serious compatibility issue with these works is that they often do not allow the use of **eval** in JavaScripts as dynamically introduced code undermines the soundness of static analysis. According to Richards et al. [79], such usage is actually very popular among high-ranked websites. Finally, it is also challenging to implement JavaScript rewriters in a way that cannot be circumvented [80].

Adjail [81] does not rewrite JavaScript itself, but its surrounding document. It moves third-party scripts into a *shadow* iframe with a different domain name, using the browser's built-in same-origin policy to isolate the execution and sets up a restricted communication channel between the shadow iframe and host page. This approach does not require any browser modification, but has several limitations including inflexible policies (parent node can only have the intersection of children's privileges), difficulty to accommodate two or more embedded collaborating advertising scripts and complicated maneuvers needed to preserve impressions and clicks. The new HTML5 standard also provides a way of executing JavaScript in different threads using *Web Workers* [82]. The goal of this is mainly to improve JavaScript performance by allowing parallel execution and preventing misbehaving scripts from halting the browser, as opposed to security improvements. However, scripts running inside a worker shall be computation-heavy and do not have access to DOM APIs, therefore the feature is not appropriate to be used by embedding third-party services. To apply Web Workers to serve isolation goals, Treehouse* [83] extended this feature and virtualizes host DOM via a hypervisor-like interface

to enforce access control policy.

Encryption. ShadowCrypt* [84] propose to isolate the DOM by presenting two different views to JavaScripts and users with the help from ShadowDOM, an upcoming HTML5 standard. Privly [85] is an open source browser extension that encrypts data, stores them in the cloud, and leaves only ciphertext for the web application. The Cryptographic measures have a slightly different goal — they not only protect user data against third-parties, but also the host party. Therefore they may jeopardize the application's original functionality — computations cannot be trivially performed on the cipher text, such as sanitization and auto completion. Supporting such computations (such as full hormomorphic encryption [86] and secure computation [87]) often result in huge performance penalties.

Server-side isolation. In addition to client-side isolation mechanisms, server side may also employ similar defenses. NodeSentry* [88] uses JavaScript Harmony membranes to wrap accesses to critical resources in Node.js and apply policy checking code within the wrapper. Blankstein et al.* [89] developed third-party component isolation framework on Django. Based on the principle of least privileges, they are able to automatically generate isolation boundaries by using dynamic analysis and developer-supplied execution traces. These works are in parallel to our work, and can be used to complement each other to provide better protection in end-to-end cases.

5.1.2 Policy generation

Compared to isolation mechanisms, there are relatively less previous works that provided automatic policy generation solutions. Conscript [90] is a system that uses AOP to support policy weaving. They proposed two automatic policy generation approaches — a tool that translates C# policy to JavaScript, and more interestingly, one that generates policy based on the principle of least privileges. However, Conscript authors only briefly mentioned the possibility of the latter and the generated policies are rather coarse-grained. Another work, XRay* [91] follows a very similar approach to our policy generation idea but targets a different goal — they aim to identify the possible factors causing different personal recommendations (in e-commerce sites) and targeted advertisements to appear across two profiles. As a semi-automatic tool, Mash-IF [92] provides a GUI tool to let developers mark sensitive information on the page, and uses information flow tracking techniques to restrict data leakage.

*Work done after our paper was published.
*Work done after our paper was published.

5.2 Security Policy

Our protection policies are separated into two categories: JavaScript execution isolation as explained in Chapter 5.2.1, and DOM access control as explained in Chapter 5.2.2.

5.2.1 Script Execution Isolation

One of our primary goals is to let the web developers easily group third party scripts so that some of them may collaborate with each other while still remaining separated from other third party scripts and host page scripts. To facilitate this we add a new attribute to the script tag: worldID=*string*. This idea originates from Barth et al.'s *isolated world* concept [71] which was developed to isolate browser extensions. Each world with a unique worldID is isolated from all other worlds. The worldID attribute also serves as the principal for scripts for controlling access to DOM nodes (discussed in Section 5.2.2).

Listing 5.1: Policy for execution context separation

```
1  <script worldID = "1">
2    var a = 3;
3    function f() {}
4    Boolean.prototype.toString = f;
5  </script>
6  <script worldID = "2">
7    var b = a; // error: a undefined
8    f(); // error: f undefined
9    new Boolean(0).toString(); // original
10 </script>
11 <script worldID = "1">
12   var b = a; // OK
13   new Boolean(0).toString(); // f()
14 </script>
```

Listing 5.1 illustrates the semantics of the worldID attribute. The custom and native objects of the first script (in worldID="1") are isolated from the second script because they have different worldIDs. This means the variable a, defined in the first script, is not visible in the second script, and the second script only sees the original toString method. Since the third script has worldID="1", it executes within the same context as

the first script and can access all the objects the first script can.

Shared Libraries. Full isolation of embedded scripts would break the functionality of many host pages. In some scenarios, the developers still want to access certain APIs provided by third-party scripts without providing reciprocal accesses. We added two new attributes to script tags: sharedLibId and useLibId. All objects inside a script tagged with a sharedLibId attribute can be accessed by the host execution context as well as all other worlds that have the corresponding useLibId attribute. On the other hand the third party scripts themselves cannot access the privileged scripts and are still bound by the DOM access policies. For example, *Google Analytics* users can use the custom variable _gaq to track business transactions: the user pushes transaction information into the array _gaq which is later processed by Google Analytics, e.g. _gaq.push(['_addTrans', '1234','11.99']);. If the two execution contexts are completely isolated, the _gaq variable would not normally be visible in other worlds. To support this, the sharedLibId attribute is defined to identify when an embedded script is a shared library:

```
1       <script src="google.com/GA.js" worldID="1" SharedLibId="GA">
```

Then, other third-party scripts can use the useLibId attribute to access objects defined in the shared library. To prevent pollution of other script objects, objects in the shared library are prefixed with the library identifier. For example, to access the variable _gaq, developer needs to append GA in the prototype chain.

```
1       <script useLibId = "GA">
2           GA._gaq.push(['_addTrans', '1234', '11.99']);
3       </script>
```

Note that as pointed out by Barth et al. [93, 94], leaking a seemingly trivial function reference to a malicious script may pose greater risks if the function is not carefully examined, and could lead to arbitrary code execution in the host context, therefore compromise the entire sandbox mechanism. Here we assume all the references given to untrusted scripts are safe, and leave this securing duty to the developers. We consider vulnerabilities introduced when leaking insecure object references out of the scope of this dissertation.

5.2.2 DOM Node Access Control

Listing 5.2: Policy for DOM access mediation

```
1  <div id="a" RACL="1,2" WACL="1">
2    Username: Alice
3  </div>
4  <script worldID = "1">
```

```
5    var b=document.getElementById('a'); // OK
6    b.innerHTML = 'changed'; // OK
7  </script>
8  <script worldID = "2">
9    var b=document.getElementById('a'); // OK
10   b.innerHTML = 'changed'; // error: 2 not in WACL
11 </script>
12 <script worldID = "3">
13   var b=document.getElementById('a');
14         // error: a not readable
15 </script>
```

In addition to isolating objects in scripts, we provide fine-grained access control over host objects in the form of DOM nodes. We introduce two additional tags for all nodes in the DOM tree: RACL for specifying read access, and WACL for specifying write access. Each access control list is a comma-separated list of worldIDs. Only scripts running in the worlds listed in the RACL list are permitted to read the node, and only scripts listed in WACL are permitted to modify the node. For example, if a third-party script wants to remove a node, it must have the privileges of modifying both that node and its parent (this is consistent with the JavaScript syntax for removing a node which requires two node handles: parentNode.removeChild(thisNode)). On the other hand, to append a node to an existing node, a script needs to have write privileges for the parent node and read privilege to the node to be inserted. The access control list of a node does not depend on its parent or children.

As shown in Listing 5.2, a script can only access a particular div element if it is present in the corresponding access control list of that element. Our policy is more fine-grained and flexible than previous works like Adjail [81] and MashupOS [68] (policy based on tree structure). Particularly, our policy allows developers to create a public-accessible node deep down a DOM tree branch where all its ancestors are protected (host embedding an advertisement under such node could prefer this configuration), or to set that node as the only protected member while all the rest are public (Only that node contains private information, and the host wants to protect it). Table 5.1 summarizes the customizable policies for providing fine-grained mediation of host objects together with the control of sharing and isolation of custom and native objects.

Special API Properties. In addition to specific DOM nodes, we also provide developers with policy options to hide selected APIs from certain scripts. These special host objects may provide scripts with powerful capabilities or private information. For example, **document**.cookie returns authentication tokens that an untrusted script might exploit. The defense mentioned above cannot prevent this because cookie is

Context	Policy syntax	Semantics
script	worldID="s"	WorldID of the script context is s
script	sharedLibId="s"	This is script from s library
script	useLibId="s"	This script requires to use s library
DOM node	RACL="$d_1, d_2, \ldots$"	Worlds that may access this
DOM node	WACL="$d_1, d_2, \ldots$"	Worlds that may modify this

Table 5.1: Summary of Policy Attributes

a special property of the **document** and is therefore not tied with any specific node. There are many other powerful APIs such as **document**.write() and **document**.open(). Therefore we add a set of new attributes for the <html> tag to allow the developer to specify these per-API/per-script policies, other examples include **document**.location, **document**.URL and **document**.title, etc. These privileges are disallowed for untrusted scripts unless explicitly annotated.

5.3 Automatic Policy Generation

In the last chapter, we described the fine-grained policies of our modified browser supports. We envision that a common policy that applies to any web application is to specify a list of private DOM nodes and protect them from untrusted scripts' access. This could be done by web application developers manually annotating nodes as public or private. Manual annotation, however, is probably too tedious for most web applications and unlikely to happen until a protection system is widely deployed. Hence, we develop a technique for automatically identifying nodes that contain private content. If we had access to the server, one strategy would be to use information flow techniques at the server to track private content and mark nodes containing private content when they are output. Since we do not assume server access, however, here we consider a dynamic technique for inferring private content solely based on the pages returned from different requests.

We define private content as content that depends on the user's credentials. Thus, any content that is different in an authenticated session from what would be retrieved for the same request in an unauthenticated session is deemed private. Nodes that directly contain private information should be marked private, but not the parent of that node. For example, if Username appears, only the inner span element is private, but not the outer div. The fine-grained nature of our policy enforcement supports this definition well. Automation is done by submitting multiple requests to the server with different credentials, and identifying the differences.

One of our design goals is to minimize the changes have to be made both on server side and on client side, so we use a proxy server to add security policies. Figure 5.1 provides an overview of our policy learner structure. Proxy server automatically identifies third-party scripts and generates the policies for the response

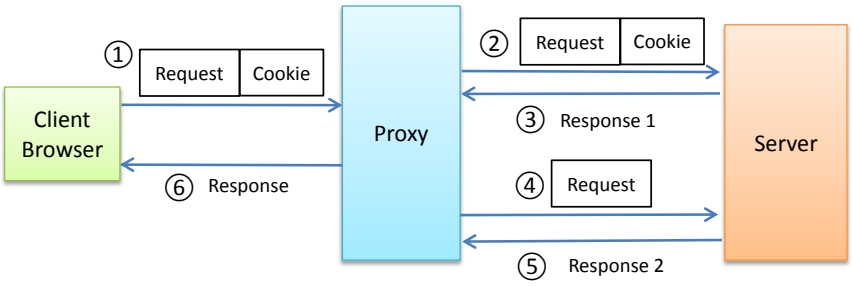


Figure 5.1: Automatic Policy Generator Workflow

when a request is captured, which is passed on to browser client to enforces them.

We use the Squid proxy server as our mediating proxy. It supports ICAP (Internet Content Adaptation Protocol), which allows us to modify web traffic on the fly. For convenience, we run the Squid server in the same machine as our modified browser, however one can definitely move that onto an intermediary node along the routing path for better centralized control. We use GreasySpoon [95] for the ICAP server implementation.

This design cannot deal with SSL web traffic since the proxy will not be able to see the decrypted traffic. The Chromium development group is currently (as of April 2014) still working to implement webRequest and webNavigation as extension APIs [96]. Once these are implemented, we can move our proxy server inside the browser thus making it work on SSL/TLS traffic and easing deployment.

The content adaptation is divided into two main functions: third-party script identification and public node marking.

Third-party Script Identification. The ICAP server examines the response header. For each script with a source tag we compare the script's source with the host domain. For scripts that come from different domains, we add worldID attributes that identify the origin and indicate that they are not trusted by the host.

Identifying Public Nodes. To identify public content, our proxy compares the responses from two requests, one with the user's credentials and one without, and denotes any content that is identical in both responses as public. For example, assume a user visits `nytimes.com` so the browser sends a request with that user's credentials as cookies to `nytimes.com`. Once our ICAP server sees the incoming response it sends another request almost identical to the first one except with the cookie stripped. Then it collects and stores this response, denoted R_{pub}, which is the response for the stripped request. The original response is denoted R_{priv}.

Once both responses are ready, the proxy executes a differencing algorithm. This is similar to a simple text diff, except it follows the node structure. Initially, all nodes are assumed to be private. Then, any node in R_{priv} that appears identically in R_{pub} is marked as public. For write accesses, we make sure all children of

a root node are the same before marking the root node public. Read access is slightly more relaxed than write access, since we already modified innerHTML function to conceal private nodes inside a subtree. We mark a node public as long as its attributes and immediate textnode children are the same. The algorithm for comparing the responses is given below.

