Understanding and Monitoring Embedded Web Scripts

Yuchen Zhou David Evans
University of Virginia
ScriptInspector.org

***Abstract*—Modern web applications make frequent use of third-party scripts, often in ways that allow scripts loaded from external servers to make unrestricted changes to the embedding page and access critical resources including private user information. This paper introduces tools to assist site administrators in understanding, monitoring, and restricting the behavior of third-party scripts embedded in their site. We developed ScriptInspector, a modified browser that can intercept, record, and check third-party script accesses to critical resources against security policies, along with a Visualizer tool that allows users to conveniently view recorded script behaviors and candidate policies and a PolicyGenerator tool that aids script providers and site administrators in writing policies. Site administrators can manually refine these policies with minimal effort to produce policies that effectively and robustly limit the behavior of embedded scripts. PolicyGenerator is able to generate effective policies for all scripts embedded on 72 out of the 100 test sites with minor human assistance. In this paper, we present the designs of our tools, report on what we've learned about script behaviors using them, evaluate the value of our approach for website administrator.**

1 Introduction

Modern web applications combine code from multiple sources in ways that pose important security and privacy challenges. Running as the same principal as the host, third-party scripts enjoy full access to host resources including sensitive user information and can make arbitrary modifications to the page. Some amount of access is necessary to provide the desired functionality — advertising scripts need to insert ads into the page, and analytics scripts need to read cookies and track user's behavior on the page. However, opening up access to all resources to allow such limited behavior is potentially dangerous. For example, an attacker who compromises hosts serving the Google Analytics script would be able to completely control more than 50% of the top websites [23, 30].

Several prior works have demonstrated the threat malicious embedded scripts pose to user security and privacy [6, 17]. Sites hosting scripts can be compromised, enabling attackers to deploy malicious scripts on unsuspecting websites [13]; scripts from respectable advertising networks may sub-contract space to increasingly less respectable networks, eventually leading to malicious scripts being included in prominent websites such as nytimes.com and spotify.com [31]. Responsible site administrators need a way to *understand* and *limit* the behavior of embedded scripts, especially those coming from servers outside their control. Without this, there is no way a site can stand behind its privacy policy short of eliminating sensitive content from pages that embed third-party scripts or disavowing responsibility for anything those scripts do.

However, JavaScript's dynamic nature makes it very hard to reason about the behavior of embedded scripts. Prevalent use of obfuscation and compression makes the code hard to analyze statically. Further, window.eval, document.write, and script element injections may introduce new executable code on-the-fly so that dynamic analysis techniques like symbolic execution will not work well either, especially when JSONP/AJAX requests are used to fetch additional code. JavaScript symbolic analysis tools (e.g. [5, 26]) often ignore calls to eval because constraint solvers cannot efficiently and soundly solve dynamically generated code inside eval. Recent improvements in such solvers [32] still lack soundness and completeness and are unable to handle typical scripts.

Contributions overview. We present and evaluate the design of a tool chain, depicted in Figure 1, intended to help site administrators understand and monitor the behavior of embedded scripts. This involves capturing the behaviors of embedded scripts precisely enough to develop effective policies. Section 2 explains how we define policies and

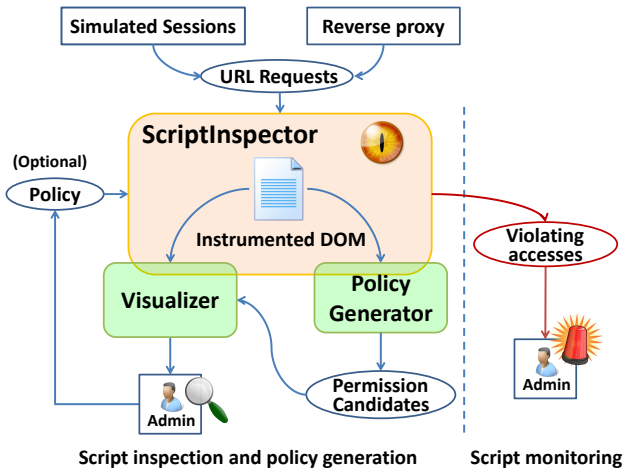

Figure 1: Overview

presents our supported policy permissions. A policy can be of two types — base or