Algorithm 1 Automatic Policy Generation Algorithm

```
while Browser initiate request Q_priv do
    R_priv ← originalResponse;
    TrustedHost.push(R_priv.header.host);
    for all script S in R_priv do
        if ! TrustedHost.contains(S) then
            S.worldID ← S.src;
            UntrustedScript.push(S.src);
        end if
    end for
    if (Q_priv.requestType == GET) then
        if host ∉ Cache ∧ timeout ≤ T then
            Send new request Q_pub with cookies stripped
            R_pub ← Q_pub.response
            Cache ← R_pub
        else
            R_pub ← Cache.get(host)
        end if
        for all node N in R_priv do
            for all node N' in R_pub do
                if N'.attrs == N.attrs ∧ N'.innerHTML == N.innerHTML then
                    N.WACL ← UntrustedScript.all;
                end if
                if N'.attrs == N.attrs ∧ N'.textChildren == N.textChildren then
                    N.RACL ← UntrustedScript.all;;
                end if
            end for
        end for
    end if
end while
```

State-Changing Requests. Our policy learning process requires sending duplicate requests to the server. This could have undesired side effects if an unauthenticated request can alter server state. To limit this, POST requests are ignored since firing them twice should result in undesired state changes at the server. Hence, the entire response from a POST request is considered private. The HTML specification suggests GET methods to be idempotent [97], but many sites do change persistent state for GET requests. For example, a forum site might use a GET request for anonymous postings. If we submit the request twice the anonymous comment may be posted twice since no credentials are required for the posting.

To prevent this, we consider two possible solutions. The first is for the server to annotate non-idempotent pages. The first time user visits a site, if the proxy has not yet seen it before, it skips the request duplication

and looks for idempotent field in the response header. Servers can send idempotent=**false** in the header to indicate that the browser should not to send duplicate requests for this page. If the idempotent field is not detected in the first response we default the proxy behavior to proceed as if the site requests are idempotent.

A second approach is to set up a third-party service like AdBlock and have users subscribe to this service. A centralized server will collect information from users and correlate responses to mark private data. If we have an authority like this we do not necessarily need to send two requests since other users may have already fired similar requests and the server should already have recorded the responses. This centralized server should be established at the ISP so that we do not introduce extra vulnerable point in the network path. In this case the ISP server's identification accuracy would affect many more users than a local proxy, but it is also convenient to manually correct the mistakes as a center server.

The biggest disadvantage against a centralized service is user's privacy concerns. In most situations, users and the host may not want to disclose decrypted traces which are collected and differenced at a third-party (even if this happens at the ISP, it is still not supposed to see SSL/TLS decrypted traffic). Techniques such as secure computation [98, 99] could be used, but are out of the scope of this dissertation.

5.4 Implementation

We base our implementation on Google's open source Chromium project. We built Chromium on Windows 7 under Visual Studio 2008. Approximately 1500 lines of code were added or modified, mostly in the WebKit DOM implementations and the bindings of V8 (the JavaScript interpreter) and WebKit DOM, but leaving V8 unmodified. Hence, we believe our implementation could be adapted to other browsers that use WebKit as well, with the effort of adding *isolated world* support.

Figure 5.2 illustrates how a DOM API call is executed in our system. In step 1, the WebKit parser parses a raw HTML file from a remote server and passes each script node to the *ScriptController* in WebKit/V8 bindings to set up the execution environment. If the context associated with the current worldID is already created, *ScriptController* instructs V8 to enter, otherwise it creates a new world. In step 2, the *ScriptController* sends the script to V8 to initialize execution. At some point, V8 may encounter a DOM API call and invoke a callback to the corresponding function using the WebKit/V8 bindings (step 3). In step 4, that callback function is modified to include policy enforcing code that checks the worldID against the ACLs of the node. After passing the policy checking, the call is forwarded to the WebKit DOM implementation (step 5). In cases where modification happens, the target node is also tainted according to rules explained in Section 5.4.3. Finally, the return value is returned from WebKit DOM back to V8 (step 6). Next, we provided details on

how we enforce script isolation and mediate access to the DOM. Section 5.4.4 discusses some special issues for handling dynamically-generated scripts.

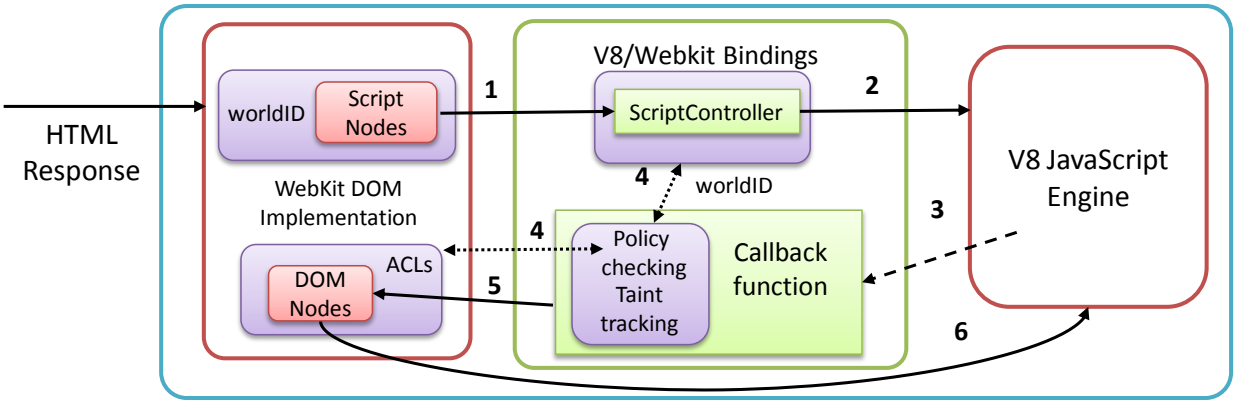


Figure 5.2: Modified Chromium Workflow

5.4.1 Script Execution Isolation

Isolating any two scripts by putting them into different execution contexts allows us to specify per-script policies. We adopt Barth et al.'s *isolated world* mechanism [71] on Google Chrome. This mechanism was originally designed to separate the execution context of different browser extensions, so that security compromise of one extension does not lead to the compromise of the host page or other extensions.

The DOM-to-JS execution context one-to-one mapping is changed to be one-to-many: each context maintains a mapping table to the DOM elements of the host page. This ensures that only host objects are shared among all worlds, but not native or custom objects. Note that a malicious script can still interfere with the host script execution by making changes to host page DOM elements (the changes are propagated all other worlds), however, page-specific policies can be enforced to disallow such modifications. In our work we extended this mechanism to apply to common embedded scripts in addition to extension content scripts. We modified Chrome to recognize the attribute worldID so that the WebCore ScriptController can support different JavaScript execution contexts according to scripts' worldID. A hashmap of all the execution contexts is instantiated on a per-page basis to enable scripts to execute in the correct context. As long as the worldIDs are not the same, the two scripts run in completely different contexts and cannot access each other's variables or functions.

Host Script Access. Now we address the compatibility issue: granting the host scripts access to the third party objects without compromising previous protections. There are two interesting facts here: First, all objects defined in the script are properties of the **window** object. Second, it is possible to inject arbitrary

objects into another context using Google V8 JavaScript engine APIs. Combining these two together, we modified the browser to automatically grab the handle of the **window** object of that context and inject it into the host context as soon as a third party script execution context with SharedLibId is created.

5.4.2 DOM Access Control

Fine-grained policies mean we can provide different access permissions to different scripts. We do this by either hiding inaccessible information from scripts based on their worldID, or in cases where more expressive policies are needed, by mediating access requests.

Our implementation mediates all DOM API getter functions to check the ACL of target node as shown in Figure 5.3. The upperleft and the lowerleft squares indicate two different execution worlds. As each world tries to grab handles of different nodes or call getterAPIs on those nodes, some of them are thwarted by our mediation according to per-page policies; the ones that get through are executed normally. When a script attempts to request a reference of a hidden node or call any API on a hidden node, it will instead receive a fabricated result. We return the v8::**null**() object for functions that should have returned a DOM node wrapper; we return an empty string object for functions that should have returned a string object. These results avoid leaking any information, but provide a good likelihood that a well-written script will be able to continue (we confirm this in our experiments, as reported in Section 5.5.2).

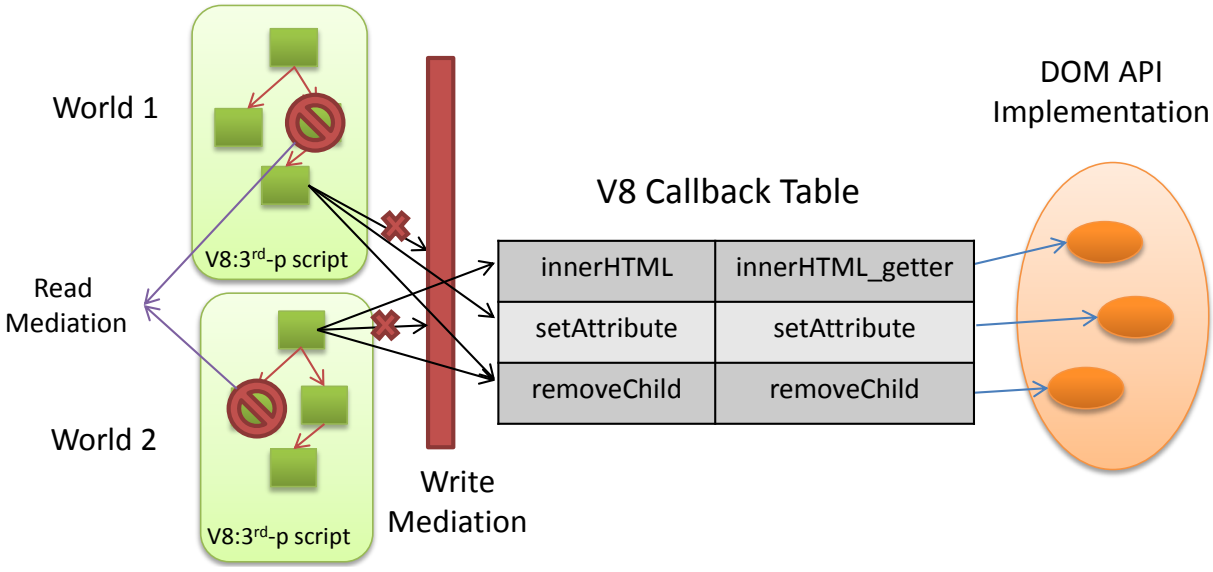


Figure 5.3: JavaScript to DOM API Mediation

Mediated DOM APIs include all node handler getters as well as APIs that can be called after a node handler is held, such as getAttribute(). One of the trickier APIs is the innerHTML getter, as well as similarly-functioning APIs. These APIs are designed to return the text/HTML markup of all children of this node. Other related

works [68, 81] that define the parent node cannot be assigned higher privileges than the intersection of its children does not have to pay specific attention to this. Since our policies are more flexible, calling such APIs may pose a privacy leakage threat in certain scenarios — e.g. some of the children nodes may have been marked private while the root node is marked public, and calling the innerHTML getter on parent nodes may reveal confidential information in its children. To remedy this, we modify the DOM implementation of the innerHTML callback and other similar APIs to filter out private nodes from the result.

Read-Only Access. Providing read-only or other restricted access is more complex since it requires giving the script a handle to the node. We mediate five ways a script may modify a node: 1) Directly changing the property (Chrome calls the internal setter function) of that node; 2) Modifying the style of that node; 3) Modifying the children of that node; 4) Modifying the attribute of that node by calling node-specific JS-DOM APIs, (e.g., setAttribute (), textContent, etc.); 5) Attaching/removing any event handlers to that node (e.g. addeventhandler()).

Each of these is handled in a completely different fashion in Chromium, so to impose a read-only policy it is necessary to address all of them. In addition, the security attributes, i.e. WACL, RACL, and worldID should never be changed by scripts other than the host since this would allow untrusted scripts to change the policy. We therefore modified the attribute setters to check attribute names and accessing script's worldID to prevent unauthorized modifications to these attributes.

5.4.3 Lightweight Taint Tracking

Since a node may initially contain only public information, but later be modified by a host script to contain private information, it is important to update the privacy of a node when it is modified by a script. To do this, we employ a conservative taint-tracking technique. A DOM node is marked as private as soon as any host script modifies it. Nodes that are modified by a script with worldID=i is also only visible to scripts in world i as well as the host scripts. This use of AJAX/XMLHTTPRequest to dynamically authenticate users and update respective content is not uncommon among the sites we have tested (for example, cnn.com uses JavaScript to update the user name box on the upper right corner of the page after the entire page is loaded).

Our experiments show that this conservative tainting policy could occasionally lead to compatibility problems when too many nodes are tainted, we relaxed tainting by adding a heuristic to only taint the nodes whose text content or source attributes have changed. This lowers the false positive rate by ignoring the CSS and location changes of the nodes. In case this policy is inappropriate for certain websites, developers can always resort to manually marking these nodes as private using the WACL and/or RACL attributes. This heuristic does not pose a privacy risk as long as the provided policies are accurate, but enables side-channels

between scripts that could otherwise not communicate since they may be able to modify a node that can be read by the other script in ways that are allowed by the heuristic. We do not consider this a serious security risk since private data is only exposed to a third-party when explicitly allowed by the policy, so although that script can now leak the data to a different third-party script it could also misuse the data directly.

5.4.4 Dynamic Scripting

Many previous works feared the consequences of allowing this dynamically-generated code and simply excluded dynamic parts of JavaScript such as **eval**. This fear is justified for any rewriting-based approach since dynamically generated code may circumvent the rewriting. In our case, we can support dynamically introduced scripts since policies are enforced at runtime, but need to be careful to assign the correct policies to these scripts. When the generated scripts execute in different contexts from the scripts that created them, it may cause broken functionalities (variables and functions that should be shared are now isolated) as well as privilege escalations (less privileged scripts are able to dynamically create a higher privileged script).

We solve this problem by propagating worldIDs: scripts should automatically inherit the worldID from their creator, thus executing within the same context as their creators. We mediate all four ways to dynamically evaluate a script: 1) calling **eval**() or setTimeOut(), etc.; 2) define an anchor element with JavaScript pseudo-protocol (i.e., javascript :code;); 3) creating a script node with arbitrary code; or 4) embedding a new script node by calling **document**.write() or **document**.writeln. The first two cases can be handled by modifying respective script initialization functions in the V8ScheduledAction and ScriptController class. In the third case we need to strip any worldID attributes from created node and append the correct one (its creator's worldID) to it. To address the last situation, we modified **document**.write() function to pre-process its argument and append the worldID attribute to all script nodes appearing in the new HTML content.

Event Handlers. In addition to the common way of executing a script by adding a DOM script node, third-party scripts may run arbitrary code in the context of host scripts by adding that code as an event handler of another DOM node, assuming the event can be triggered (e.g., using the onload event). Event handlers can be dynamically added by JavaScript APIs or explicitly defined as an attribute, for example: <div onclick = >. There are four possible ways to attach an event handler: 1) direct assignment, e.g. someNode.onclick =; 2) setAttribute ; 3) addEventListener; or 4) creating an attribute node an attaching it to a node. To preserve policy enforcement and execution context, we must ensure the event handler code executes within the same context of the script that attached them. For each of the four ways of attaching event handlers, we propagate the worldID to make sure that the event handler attached executes within the correct context.

Note that after the host script registers an event handler, third party scripts may attempt to call that event handler (assuming it has read access to that node), by either retrieving the event handler function directly or synthesize an event. The former exploitable path can be blocked by mediating all getters of event handlers to make sure the caller's worldID is identical to the callee's, however, our current implementation does not block the latter scenario (synthesizing events) for compatibility purposes. We are aware that a clever attacker may construct an attack by concatenating pieces of event handlers together (similar to return-oriented programming [100]), but such possibilities shall be very low since the number of event handlers are relatively low in comparison to native programs.

5.5 System Evaluation

We evaluated security or our implementation by manually testing a range of possible attacks, its compatibility with a sample of web applications, and the effectiveness of the automatic policy generator.

5.5.1 Security

We tested our implementation against all the attack vectors we could identify from the W3C DOM [101] and ECMA specification [102]. Table 5.2 lists the attack vectors and examples of the attacks we tested. For each attack vector, we followed the W3C DOM/ECMAscript specification and created at least one test case for each feature in the specification and confirmed that the attack is thwarted by our implementation. Since most of these attack vectors are handled by a few functions in the Chromium implementation, this provides a sufficiently high level of confidence that our implementation is not vulnerable to these attacks.

Attack Type	Examples
Directly calling DOM API to get node handlers	**document**.getElementById(), nextSibling ()
Directly calling DOM API to modify nodes	nodeHandler. setAttribute (), innerHTML=
Probing host context for private vars	accessing host objects, explicitly calling event handlers
Accessing special properties	**document**.cookie, open(), **document**.location

Table 5.2: Evaluated Attack Vectors

5.5.2 Compatibility

To evaluate how much our defense mechanisms disrupts benign functionality of typical web applications, we conducted experiments on a proof-of-concept website we built ourselves and on a broad sampling of existing websites.

The first experiment we do is to construct a pre-configured webpage that already has all the required annotations and third-party scripts. This page functions well in our modified browser. Both advertising networks we tested (Google Adsense and Clicksor) behave normally during testing with no errors even while hiding as much user information as possible from those scripts by marking content nodes as private. Security properties verified previously ensure that embedded third-party scripts cannot access those information.

For real-world web applications, we picked 60 sites altogether to test the compatibility against existing websites, sampling a range of sites based on popularity. We chose 20 from top US 50 sites according to Alexa.com, another 20 sites from sites ranked 50-300 and 20 sites from below the rank of 1000 and tested basic functionalities such as login and site-specific operations. These sites contained a variety of third-party scripts including advertising networks (e.g. Doubleclick, Adsonar, Ad4game, etc.) and web analytics and tracking scripts. We isolated the third-party scripts and added the security policies on nodes that carry user data. We did not modify the embedded scripts. Policies for these pages are automatically generated by our policy learner which we will evaluate more extensively in the following chapter. Here we ensure the third-party script identification are correct. That is, in case a compatibility issue arises and that's due to the errors of automatic third-party script identification (the same issues as explained in Chapter 4.5.2, where we discussed the detection accuracy of SSOScan), we manually correct the policies and redo the test. We discuss situations where policy learner produces an incorrect policy in Section 5.5.3.

To ensure maximum compatibility, we relaxed our policy learner to always give the <head> tag's write access to third-party scripts. This was necessary since some analytics and ad network scripts inject script nodes in the head region. This relaxation is done without compromising confidentiality due to the fine-grained nature of our policy: user-sensitive data is never revealed from children nodes of the <head> tag as long as the tags that directly containing the private information is marked private.

With the assistance of our automatic policy generator and minimum manual annotation effort (mainly helping proxy server to recognize important library scripts as host scripts, e.g. jQuery), 46 out of 60 sites functioned without a problem. However, 23 of which do require our manual identification of third-party scripts. For example, we added aolcdn.com to aol.com's whitelist as trusted domain. Four sites have non-standard HTML responses which triggered unrecoverable errors in our HTML parser; Four sites do not contain private information/use only SSL traffic which our current implementation of policy learner cannot handle; Three sites have third-party scripts that try to access a private node and crashed in the process. After a closer look at all these accesses, we find out that the private nodes identified by our policy learner are actually all false positives. However we do not know what those third-party scripts will access after touching on a private node since it will crash immediately. Another three sites show many JavaScript console errors mainly due to host script trying to access many guest objects (e.g. `_gaq` as mentioned before) but our policy learner cannot

automatically append the guest global object before these accesses. This scenario also happened in some other sites, but the access is simple and we manually added the object easily. For more complicated cases, they can be addressed by either web developer's effort or dynamic modification within JavaScript Engine.

5.5.3 Policy Learning

To evaluate our automatic policy generator we ran our proxy on the sample websites and report the result of third-party script identification and private node marking here.

Third-party script identification. We have explained that a trusted third-party script may be misclassified as untrusted because it is from a different origin in Chapter 4.5.2. In addition, an untrusted script may be misidentified as trusted. This occurs when the host includes a third-party script using cut-and-paste. For instance, Google Adsense and Google Analytics require host pages to include an inlined code snippet. This is safer than an embedded script loaded from the remote site, since at least the host has the opportunity to see the script and knows that it is not vulnerable to a compromise of the remote server, but should still not have access to protected data on the page. Our policy generator has no way to tell whether an inlined script is associated with another third-party script. This causes certain functions to break due to the isolated execution environment of two mutually dependent scripts. Our ad-hoc solution is to use heuristics to identify specific patterns in inlined scripts that correspond to commonly inlined scripts. For example, we look for `_gat` or `_gaq` in an inlined script and mark scripts that contain them with the same `worldID` as the embedded Google Analytics script. Since other scripts may now intentionally add such tokens in their scripts to confuse our policy generator, this is only a ad-hoc solution. Ideally, service providers would add this `worldID` permanently in their snippets.

Private Node Marking. In order to test the real-world private node marking accuracy of the policy generator, we tested the basic functionalities such as login and site-specific operations on the same sample sites used for the compatibility experiment. The traffic is redirected to go through the proxy server where modifications are made to the responses (e.g. adding ACLs and `worldIDs`). We recorded the total number of nodes, total number of nodes marked public before login and after login, total number of third-party scripts existing on the page and the trusted domains needed to be manually added (e.g., scripts stored in content distribution networks and library scripts such as jQuery).

Table 5.3 summarizes the results of our policy generation experiments. The sample size and ranking denotes the total number of sites we selected from that range of ranking at Alexa.com. Pct_{Before} is the percentage of nodes that are marked private before login, and Pct_{After} is the percentage after login. $Pct_{Switched}$ is the percentage of node that switched from public to private after login. The last two columns show the total

Size	Alexa Ranking	Pct_{Before}	Pct_{After}	$Pct_{Switched}$	3rd-p scripts	Trusted Domain
13	1-50	71.6%	52.6%	73.4%	0.84	0.73
11	50-300	95.7%	78.4%	82%	2.61	0.46
18	1000+	98%	83%	84.7%	2.2	0.5

Table 5.3: Summary of Automatic Policy Generation Results

number of third-party scripts on the host page and the number of trusted domains that need to be added to maintain functionality.

There is a reasonable drop in the fraction of nodes that are public after we login to the page, which is exactly what we are expecting. We can also see an increase in public content share after login as the ranking of sites goes lower, which indicates less private information are marked in less important sites.

Statistically, the average number of third-party scripts on a page grows as the sites become less popular. This indicates that less popular sites might take bigger risks in embedding untrusted scripts than larger sites. Finally, the number of trusted domains that have to be added averages less than one per site and drops lower as the ranking goes lower, consistent with the expectation that hosting scripts on alternate domains is more common with large sites. This result also indicates that the effort required for developers to denote trusted sites is minimal.

We also inspected the nodes that were marked as private in the abovementioned sites. Most of them are information that most people would consider private such as user names, email addresses, personal recommendations and preferences. In addition, some nodes that contain session-related advertisements and tags are also marked private, due to varying session-related attribute in those tags. These false positives are more frequently seen on more popular sites, as their sites are more dynamic and complex. The high false positive rate in high-profile sites is due to the limitation of our assumption: we do not assume the control over server side. As we have stated before, one heavy-weight alternative would be to modify existing web frameworks and add private data taint-tracking to server side: Nodes would be automatically marked private if the information it contains are tainted.

Chapter 6

Understanding and Monitoring Untrusted Code[1]

In Chapter 5 we presented how fine-grained DOM access control can be achieved by modifying the Chromium browser, and a proxy-based automatic policy generator that infers private information based on differences of two responses. While this is a solid first step, we think there are several issues with the design of the policy language and generator that may affect the usability of the tool.

First, the policy generator yields plenty of false positives on complex sites that include lots of dynamic content. Because the content served each time could be different even in the same session, according to the design the policy generator will deem these differences as private content and thus over-restrict the third-party scripts.

Second, our previous solution does not offer site administrators a way to inspect and understand what is going on under the hood for third-party scripts. Since the policy generator is likely to produce false positives (marking public content as private), admins will likely need to use a trial-and-error approach to make sure the strict policies do not break the scripts. This process not only is tedious but may also expose more information than necessary for the scripts to work. As add-on attributes to individual DOM nodes, the access control policies are hard to edit and maintain, especially across the entire website which may be consisted of many pages.

Additionally, the proxy-based policy generator will not work very well in practice due to the duplicated traffic overhead and unwanted server-side state change. To comply with the policy generator, a website will have to ensure no unauthenticated requests can make any changes to server state, which is often not the case

[1]The contents of this chapter is based on the paper: Understanding and Monitoring Embedded Web Scripts [103].

nowadays. This drawback will likely be the most important resistance to the deployment of our previous approach.

To avoid these issues, we now present an alternative design of a chain of server-side tools to assist developers to understand and monitor the third-party JavaScripts behavior. Accesses to critical resources and policies for third-party scripts can be visually presented to the site administrators. The visualization together with intelligible CSS/XPath selectors used in policy entries make them much easier to edit and maintain across the site.

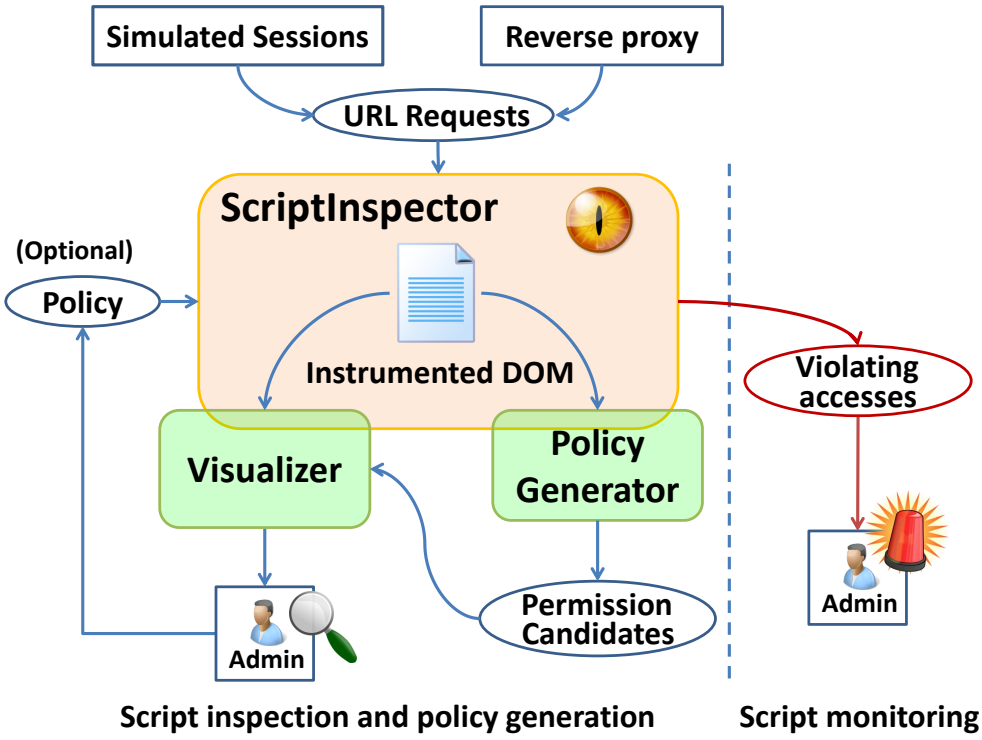


Figure 6.1: Overview

Figure 6.1 gives an overview of our tool chain. The rest of this chapter is organized as follows to describe each component separately: we first discuss the new policy language design that offers better flexibility and maintainability in Chapter 6.1. Then, we present our implementation of ScriptInspector in Chapter 6.2, a modified version of the Firefox browser that is capable of intercepting and recording API calls from third-party scripts to critical resources, including the DOM, local storage, and network. Chapter 6.3 describes the Visualizer, a Firefox extension that uses the instrumented DOM maintained by ScriptInspector to highlight nodes accessed by third-party scripts and help a site administrator understand script behaviors. Chapter 6.4 describes some interesting findings we discovered using the Visualizer. In Chapter 6.5 and 6.6 we present some base policy examples, and then explain the new PolicyGenerator tool which is used to automatically

generate site-specific policies for site administrators. We evaluate the effectiveness of PolicyGenerator and how the entire tool chain performs under an anomaly detection deployment scenario in Chapter 6.7. Finally, we conclude with related works (Chapter 6.8) and deployment scenarios (Chapter 6.9) of the system.

6.1 Policies

A policy used in ScriptInspector is a set of permissions that describe the permissible behaviors for a script. Our goal is to develop policies that are precise enough to limit the behavior of a script to provide a desired level of privacy and security, without needing to generate custom policies for each page on a site. Further, as much as possible, we want to be able to reuse a script policy across all sites embedding that script. The challenge is the way a script behaves depends on the containing page and how the script is embedded, especially in the specific DOM nodes it accesses. Our solution to this aims to make the right tradeoffs between over-generalizing policies and requiring page-specific policies, which we believe would place an unreasonable burden on site administrators.

Policies are described by the following grammar:

⟨*Policy*⟩ ::= ⟨*Permission*⟩*

⟨*Permission*⟩ ::= [⟨*NodeDescriptor*⟩:]⟨*Action*⟩[:⟨*Param*⟩*]

⟨*Action*⟩ ::= ⟨*LocalStorage*⟩ | ⟨*BrowserConfiguration*⟩ | ⟨*NetworkRequest*⟩ | ⟨*DOMAPI*⟩

We explain different types of actions in Section 6.1.1 and node descriptors in Section 6.1.2. Section 6.1.3 addresses the problem of interference between permissions.

6.1.1 Resources

The main resource accessible to scripts is the web page content, represented by the Document Object Model (DOM) in the browser. DOM access includes all reads and modifications to the DOM tree, including node insertion and removal, reading its innerHTML, and setting or getting attributes. For DOM permissions, the *Action* is the DOM API itself (e.g., appendChild) and the *NodeDescriptor* (see Section 6.1.2) specifies the set of nodes on which the action is allowed. The *NodeDescriptor* can be omitted to allow an API to be called on any DOM node. The argument restriction is also optional as some APIs may be called without any argument. Arguments are represented as regular expressions that match strings or nodes. A node used in an argument is matched by its outerHTML. In certain scenarios, the site owner may need to make sure the node in the argument is a node that was created by a third-party script (e.g., a node created by an advertising script to display ad content). To enable this restriction, the [o] (stands for 'owned') tag may be inserted before the

argument. For example, //DIV:RemoveChild:[o]<img src='pic.jpg'> allows the third-party script to remove an image, with the restriction that the image element must have been created by a script hosted by that same party (not from the original host).

In addition to the DOM, scripts have access to other critical resources. These accesses are only allowed if a permission allowing the corresponding *Action* is included in the policy. These permissions do not include *NodeDescriptor*s, since they are not associated with particular DOM nodes.

Local storage. Accesses to **document**.cookie require the getCookie or setCookie permission, while all other accesses to local storage APIs (such as the localStorage associative array and *indexedDB*) require the localStorage permission.

Browser configuration. Third-party scripts may access user-identifying browser configuration, possibly to serve customized scripts to different user groups. However, such information can also be used to fingerprint browsers [104] and identify vulnerable targets. ScriptInspector ensures all accesses to these objects require corresponding permissions. For example, the navigator.userAgent action permission allows a script to obtain the name and version of the client browser.

Network. Ensuring third-party scripts only communicate with intended domains is critical for limiting information leakage. A script can initiate a network request many ways, including calling **document**.write or related DOM insertion APIs, setting the *src* attribute of a *img* node, submitting a form with a carefully crafted *action* attribute, or sending an explicit asynchronous JavaScript request (AJAX). Regardless of the method used to access the network, transmissions are only allowed if the policy includes the network permission with a matching domain.

6.1.2 Node descriptors

A node descriptor is an optional matching mode (introduced later in this section) followed by a node representation:

⟨*NodeDescriptor*⟩ ::= [⟨*MatchingMode*⟩:] ⟨*NodeSelector*⟩

⟨*NodeSelector*⟩ ::= ⟨*AbsoluteXPath*⟩ | ⟨*SelectorXPath*⟩ | ⟨*RegexpXPath*⟩ | ^ ⟨*NodeSelector*⟩

⟨*MatchingMode:*⟩ ::= **sub** | **root**

Absolute XPaths. A DOM node can be specified using an absolute XPath. For example, /HTML[1]/BODY[1]/DIV[1]/ is an absolute XPath that specifies the first DIV child of the BODY element of the page. Absolute XPaths are often useful for matching generic invisible tracking pixels injected by third-party scripts.

Attribute-Based selectors. Nodes can also be specified using Selector XPaths. For example, //DIV[@class='ad'] specifies the set of all DIVs that have the class name ad. This permission is often used to capture the placeholder node under which the third-party scripts should inject the advertisements. Using a selector may compromise security in that there might be other nodes on the webpage that can be accidentally matched using the same selector. Therefore, care has to be taken to make the selectors as restrictive as possible to avoid matching unintended elements. We discuss how the PolicyGenerator can assist administrators to achieve this goal in Section 6.6.1. Another concern is that a third-party script may modify the node attribute to make that node match the selector on purpose. To prevent this, the policy must not allow calls to modify the attributes used in selectors (see Section 6.1.3).

Regular expressions. To offer more robustness and flexibility, our node selector supports regular expressions in XPaths.[1] We found this necessary since many sites introduce randomness in page structure and node attributes. For example, we found that websites may embed an advertisement by defining its placeholder DIV's ID as a string that starts with "adSize−", followed by the size of the ad (e.g. 300x250). We use this descriptor to specify these nodes: //DIV[@ID='adSize−\d∗x\d∗'].

Contextual selectors. A node may be described by another selected node's parent. This is especially convenient when the accessed node does not have any identifying attribute, but one of its children does. We support this by allowing a caret (^) to be added before a node selector to indicate how many levels to walk up the tree when looking for a match. For example, ^^//DIV[@ID='adPos'] specifics the node two levels above the DIV element whose id is adPos.

Similar to the parental context, a node can be described as another selected node's child node. For example, //DIV[@ID='adPos']/DIV[2] specifies the second DIV child of the DIV element whose id is adPos.

Matching mode. Many site-specific DOM accesses happen as a result of advertising and widget scripts injecting content into the page. These scripts often follow similar access patterns, and we define two matching modes that can be used to adjust matching. When no mode is provided, only nodes specified by the given node representation match.

[1] XPaths that accept regular expressions have been proposed for XPath 3.0, but are not yet supported by any major browser.

The *subtree* matching mode matches all children of nodes that are matched by the node selector. For example, **sub**://DIV[@id='adPos'] selects all children of the DIV element whose id is adPos. This matching mode is particularly useful for scripts such as advertising and social widgets that add content into the page. They often touch all children of the placeholder node. However, the node structure inside the injected node may be different between requests, making it hard to describe using the strict-matching mode. In this scenario, a policy that limits script access to a subtree is more plausible.

Root mode covers all nodes that are ancestors to the selected node. For example, **root**://DIV[@id='adPos'] describes all ancestor nodes of the DIV element whose id is adPos.

Listing 2 Script access pattern example

```
/BODY[1]/DIV[3]/DIV[4]/DIV[1]:AppendChild:...
/BODY[1]/DIV[3]/DIV[4]:GetClientWidth
/BODY[1]/DIV[3]:GetClientWidth
/BODY[1]:GetClientWidth
```

We use a commonly seen advertising script access pattern, shown in Listing 2, to explain why this is useful. In this example, the actual meaningful operation is done on the node described in the first line, but the script accesses all of its ancestor nodes. The APIs called on these nodes usually do not reveal much information, e.g. GetID or GetClientWidth. We suspect this is due to the script using a JavaScript library helper such as *jQuery*. To capture this behavior pattern, the root matching mode can be used to match all three accesses to GetClientWidth in Listing 2, as shown here:

/BODY[1]/DIV[3]/DIV[4]/DIV[1]:AppendChild

root:/BODY[1]/DIV[3]/DIV[4]/DIV[1]:GetClientWidth

6.1.3 Permission interference

Attribute-based selectors open the possibility that one permission interferes with another, undesirably extending the collection of matched nodes and allowed APIs. For example, suppose the following two permissions were granted:

//DIV[@class='tracker ']: SetId

//DIV[@id='adPos']:AppendChild

The first permission allows the *id* attribute to be set on any DIV node that has a certain *class*; the second allows appendChild to be called on any DIV node that has a certain *id*. In combination, they allow the script to set *id* attribute of any DIV that has class tracker, thus gaining permission to call appendChild on those nodes.

Manually-created policies need to be carefully examined to exclude the use of certain attributes as selectors if policies from the same third party allows them to be set freely. The PolicyGenerator tool is designed to automatically avoid such conflicts (Section 6.6.1).

Sometimes the site owner wants to grant third-party scripts permission to call any API on certain nodes (e.g., placeholder nodes to be replaced by ads or widgets). However, enabling a wild card action that matches all DOM APIs is dangerous due to possible interference scenarios. To support this scenario, we created a special action denoted by the exclamation mark to indicate all API calls except those that may cause policy interferences.

For example, the permission, //DIV[@class='content']:!, allows the script to call any API on any DIV node with class content, except ones that may set the class attribute to prevent self-interference. Similarly, the permission, //DIV[@id='adPos']:!, allows any API on the DIV with id adPos, except for ones that may set its id attribute. However, when these two permissions co-exist for a script, they will forbid API calls to both setClass and setID, to prevent self and mutual interference. This feature proved to be extremely helpful in our evaluation (Section 6.6.2).

6.2 Inspecting Script Behavior

ScriptInspector is a tool for inspecting the behavior of scripts, focused on monitoring how they manipulate resources and detecting violations of the permissions we defined in Section 6.1. Next, we explain how ScriptInspector records third-party script behavior. Section 6.2.2 discusses how the records are checked against policies or output to logs for admin's inspection.

6.2.1 Recording accesses

ScriptInspector is implemented by modifying Firefox to add hooks to JavaScript API calls. ScriptInspector records all API calls made by scripts that involve any of the critical resources mentioned in Section 6.1.1. Modifications are primarily made in the DOM-JS binding section, and approximately 2000 lines of code were added. The modified browser retains the same functionality as a normal browser, with the addition of access recording capability.

DOM access recording. We modified Firefox's C++ implementations of relevant DOM APIs such as insertBefore, setAttribute and **document**.write to record DOM accesses. Complete mediation is ensured as Firefox uses a code generator to generate the C++ implementation according to the defined interfaces, and our modifications are inserted to the code generator, rather than individual implementations.

ScriptInspector augments each DOM node with a hidden field to record accesses to that node. For each access, the caller identity as well as the API name and optional arguments are recorded in this hidden field. Thus, function call information is preserved with the node for the node's lifetime. We discuss necessary steps to address node removal events in Section 6.2.2.

Recording other actions. For non-DOM resources, we also add hooks to their C++ implementations. Since these calls are not tied to a particular node, the caller records are stored in the hidden field on the **document** object of the page.

Script-injected nodes. To support the [o] tag (Section 6.1.1), ScriptInspector tracks the *ownership* of a node by augmenting node-creation APIs such as **document**.write and **document**.createElement. If a third-party script creates a node and inserts it into the document, we record the responsible party as that node's owner. This feature is especially useful since the host can simply ignore accesses to third-party owned nodes and ensure its own nodes are not used in any arguments.

Attribution. Correctly attributing actions to scripts is important, since policies are applied based on the domain hosting the script. To obtain the identity of the caller, ScriptInspector leverages existing error handling mechanisms implemented in Firefox. Firefox's JavaScript engine, SpiderMonkey, provides a convenient API to obtain the current call stack. It contains information about the script's origin and line number, which are all we need to attribute the accesses. Additionally, a site administrator can provide ScriptInspector with a list of whitelist domains. When a call takes place, ScriptInspector records all third party domains on the stack, except for host domain and whitelisted domains.

The call stack information is sufficient to correctly attribute most introduced dynamic inline code (e.g., through **eval** or **document**.write), but falls short when an inline event handler is registered to an object and is later triggered. ScriptInspector currently cannot handle this scenario and we rely on future Mozilla patches to fix this issue.

6.2.2 Checking policies

To check recorded accesses against a given policy, ScriptInspector introduces the checkPolicy function. When this function is called, ScriptInspector walks the DOM tree, collects all DOM accesses and other accesses stored in the document node, and checks them against the policy. The violating accesses can be used for visualization by the Visualizer (Section 6.3) or as input to PolicyGenerator to use in automatic policy generation (Section 6.5). A log file is also produces for site administrators to inspect manually (an example log file with violations serialized to absolute XPaths is shown in Listing 2).

As opposed to the straightforward design of recording and serializing violations as the accesses happen, our design only records them at access time, but collects and serializes the access records when the page unloads. The additional complexity here may be counter-intuitive; however, such a delay is important for improving the robustness of DOM permissions. For example, a third-party script may obtain the height of a node before the host script sets its id to 'adPos'. In this case, the permission //DIV[@id='adPos']:GetHeight cannot cover this access if the policy is checked immediately, since its id is yet to be set to 'adPos'. Due to the nondeterministic execution order of scripts, checking policy when the access happens is not a good choice.

However, this leaves opportunities to evade mediation if a script reads content of a node and later removes that node. To eliminate this leak, we tweak the APIs that directly (e.g. removeChild) or implicitly (e.g. setInnerHTML) remove DOM nodes: ScriptInspector automatically performs checkPolicy on the to-be-removed nodes and stores the violations into the hidden field of its owner document before they are actually removed.

6.3 Visualization

To visualize the access violations and permission candidates, we built Visualizer, a ScriptInspector extension that takes the instrumented DOM and accesses as input, offers a convenient user interface (shown in Figure 6.2) to display the page content that is read or modified by the third party. We envision Visualizer being used by concerned site administrators who hope to gain insight as to how the embedded scripts interact their websites. With this goal in mind, we next describe using Visualizer from the perspective of a site administrator seeking to understand the behaviors of scripts on her site. We present some interesting discoveries from our experiments visualizing script behaviors in Section 6.4. The Visualizer may also be used to visualize the node collections that are matched by a permission. This can be used together with the PolicyGenerator to help administrators make their decisions (see Section 6.6.1).

Figure 6.2 is a screenshot of visualized accesses at foxnews.com. The left sidebar displays the domains of all third-party scripts embedded on the page (googleadservices.com and googlesyndication.com shown in the screenshot), and the site administrator may click a domain (googlesyndication.com in this example) to expand it and highlight the nodes access by scripts from that domain.

Visualizer classifies DOM accesses into three subcategories — *getContentRecords* (reading the content of a node, e.g. getInnerHTML), *setterRecords* (modifying the attribute or content of the node), and *getterRecords* (reading properties of a node other than the content itself). These categories help administrators quickly locate the accesses of interest. Uses may click on these categories to further expand them into individual accesses.

Users may hover over each entry to highlight the accessed node on the right side. They may also click on a category to see all nodes that were accessed that way (enclosed by double blue border and covered in a faint blue overlay in the screenshot). For accesses to nodes that any third-party *owns* (See Section 6.1.1), the entry is displayed in green and enclosed by a dashed border. In Figure 6.2, the scripts form `googlesyndication.com`, representing one of Google Ad networks, accessed non-content properties of its own inserted ads on `foxnews.com`. Upon seeing this, the administrator may proceed to make decisions to approve embedding Google Ads on their site.

6.4 Findings

We used ScriptInspector and Visualizer to examine the top 200-ranked US websites from `Alexa.com`. Assuming the role as their administrators, we aim to understand how resources are accessed by embedded scripts. We created accounts at these sites and logged into them if possible. We also attempted some typical user actions on these sites, such as adding merchandise into cart for e-commerce websites, and visiting user profile pages for sites that support login.

Browser Properties. Almost all of the tested third-party scripts access a number of generic properties of the browser including the `navigator`, `screen` object, and some properties of root DOM nodes such as the *document* and *body* element. Although this information information could also be used for fingerprinting browsers [104], we consider this behavior reasonable for most scripts since they determine what content to serve based on the browser's vendor and version (`navigator.userAgent`), and user's screen size (`screen.height` or

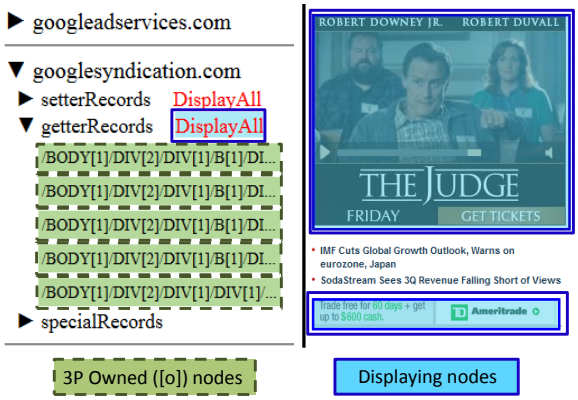


Figure 6.2: Visualizer interface

body. clientHeight).

Network. Another unsurprising but concerning fact is that most scripts we observed send out network requests, at least to their own host servers. Quite a few advertising scripts (e.g., googleadservices.com, moatads.com) also send requests to many other destinations (mostly affiliate networks).

Modifying page contents. Advertising scripts and social widget scripts often modify the container page by injecting or replacing placeholders with content in the page, as seen in Figure 6.2. In addition, multiple tracking scripts insert tracking pixels and scripts from other tracking companies, and advertising scripts may inject other scripts from affiliate programs. However, it is hard for us to determine if such modifications violate the site administrator's expectations.

Reading page content. Finding scripts that read specific page content was less common, but we did observe this for several scripts. The content read ranges from a specific element to the full document. Reading page content may compromise user privacy. Visualizer alerts the site administrators to scripts that read page content by presenting them in a different category, especially when network access also happens in the same session.

Scripts from adroll.com read the DOM element that displays the SKU number of online merchandises on certain e-commerce pages. We initially discovered this when using Visualizer on newegg.com, but later found similar behaviors on dx.com and bhphotovideo.com. We manually examined Adroll's script after using deobfuscation and code beautifiers, and found out that it indeed contains case-switch statements to handle many particular e-commerce site layouts.

According to *Adroll*'s company website, the script is used for ad-"retargeting" purposes. This means the company learns the user's purchasing interests from shopping sites that embed the script, and then displays relevant advertisements to the user on other sites, hoping for a better conversion rate. It seems likely that in this case Newegg agreed to embed the adroll.com scripts, however, its users have no idea their detailed interactions with newegg.com are being tracked by an external service. Newegg's privacy policy states vaguely that it may employ third-party companies to perform functions on their behalf and the third parties have limited access to user's personal information, but nothing that suggests user's shopping actions are immediately revealed to third parties.

Similar behaviors were observed on other e-commerce sites (e.g., autozone.com, eddiebauer.com) involving other third-party service providers (e.g., monetate.com, certona.com, tiqcdn.com). For example, a script from monetate.com that is embedded by EddieBauer's site accesses and transmits user's shopping cart information (Figure 6.3). Figure 6.4 shows a captured form submission request in Fiddler [37] that

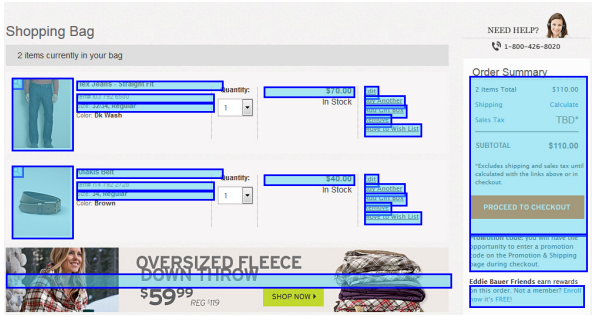


Figure 6.3: Script reading EddieBauer's shopping cart

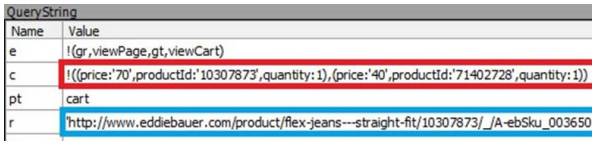


Figure 6.4: Script sending out EddieBauer's shopping cart information

was initiated by a script from monetate.com, with red box highlighting the cart information, and blue box highlighting the product page.

In another case, crwdctnrl.net scripts read user inputs on some sites. On ask.com, the script reads the main text input box used by users to submit questions. On weather.com, it reads the text input in which the user enters their zip code or address to request local weather. The script subsequently sends traffic back to its own server, although we are not able to confirm that the traffic leaks the user input.

Many scripts occasionally read certain properties from a particular node type, for example, scorecardresearch.com scripts read all images' outerHTML on allrecipes.com. Since allrecipes.com's user login information is included as an attribute of user's login avatar, comScore (the company who operates scorecardresearch.com) may obtain user's information through the image accesses. Other commonly accessed tag/attribute pairs include getting the *href* attribute of all *a* elements or the *content* attribute of all *meta* elements. Either of these may contain sensitive information.

The most concerning scenario comes from the sites that embed krxd.net scripts, which *consistently* read *all* content of the embedding host page. This includes all values of all text nodes, and innerHTML of all elements. We have also observed that scripts from this domain send requests to at least 25 different domains, requesting additional scripts and tracking pixels. We find this behavior appalling, and cannot fathom the reasons behind it by looking at its description on Krux's company website.

6.5 Developing Base Policies

The base policy for each script is a policy intended to be used across all sites that embed that script. Obtaining a script's base policy only requires a one-time effort, with perhaps additional work required to update the policy when the script is modified significantly. Hence, it is not necessary to automate developing base policies. In deployment, the base policies could be centrally maintained and distributed, either by the script's service provider (perhaps motivated to disclose its behavior to earn the trust of potential integrators), or by a trusted security service.

In this section, we report on our experience using the logs generated by ScriptInspector to develop base policies for 25 popular third-party scripts. The PolicyGenerator and Visualizer are often not needed to develop base policies (especially for the script's author who should already understand its expected behavior), as the base policy often does not contain specific DOM accesses.

6.5.1 Evaluation method

We manually developed policies for 25 selected scripts. The manual effort required to develop these policies limits the number of scripts we can examine. However, the scripts we examined are the most popular ones among their respective categories, and our experience from Section 6.4 indicates that their behavior is a good representation of extant third-party scripts.

To select the 25 scripts, we first took the 1000 highest-ranked websites from `alexa.com` and visited their homepages repeatedly over a week, crawling pages to collect embedded scripts. We extracted all third-party scripts seen in this process, and sorted them based on the number of occurrences. We took the top 25 on the list and manually visited 100 sites which embed each script, sampled randomly from the 1000 sites on the `alexa.com` list.[1]

Of those 100 sites, 77 include user registration and login functionality. For each of these, we manually created a test account and logged in to the website to mimic a typical user's experience. After user sessions are prepared, we first visit each site's homepage, follow links to navigate to several subpages, and call **document**.checkPolicy to output an access report. We repeat this process until no new accesses are seen in five consecutive requests. We then manually extract the most commonly observed accesses to form the base policy. Our overall goal in writing base policies is to allow all behaviors that an integrator is likely to encounter without over-generalizing. The base policy should contain mostly special API accesses and generic DOM

[1] In selecting the container sites, we excluded inappropriate sites including those which the embedded scripts do not access anything, trivial sites that have few subpages and content, sites with objectionable content (e.g., porn), and foreign language sites for which we were unable to register and login. The full list is available at `ScriptInspector.org/sites`.

Policy 1 Google Analytics Base Policy

```
/HTML[1]/BODY[1]:GetId; /HTML[1]/BODY[1]:ClientHeight; /HTML[1]/BODY[1]:ClientWidth;
/HTML[1]:ClientHeight; /HTML[1]:ClientWidth;
navigator.javaEnabled, navigator.language, navigator.plugins; navigator.cookieEnabled; navigator.userAgent;
screen.height; screen.width; screen.colorDepth
GetCookie; SetCookie
NetworkSend:doubleclick.net; NetworkSend:google.com; NetworkSend:google−analytics.com
/HTML[1]/HEAD[1]>InsertBefore:\[o\]
    <script[^<>]*></script>
```

accesses to the root elements of the page (*document*, *body* and *head*). However, if certain DOM accesses with fixed patterns are seen consistently, they are also included in the base policy.

6.5.2 Base policy examples

For clearer presentation, we discuss base policies organized by grouping scripts into four categories: *analytics*, *advertising*, *social widgets*, and *web development*.

Analytics. Analytics scripts are typically transparent to the site visitors and do not affect the functionality or visual output of the embedding webpage. Their base policies include a fixed group of sensitive APIs such as setting and reading **document**.cookie, but not any specific DOM accesses.

As the most frequently embedded script by far, Google Analytics exhibits a very stable behavior pattern described by Policy 1. Other than the final permission, all its accesses can be categorized into three categories: 1) Generic DOM access: reading the size of the *body* element; 2) special property access: testing the browser's configuration, reading and writing cookies; and 3) network access: sending information back to Google Analytics servers via setting the src property of an image element. This reassures the site owner that the Google analytics script is not accessing site-specific information or making content changes to the site. The final permission is needed because the Google Analytics script inserts dynamically-loaded scripts into the page. The \[o\] limits the insertions to nodes owned by the script. The parameter is a regular expression that specifies the element type inserted must be a script. Note that the network policies still apply, restricting the domain hosting the dynamically-loaded script. Also, recall that this same base policy still applies to any scripts from the same domain due to our access attribution implementation, so dynamically loading scripts does not provide extra capabilities.

Another popular embedded script, Quantcast analytics, exhibits similar behavior with the addition of reading the *content* attribute of all *meta* elements. Common practice suggests using these attributes to mark keywords for the document, so Quantcast is likely collecting these keywords. In sites that embed Chartbeat analytics, the *src* attributes of all *script* elements on the page are read by Chartbeat, along with *href* and *rel*

Policy 2 Google Adsense Base Policy Excerpt

```
//:GetAttribute:data−ad−container; //:GetId
//DIV[@id='div−gpt−ad−.*']:!
/HTML[1]/BODY[1]:document.write:
    <img src='googleadservices.com'/>
```

attributes of *link* elements. This is a somewhat surprising, yet common behavior that was also observed for several other scripts. Chartbeat also maintains a heartbeat message protocol with the backend server and multiple request-response pairs are observed per session.

All the major analytics scripts appear to have sufficiently limited behaviors that containing sites will not need site-specific policies. Base policies can cover all observed behaviors without including any permissions that allow access to obviously sensitive resources and content-changing APIs.

Advertisements. Google offers the most widely used advertising service through AdSense and DoubleClick. Policy 2 is an excerpt of the policy for `googleadservices.com`, whose behaviors are representative of most advertising scripts.

The AdSense script accesses several browser properties similar to the analytics scripts, but also injects advertisements into the page and often inserts a hidden empty frame or tiny image pixel for bootstrapping or tracking purposes. The tracking pixels are always injected into the *body* or *document* element, and the last permission in Policy 2 allows such behavior.

The node where the actual advertisements are injected, however, varies from site to site. As a result, the base policy only covers the most popular use, described by the `//DIV[@id='div−gpt−ad−.*']:!` permission. This allows any APIs to be called on a node whose id starts with *div-gpt-ad-*, except those that may modify attribute names in node descriptors of itself and any other permissions that belong to the same script. Behaviors of other ways of integrating AdSense need to be covered by site-specific policies (Section 6.6).

Scripts from `moatads.com`, `rubiconproject.com`, `adroll.com` and `adnxs.com` all exhibit similar behaviors to Google advertising scripts. In addition, the `moatads.com` scripts occasionally read the *src* attribute of all *script* and *img* elements. This behavior is dangerous and could result in information leakage. Since it is observed on many containing sites, however, we add it to the base policy despite the fact that it may only happen once in many visits. Scripts from `betrad.com` also access localStorage APIs, presumably adding an extra tracking approach should the user reject or delete their cookie.

Compared to analytics scripts, the advertising scripts exhibit behaviors that vary substantially across sites. Hence, additional site-specific permissions are often necessary to accurately describe their behavior. The base policies for ad scripts also include more permissions, such as reading attributes of all nodes with a

certain tag name and appending ad contents and tracking pixels to the *body* element.

Social widgets. Social widget scripts behave similarly to advertising scripts. As an example, Policy 3 shows the base policy for `twitter.com` scripts which includes permissions for getting and setting twitter-specific attributes, accessing content size, injecting content, and replacing placeholders of the page. As we see in Section 6.6, social widget scripts often require site-specific policies.

Policy 3 Twitter Base Policy Excerpt

```
//:GetAttribute:data−.*; //:SetAttribute:data−twttr−.*;
//:ReplaceChild:[o]
    <iframe data−twttr−rendered='true'></iframe>
    <a class='twitter−share−button'></a>
sub:^//A[@class='twitter−share−button']:GetAttribute:height
```

Web development. Finally, we consider web development scripts such as web A/B testing scripts from `optimizely.com` and the jQuery library hosted on `googleapis.com`. Due to their broad purpose, the behavior of these scripts differs significantly from those in the previous three categories. For example, the `optimizely.com` script modifies all "comment" buttons on `guardian.com`, inserts a style element on `magento.com`, but did not access any part of the DOM on `techcrunch.com`. What these scripts do is depends a great deal on how the site owners use them.

Effective base policies cannot be developed for these scripts — their behavior varies too much across sites and even across requests on the same site. Web developers using these scripts would need to develop a custom policy for them, based on understanding what the script should do given their intended use.

6.6 Developing Site-Specific Policies

Site-specific policies are needed for scripts that require different permissions depending on how they are embedded. To aid site administrators in developing appropriate site-specific policies, we developed the PolicyGenerator tool to partially automate the process. The PolicyGenerator generates permission candidates based on violations to existing policies reported by ScriptInspector. The site administrator can use Visualizer to examine the candidate permissions and either select appropriate permissions to include in the script's policy or decide not to embed the script if it requires excessive access. Section 6.6.1 introduces the PolicyGenerator tool and Section 6.6.2 reports on our experience using it.

6.6.1 PolicyGenerator

With the base policies in place, the site-specific permissions typically need to allow access to specific DOM elements such as placeholders for advertisements. Our key observation behind the PolicyGenerator is that although absolute properties of these nodes vary across different pages and requests, good selector patterns can often be found that hold site-wide, due to consistencies within the web application design. For example, the DOM representations of the access nodes often have some common property such as sharing a class which other elements on the same page do not have.

For example, consider these two CSS selectors describing advertisements observed on two requests to mtv.com:

```
div#adPos300x250
div#adPos728x90
```

These ad containers have different *id*s, but their *id* always starts with the string 'adPos', followed by a pair of height and width parameters. Patterns like these are straightforward for site administrators to understand and can be inferred automatically from access reports.

To use PolicyGenerator, the user starts ScriptInspector with the PolicyGenerator extension. ScriptInspector is configured to load base policies from a file to provide the initial base policy for the scripts. The user visits the page of interest, and then clicks on the PolicyGenerator extension button to start the process. PolicyGenerator generates permission candidates, which are presented to the user using Visualizer. The user can then select permissions to add to the site-specific policy for the script. The user can continue visiting additional pages from the site, invoking PolicyGenerator, and refining policies based on automated suggestions.

When the user invokes PolicyGenerator, it obtains a list of violating records from the instrumented DOM by calling **document**.checkPolicy. It initially generates a set of simple *tag permission* candidates which match DOM accesses only by their node name, API name, and arguments, but not by attribute-value pairs. This searches for simple permissions like //DIV:getID. These candidates are selected by counting the number of accesses to each node name and comparing it to the total number of occurrences of that tag in the document. If the ratio reaches a threshold (customizable by the user; the default is 25% which we have found works well), a tag permission is proposed for that API call. Accesses that match this permission are removed from the set of violating accesses for the next phase.

Finding good permission candidates that involve complex DOM selectors is more challenging, but important to do well since these are the permissions that would be hardest for a site administrator to derive without assistance. In Section 6.4, we observed that most accesses by a third-party script can be divided into two categories: those that happen on "nodes of interest", and those that can be covered by adding root, parent or

sub prefix to the nodes of interest. The node of interest often has content modification APIs called upon them (e.g., appendChild, setInnerHTML), or is the deepest node accessed along the same path with other accessed nodes. For example, the first node listed in Listing 2 would be a node of interest because it's the deepest node accessed.

Following this observation, the next step is for PolicyGenerator to develop a set of selector patterns using attribute names for all nodes of interest, excluding those that could interfere with current permissions (as described in Section 6.1.3). Then, it produces an attribute-value pair pattern candidates for each node of interest. Our implementation uses heuristics to synthesize four types of patterns for each attribute name candidate: *matches exactly*, *matches all*, *starts with*, and *ends with*. For the latter two pattern types the generator calculates the strictest restriction without sacrificing the number of accesses it matches, and then selects the best pattern type that matches the most violations out of the four. After a best pattern and pattern type has been determined for each attribute name, the generator sorts them by the number of matched violations of each attribute name's best pattern, but excludes those which also accidentally match any untouched nodes.

We provide an option to allow the generator to accept a pattern if it matches some untouched nodes below a threshold ratio. An example of where this is useful occurs with advertising scripts that do not fill not all advertisement spaces in one response, but instead leaves some untouched for later use. The decision regarding whether to accept over-matched patterns is left to the developer, but it is important that such patterns are clearly presented by Visualizer.

The best qualifying permission is then added to the set of permission candidates that will be presented to the user, and all accesses matching that permission are removed. The process repeats until there are no nodes of interest left.

If any violating accesses remain after all nodes of interest have been removed, PolicyGenerator examines if the remaining unmatched violations involve either a parent, ancestor, or descendent of any node of interest. If so, a corresponding parent, **root**, or **sub** permission is proposed.

It is ultimately up to the developer's discretion to accept, reject, or tweak the generated permissions. To ease this process, the policy candidates can be viewed using Visualizer. The presentation is similar to what is described in Section 6.3, and developers may click on individual permissions to view the nodes they cover.

In the next section, we see that although the initial guessed permissions are not always perfect, only minor tweaks are needed to reach effective site-specific policies for most scripts and sites. This manual approval process is important for developing good policies, but also valuable since our goal is not to produce policies that completely describe the behaviors of all scripts on the site, but to help site administrators identify scripts that are exhibiting undesirable behaviors on their site.

6.6.2 Adjusting permission candidates

We want to understand how much work is required to develop effective site-specific policies for a typical website using our tools, assuming base policies are provided for all scripts on the site. In this section, we describe manual efforts involved in using PolicyGenerator on typical sites and show some examples of site-specific policies. We defer the discussion of quantitative results to Section 6.7.

To evaluate the PolicyGenerator from a site administrator's point of view, we first set up a crawler robot that visits a new set of 100 test sites using ScriptInspector. The goal of the robot is to simulate regular users' browsing actions to explore site-specific behavior of embedded scripts. Given the URL of a test site, the robot first visits that URL using the ScriptInspector, and waits for 30 seconds after the page has finished loading (or times out after 60 seconds). Then, it navigates to a randomly selected link on the page whose target has the same domain as the test site. We chose not to navigate the robot away from each page right after it completes loading because certain third-party scripts may not start to execute after the page has loaded. Upon page unloading, the robot calls **document**.checkPolicy to log the violating traces.

The robot repeats the navigation until it has successfully navigated five times for that test site, before moving on to the next. If the robot cannot find a link on the current page (or if the browser crashes), it goes back to the homepage and resumes from there. The scan explores each site five levels deep to simulate user browsing behavior. Finally, after the robot successfully navigated 5 times for all 100 sites, it restarts the whole process from the first site.

Whenever ScriptInspector outputs a violation to existing policies on a site, further visits to that site are postponed until we manually examine the violation using PolicyGenerator and add necessary permissions to the policy. This is to prevent similar violations being recorded multiple times.

We ran the evaluation from 28 December 2014 to 6 February 2015, a total of 40 days. For each site, the experiment contains two stages: an initial stage to train a reasonably stable model, and a testing stage to estimate how many violations would be reported if the policy were deployed. We initially define the training phase to last until after ScriptInspector completes 100 requests to that site without recording a single alarm (we show how this threshold can be tuned in Section 6.7.2).

Permission adjustment examples. Depending on the complexity of the site, generating and tweaking the policy for one site may take from a few minutes up to half an hour based on our experience. The cost varies due to factors such as the number and complexity of scripts the site embeds, and how much pages differ across the site (for example, in where they place advertisements). The results here are based on the first author's experience, who, as creator of PolicyGenerator and Visualizer, is intimately familiar with their

operation. A developer using PolicyGenerator for the first time will need more time to become familiar with the tool, but we expect the tool would not be difficult for a fairly experienced web developer to learn to use.

We evaluate the effort required for each manual permission by considering three of the most common ways auto-generated permissions needed to be manually changed. The first example, taken from mtv.com's policy for doubleclick.net, is a result of auto-generated permission over-fitting the accesses on this particular request:

//DIV[@id='adPos300x250']

It includes overly-specific size properties, and was fixed by manually replacing them with a regular expression pattern:

//DIV[@id='adPos\d*x\d*']

The other way to relax over-fitting permissions is to increase the matching threshold PolicyGenerator uses to eliminate permissions. This threshold controls the number of nodes that may be matched by a generated node descriptor but are not accessed by the third-party script. Adjusting this threshold is often enough to eliminate further manual effort.

For example, these are the three permission candidates proposed for foxnews.com by PolicyGenerator, with matching threshold set to default value 1 which does not allow any additional node matches:

//DIV[@id='trending−292x30']

//DIV[@id='board1−970x66']

//DIV[@id='frame2−300x100']

//DIV[@id='stocksearch−292x30']

The PolicyGenerator did not yield the more elegant and representative permission,

//DIV[@class='ad']

because another node that has class 'ad' is not accessed. After the threshold is raised to 2 (i.e. the selector is allowed to match at most twice the number of accessed nodes), this better permission is proposed.

However, increasing the threshold may adversely cause PolicyGenerator to generate very broad permissions. For example, if all the advertisement content are injected into borderless DIV frames, //DIV[@frameborder='0'] could be a potential permission candidate, but a rather meaningless one that may match other nodes containing sensitive information. To avoid this issue, PolicyGenerator normally gives more weight to attributes like *class* and *id*, and propose them more often; however, if an overly broad permission is indeed proposed, the administrator may need to ignore the candidate and look into the DOM structure and find identifiers from its parents or children to produce a better permission such as //DIV[@class='ad'].

Policy 4 Site-specific policy for ticketmaster.com
(The getSize action is a special DOM permission, it includes APIs related to size and position information such as getClientHeight and getScrollWidth.)

```
−googleadservices.com & doubleclick.net−
    //DIV[@class='gpt−ad−container']:AppendChild
    //DIV[@class='gpt−ad−container']:getSize
−facebook.net−
    Send:ticketmaster.com
```

After changes are made to a permission, Visualizer will highlight the node collections matching the adjusted permission. The site administrator can examine the highlighted nodes and determine if the permission should be used.

Another common scenario that requires manual attention is when the PolicyGenerator over-emphasizes particular attributes. For example, all cnet.com ad placeholders have an attribute named data−ad, which makes it a good descriptor to use for site-specific permissions. However, because PolicyGenerator favors *id* and *class* attributes over others, it generates complex and meaningless policies across different pages using *id* and *class* as selector attribute names.

Site-specific policy examples. Here we show a few examples of site-specific policies to illustrate the integrity and privacy properties that can be expressed.

Ticketmaster.com is a ticket-selling website. Policy 4 shows the site-specific policy extensions that were needed for three scripts embedded on this site, one for Facebook and the other two for Google ad services. The Facebook permission allows its script to send network requests back to the host domain, ticketmaster.com. Although this behavior may be safe and intended by the site administrator, a site-specific policy is required because this behavior is not part of the script's base policy.

The site-specific permissions for googleadservices.com and doubleclick.net scripts are based on the same node descriptor: //DIV[@class='gpt−ad−container']. This permission entry is necessary because it is specific to the website and different from the most popular implementation //DIV[@id='div−gpt−ad−.∗'] covered in the base policy. The two permissions together give scripts from the Google ad sites permission to write to the matching nodes, as well as to read their size information. Other than this, the embedded scripts are not allowed to read or write any content, offering strong privacy and integrity for ticketmaster.com.

However, not all permissions match accessed nodes perfectly. For example, we added this site-specific Google ad permission for theverge.com:

```
//DIV[@class='dfp_ad']:document.write
```

[1]The getSize action is a special DOM permission, it includes APIs related to size and position information such as getClientHeight and getScrollWidth.

This entry matches a total of ten nodes in the homepage, but only four nodes are actually accessed on the page. This means the selector over-matches six additional nodes. After a closer examination of the these nodes, we confirmed that none of them contain any sensitive content and four are adjacent to nodes that contain the advertisement.

Finally, we are not able to obtain meaningful and robust site-specific policies for two sites (forbes.com and buzzfeed.com), as their advertisement integration lacks generalizable node selector patterns. In addition, omtrdc.net scripts seem to be reading and writing all nodes on certain pages on lowes.com. In the above cases, whitelisting the problematic script domain as trusted seems to be the best solution. This prevents the frequent violations for these scripts, but enables ScriptInspector to restrict the behavior of other scripts in the page.

6.7 Policy Evaluation

In this section, we analyze experimentally how many site-specific permissions are required for each site, their distribution in third-party domains, the length of the optimal training phase (for the deployment scenario described on the right side of Figure 6.1), and the robustness of trained policies. Although our results reveal that producing good policies can be challenging, they provide reasons to be optimistic that the effort required to produce robust policies is reasonable for nearly all websites.

6.7.1 Policy size

Of the 100 sites tested, 72 sites needed at least one site-specific permission. Table 6.1 shows the number of sites requiring at least one site-specific permission for particular script domains (scripts embedded in fewer than ten sites not included in the table). The table is sorted by the fraction of sites embedding scripts from a given domain that needed site-specific permissions. For example, of the 61 sites from our sample that embed doubleclick.net, we found 48 of them needed site-specific permissions, while 13 of sites only needed the generic base policy.

Many sites need site-specific permissions for the advertising scripts (doubleclick.net, googleadservices.com). This is not surprising since they are injecting advertisements into different locations of the page for different sites. Only those serving as beacons for Google Ad networks (tracking user's browsing history as opposed to injecting advertisements) or those which embed the scripts using the conventions covered by the base policy do not need site-specific permissions.

Similarly, social widgets (70% for twitter.com and 32% for facebook.net) also require a high number of site-specific permissions. The reason that Twitter's number is significantly higher than Facebook's is partly

Script Domain	Sites Embedding	Sites Needing Permissions	Percentage
twitter.com	41	36	88%
googleadservices.com	72	57	79%
doubleclick.net	61	48	79%
moatads.com	16	7	44%
2mdn.net	17	6	35%
betrad.com	16	5	31%
facebook.net	66	20	30%
doubleverify.com	11	2	18%
adroll.com	14	2	14%
rubiconproject.com	10	1	10%
chartbeat.com	17	1	6%
google-analytics.com	83	4	5%
scorecardresearch.com	40	1	3%
newrelic.com	32	0	0%
quantserve.com	25	0	0%
criteo.com	17	0	0%
Total (24 domains)	580	207	36%

Table 6.1: Scripts needing site-specific permissions.

because Facebook's content insertion behavior can be covered by the base policy //DIV[@id='fb−**root**']>!, while most content insertion behaviors of Twitter are more flexible and cannot be covered by a simple base policy.

On the contrary, analytics scripts rarely require site-specific policies. Google Analytics is embedded by 83 of the 100 test sites, but only four sites needed site-specific permissions. None of the 25 sites embedding QuantServe Analytics required any site-specific permissions. The low fraction of sites needing specific permissions for analytics scripts is consistent with our observations in Section 6.4.

The overall number of permissions needed is manageable per site. We count permissions based on their DOM node representation (for permissions involving DOM access), but not the API called or arguments used. So //DIV[@id='a']:GetAttribute and //DIV[@id='a']:SetAttribute would be counted as one site-specific permission, but //DIV[@id='b'] and //DIV[@id='a'] would count as two.

A total of 436 total site-specific permissions are added for all 100 sites, so each of the 72 sites that needed at least one permission needed an average of 6.1 permissions. The largest number was for mlb.com which embeds many advertisements and needed 26 site-specific permissions, followed by 14 for businessweek.com and people.com.

Very few individual script domains required more than two permissions on average for embedding sites. The highest was 4.16 permissions per embedding site for 2mdn.net scripts (embedded on 17 sites).

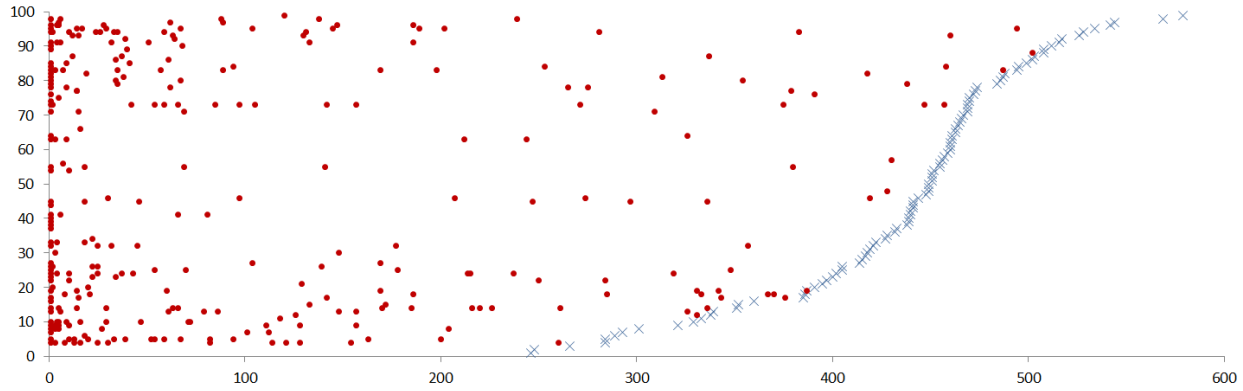


Figure 6.5: Policy convergence vs. number of revisions

Other frequently used scripts requiring more than two permissions per embedding site include moatads.com, doubleclick.net, krxd.net, facebook.net, serving-sys.com, and googleadservices.com. Their site-specific policies consist of mostly read and write accesses to designated ad or social content placeholders, with few additional size and location queries for surrounding elements.

The number of site-specific permissions per site gives some sense of the manual effort required, but the amount of effort also depends on how close the permissions generated by PolicyGenerator are to the desired permissions. Of the 436 total site-specific permissions needed across all tested sites, 78 (18%) were created manually from scratch. Only 28 of the 100 sites needed any manually-created permissions, and only ten sites required more than one. Based on this, we believe the human effort required is low enough to be reasonable for deployment by major sites.

6.7.2 Policy robustness

Since the policies are developed based on the script behaviors observed for a small number of requests on a few of the site's pages, there is a risk that the scripts exhibit different behaviors on different requests and generate too many false alarms to be useful in practice. To understand this, we study the policy convergence speed and alarm rates. We also selected some suspicious alarms and discuss them in Section 6.7.3. Section 6.7.4 considers several violation scenarios due to major updates in host sites or third-party scripts.

Figure 6.5 show all the alarms ScriptInspector reported in the experiment. Each site corresponds to a vertical coordinate in the figure, and they are sorted according to the total number of requests executed (due to different ending stage of training phase and stoppage time between reported alarm and manual inspection, the number of requests done averages to 434 per site but varies from 246 to 579). The horizontal axis represents the sequence number of requests, and alarms are marked along this axis to show when they occurred. The majority of alarms are issued at the beginning of the training phase. The total number of

alarms ScriptInspector reported for all 100 sites is 301 over 40 days, making the average less than three alarms per site per month. The highest number of alarms is the 11 alarms reported from `mlb.com`. Of the 100 test sites, 28 issued no alarms at all, which means they do not need site-specific policies. Policies of more than half (57) of the 100 sites converge within two policy revisions, and 83 sites converge within six policy revisions.

Training phase duration. A longer-lasting training phase observes more behavior but also requires more time and effort. Figure 6.6 shows the relationship between training phase duration and alarm rates. Setting the training phase to conclude after executing 177 requests without an alarm will yield 10% of the total alarms. This number is 20% for 114 requests, and 70% for 80 requests. Based on this, we conclude that ScriptInspector has observed most relevant behavior after 200 alarm-free requests and suggest using this for the convergence threshold after which a learned policy would be transitions to deployment.

Reasons for alarms. We manually examined all the alarms. Violations can be roughly classified into two general categories, with a few exceptional cases. The most common reason for a violating record (173 out of 301) is that a new type of advertisement shows up which was not seen in previous requests. For example, *skyscraper* (long and vertical) ads are only shown on a specific part of the site, whereas other parts show a *banner* (long and horizontal) ad. The second category is because of social network widgets (93 out of 301). A common scenario is that news sites serve articles from different sources which do not necessarily share the same coding pattern and content layout. This could result in scripts from `twitter.com` injecting Twitter feeds into containers with different attributes. Occasionally, the violation is a result of apparently suspicious behavior or a major script update, discussed in the next two sections.

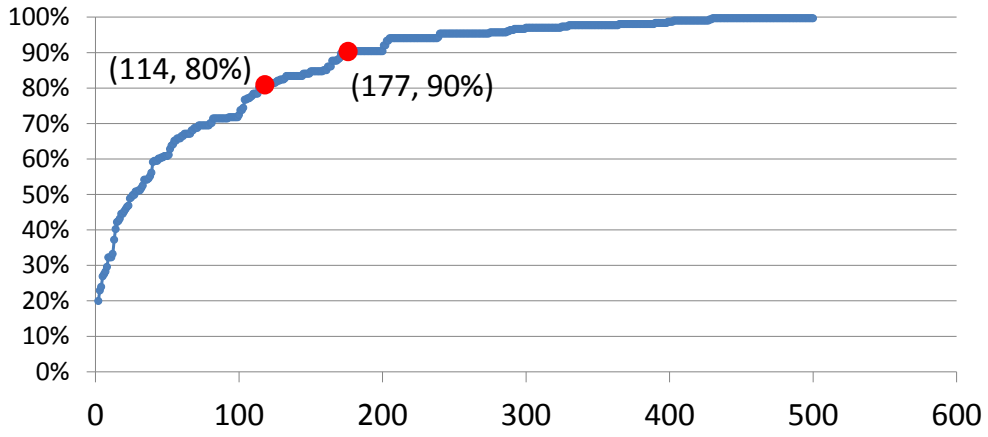


Figure 6.6: Training phase duration vs. Alarm rates

6.7.3 Suspicious violations

In the robustness experiment, we found that on rare occasions the Facebook scripts load other ad networks and analytics scripts, which raised massive numbers of alerts. In some sites such as staples.com and dailymotion.com, Facebook scripts access the exact same information as did krxd.net and enlighten.com, essentially reading the entire page content. In other cases, Facebook scripts read the action attribute of all forms on the host page (e.g., goodreads.com, hostgator.com). This behavior is only visible during certain periods of time, and we observed this access on multiple websites only at the start of the experiment as well as 18 days after. In extremely rare occasions (tutsplus.com and drupal.org), ScriptInspector caught that Facebook scripts read the value of user's user name and password input on the host page, which is disturbing. We are not sure if this is intended by the site owners, and suspect they are unaware of this behavior.

We have also seen Google advertising scripts read the entire page contents by calling documentElement.innerHTML. This behavior was only observed once, and only on nfl.com. This could be a bug in an advertising script, or it could indicate that Google advertising is crawling the page content and indexing it for future contextual targeting.

6.7.4 Impact of major updates

Throughout the course of our evaluation, we saw major changes to three third-party scripts which resulted in multiple duplicate alarms reported across most sites embedding them. Particularly, facebook.net scripts began reading properties (e.g. *href*, *rel*) of all *link* elements on the page since 30 December 2014, and doubleverify.com scripts showed similar behavior changes since 5 February 2015. Additionally, krxd.net scripts began injecting an invisible *DIV* element into all pages embedding it since 26 January 2015. We handled these violations by updating their base policies since the new behaviors are not site-specific. These cases show that while major third-party scripts changes may require updates to policy, they only occur rarely and trivial changes to base policy are often enough to cover the new behavior.

It is harder to determine whether a page went through major changes over our experiment. However, we did see this for two sites, theblaze.com, which added an advertising slot on their frontpage, and inc.com, which redesigned its general user interface. For both cases, we added an additional advertising container permission to their site-specific policies. It can be annoying to the developers to approve new permissions when a major site update happens, however, we do not expect policy-breaking changes frequently for most sites. Further, we argue that it may be useful to site administrators to learn when a change impacts the behaviors of scripts significantly enough to require a policy change.

6.8 Prior Works

ScriptInspector is directly inspired by our previous work [66] and intends to improve on some of its limitations, as described in the beginning of this Chapter. Therefore, we have already covered many related works in Chapter 5.1, including sandboxing JavaScript behavior by extending browsers, rewriting scripts, and encrypting content. In this section we only focus on comparing ScriptInspector with previous tools and works that help site administrators understand third-party JavaScript behavior.

Script behavior visualization. Several tools present script behaviors in a user-understandable way. Wang et al. [2] use a browser-based interface to explore relationships between requests and discover vulnerabilities. Popular browser extensions like Ghostery [105] and Abine [106] help users and site administrators understand what third-party services exist on the current page. A recent Chrome developer tool [107] informs a user what resource a Chrome extension is accessing on the page, albeit at coarse granularity. The success of these tools supports our hope that our tool chain can be of great value to web developers in understanding scripts and developing policies.

6.9 Deployment

In this section, we discuss some possible deployment scenarios. So far, we have focused on the scenario where a site administrator wants to understand the behaviors of embedded scripts on a web site to protect clients from privacy compromises by malicious or compromised scripts and ensure the integrity of the site from unintended modifications. The tools we developed could be used in several other ways, discussed below.

Access visualization. Visualizer can be used by either an interested web developer or sophisticated user. After examining the accessed resources, a developer can make an informed decision to choose the service provider that most respects site integrity and user privacy. A sophisticated user may use extensions like noscript [108] to block third-party scripts with suspicious behaviors revealed by Visualizer.

Policy generation service. A third-party service provider or dedicated security service could develop base policies for commonly-used scripts. A cooperating third-party service provider may make site-specific policy generation part of the implementation process. For example, policies can be inferred by analyzing the implementation code. In a less ideal scenario, the policy generation service could provide a description of how to generate a site-specific policy for the script based on properties of the embedding site. Site administrators

would then use that description to manually generate their own site-specific policies.

Access monitoring. After a policy has been generated, we envision two ways a site administrator can monitor future accesses. An easy-to-adopt approach is to continue running ScriptInspector with the policies on simulated sessions. An alternative approach is to sample real world traffic using a reverse proxy and forward sampled requests to run in ScriptInspector with user credentials. The second approach gives higher confidence that the integrity and privacy properties are not violated in real sessions, but risks interfering with the normal behavior of the site if repeating requests alters server state. For both cases, the site administrator would examine alerts and respond by either changing policies or altering the site to remove misbehaving scripts. More security-focused sites could automate this process to automatically remove scripts that generate alarms.

Policy enforcement. Our prototype ScriptInspector is not intended to be used by end users to enforce the policies at runtime mainly due to its high runtime overhead. The key reason is that each DOM API access requires at least one stack computation and node removal APIs require walking the subtree and compute access violations. However, some policies may be enforced by other browser security mechanisms, for example, scripts from a particular domain can be blacklisted by content security policy, which is currently supported by major browsers. We envision a future, though, where a more expressive analog to CSP is adopted by popular browsers and servers can provide headers with restrictive policies for embedded scripts that would be enforced by browsers at runtime. This would offer the best protection, ensuring that the actual behavior of the dynamically-loaded script on client's browser does not behave in ways that violate the server's script policy.

Chapter 7

Conclusion

This chapter begins with a summary of the thesis work (Chapter 7.1) and closes with Final remarks in Chapter 7.2.

7.1 Summary

Protecting applications enabled by third-party services is a difficult yet crucial task. In this dissertation we have presented our solutions to attack two aspects of this problem — protecting integrated applications against outside threat, and protecting host applications against untrusted third-party code.

Our work on explicating SDKs and APIs has shown that discovering vulnerabilities and documentation mistakes in third-party services can be done in a systematic fashion. By modeling different system parts (concrete, restricted, and free modules) using different approaches, the vulnerabilities and pitfalls can be found in two ways — one can learn about logic of critical parts of the system in the modeling process and discover vulnerabilities on their own; or the model checker can present counterexamples to the researcher and the counterexamples can be used to recreate the real-world attack vector.

The SSOScan tool we developed can be used to quickly check applications for the discovered vulnerabilities and implementation mistakes. Ideally, SSOScan should be deployed at application's distribution center such as Apple app store or Facebook app center. The testing involves zero to minimum human assistance, usually lasts less than 10 minutes, and the success rate of automation is very high (80%). Although SSOScan prototype only checks for five vulnerabilities and only for Facebook Single Sign-On implementation, we believe that the automation framework we developed can be easily extended to check any vulnerabilities for any service, unless it requires application-specific user interaction pattern to simulate the attack vector.

By inserting access mediation code into Chromium and Firefox browsers, we showed that fine-grained access monitoring and restriction on host resources can be achieved compared to same-origin policy which only offers an all-or-nothing access control model. Our tools give site administrators a clear understanding of embedded third-party scripts behavior, boosting their confidences about the security and privacy of their websites.

The various tools we developed can also enforce access control policies at client side or perform intrusion detection based on the policies. To alleviate burdens from the site administrators, our tool chain also supports automatic policy generation; This automatic process only requires minimum human assistance, and the policy candidates proposed are easy to understand and maintain over time.

7.2 Conclusion

We present solid steps toward improving the security and privacy of third-party service enabled applications. We developed automatic techniques to discover SDK vulnerabilities, documentation mistakes, and vulnerable implementations; We also help developers understand, monitor, and restrict the behavior of third-party web scripts. Although many challenges still await to be solved, the tools and techniques we developed advances the state-of-the-art and we anticipate they will shed light on future research to better protect third-party service integrations against security and privacy threats.

Bibliography

[1] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-Scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[2] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[3] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[4] Wikipedia. Mashup (Web Application Hybrid). http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)/.

[5] Wikipedia. Same-Origin Policy. http://en.wikipedia.org/wiki/Same-origin_policy.

[6] Phillipa Gill, Vijay Erramilli, Augustin Chaintreau, Balachander Krishnamurthy, Konstantina Papagiannaki, and Pablo Rodriguez. Follow the Money: Understanding Economics of Online Aggregation and Advertising. In *Proceedings of the 2013 Internet Measurement Conference*, 2013.

[7] Internet Engineering Task Force. The OAuth 2.0 Authorization Framework. http://tools.ietf.org/html/rfc6749.

[8] OpenID Foundation. OpenID Foundation Website. http://openid.net/.

[9] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[10] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[11] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[12] Diomidis Spinellis and Panagiotis Louridas. A Framework for the Static Verification of API Calls. *Journal of System and Software*, 80(7):1156–1168, July 2007.

[13] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of the 8th European Software Engineering Conference*, 2001.

[14] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web Service Interfaces. In *Proceedings of the 14th International Conference on World Wide Web*, 2005.

[15] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.

[16] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically Discovering Pointer-Based Program Invariants. Technical Report UW-CSE-99-11-02, University of Washington Department of Computer Science and Engineering, 1999.

[17] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, 2000.

[18] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[19] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[20] Jinlin Yang and David Evans. Dynamically Inferring Temporal Properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[21] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[22] Westley Weimer and George C. Necula. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[23] Claire Le Goues and Westley Weimer. Measuring Code Quality to Improve Specification Mining. *IEEE Trans. Softw. Eng.*, 38(1):175–190, January 2012.

[24] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[25] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *Proceedings of the 25th Computer Security Foundations Symposium*, 2012.

[26] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, 2001.

[27] S. Pai, Y. Sharma, S. Kumar, R.M. Pai, and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. In *Proceedings of the first International Conference on Communication Systems and Network Technologies*, 2011.

[28] Daniel Jackson. Alloy: A Language and Tool for Relational Models. http://alloy.mit.edu/alloy/index.html.

[29] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the 39th ACM Conference on Computer and Communications Security*, 2012.

[30] Daniel Fett, Ralf Küsters, and Guido Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[31] Michael Dietz and Dan S. Wallach. Hardening persona – improving federated web login. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.

[32] Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Protecting Web Single Sign-on against Relying Party Impersonation Attacks through a Bi-directional Secure Channel with Authentication. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrustions and Defenses*, 2014.

[33] Microsoft. Microsoft. Live SDK developer guide - Signing users in. http://msdn.microsoft.com/en-us/library/live/hh243641#signin/.

[34] John Bradley. Posts about Federated Identity and Security Issues. http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html.

[35] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A Solver for Reachability Modulo Theories. In *Proceedings of the 24th International Conference on Computer Aided Verification*, 2012.

[36] Microsoft Research. Boogie: An Intermediate Verification Language. http://research.microsoft.com/en-us/projects/boogie/.

[37] Lawrence, Eric. Fiddler - The Free Web Debugging Proxy by Telerik. http://www.telerik.com/fiddler.

[38] Yuchen Zhou and David Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[39] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[40] Cristian Cadar and Dawson Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International Conference on Model Checking Software*, 2005.

[41] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[42] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14^{th} Network and Distributed System Security Symposium*, 2007.

[43] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting Logic Vulnerabilities in E-Commerce Applications. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.

[44] G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. De Carlini. Testing Web applications. In *Journal of Software Maintenance*, 2002.

[45] Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *23^{rd} International Conference on Software Engineering*, 2001.

[46] Nadia Alshahwan and Mark Harman. Automated Web Application Testing Using Search Based Software Engineering. In *26^{th} IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[47] Whitehat Security. Your Web Application Security Company. https://www.whitehatsec.com/.

[48] Redspin Inc. Penetration Testing, Vulnerability Assessments and IT Security Audits. https://www.redspin.com/.

[49] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated Replay and Failure Detection for Web Applications. In *20^{th} IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[50] Sara Sprenkle, Emily Hill, and Lori Pollock. Learning Effective Oracle Comparator Combinations for Web Applications. In *International Conference on Quality Software*, 2007.

[51] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Third ACM Conference on Data and Application Security and Privacy*, 2013.

[52] Qing Xie and Atif M. Memon. Model-Based Testing of Community-Driven Open-Source GUI Applications. In *22nd IEEE International Conference on Software Maintenance*, 2006.

[53] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *11th International World Wide Web Conference*, 2002.

[54] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *12th International Conference on World Wide Web*, 2003.

[55] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[56] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *20th ACM Conference on Computer and Communications Security*, 2013.

[57] Selenium development team. Selenium: Web application testing system. https://selenium.org/.

[58] TestingBot. Selenium Testing in the Cloud - Run Your Cross Browser Tests in Our Online Selenium Grid. http://testingbot.com/.

[59] BugBuster. BugBuster is a Software-as-a-Service to Test Web Applications. http://bugbuster.com/.

[60] Peter Pirolli, Wai-Tat Fu, Robert Reeder, and Stuart K. Card. A User-tracing Architecture for Modeling Interaction with the World Wide Web. In *First Working Conference on Advanced Visual Interfaces*, 2002.

[61] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web Application Testing with User Session Data. In *25th International Conference on Software Engineering*, 2003.

[62] Giancarlo Pellegrino and Davide Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In *21st Network and Distributed System Security Symposium*, 2014.

[63] Luyi Xing, Yangyi Chen, Xiaofeng Wang, and Shuo Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[64] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Suny, Yang Liuz, and Jin Song Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[65] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[66] Yuchen Zhou and David Evans. Protecting Private Web Content From Embedded Scripts. In *16th European Symposium On Research In Computer Security*, 2011.

[67] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International Conference on World Wide Web*, 2007.

[68] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[69] Steven Crites, Francis Hsu, and Hao Chen. OMash: Enabling Secure Web Mashups via Object Abstractions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[70] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. ESCUDO: A Fine-Grained Protection Model for Web Browsers. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, 2010.

[71] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[72] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *2011 International Conference on Distributed Computing Systems*, 2011.

[73] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the 11st Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[74] Vincent Toubiana, Helen Nissenbaum, Arvind Narayanan, Solon Barocas, and Dan Boneh. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[75] Matthew Fredrikson and Benjamin Livshits. RePriv: Re-Envisioning In-Browser Privacy. In *IEEE Symposium on Security and Privacy*, 2011.

[76] Douglas Crockford. ADsafe: Making JavaScript Safe for Advertising. www.adsafe.org, 2007.

[77] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe Active Content in Sanitized Javascript. google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf, 2007.

[78] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-Side Sandboxing of Third-party JavaScript without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[79] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[80] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *15th Nordic Conference in Secure IT Systems*, 2010.

[81] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[82] Ian Hickson. Web Workers in HTML5 Standard. http://www.whatwg.org/specs/web-workers/current-work/.

[83] Lon Ingram and Michael Walfish. TreeHouse: JavaScript sandboxes to help web developers help themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[84] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[85] Privly. Privly Browse Extension. https://priv.ly/.

[86] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[87] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[88] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[89] Aaron Blankstein and Michael J. Freedman. Automating Isolation and Least Privilege in Web Services. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[90] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[91] Mathias Lécuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios, Riley Spahn, Augustin Chaintreau, and Roxana Geambasu. Xray: Enhancing the web's transparency with differential correlation. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[92] Zhou Li, Kehuan Zhang, and XiaoFeng Wang. Mash-if: Practical Information-Flow Control within Client-Side Mashups. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[93] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[94] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[95] Karel Mittig. GreasySpoon, Scripting Factory for Core Network Services. http://greasyspoon.sourceforge.net/.

[96] The Chromium Development Group. The Chromium Projects: Notifications of Web Request and Navigation. https://sites.google.com/a/chromium.org/dev/developers/design-documents/extensions/proposed-changes/apis-under-development/notifications-of-web-request-and-navigation.

[97] R. Fielding, J.Gettys, J. Mogul, H.Frystyk, L.Masinter, P.Leach, and T.Berners-Lee. RFC2616: Hypertext Transfer Protocol - HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1.

[98] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986.

[99] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-party Computation Using Garbled Circuits. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[100] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[101] W3C. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR.

[102] ECMA International. ECMA JavaScript specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[103] Yuchen Zhou and David Evans. Understanding and monitoring embedded web scripts. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[104] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the Web for Fingerprinters. In *20th ACM Conference on Computer and Communications Security*, 2013.

[105] Ghostery, Inc. Ghostery. http://www.ghostery.com/.

[106] Abine, Inc. Protect your privacy with DoNotTrackMe from Abine. https://www.abine.com/index.html.

[107] Adrienne Porter Felt. See What Your Apps and Extensions Have Been Up To. http://blog.chromium.org/2014/06/see-what-your-apps-extensions-have-been.html.

[108] InformAction. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience. http://noscript.net/.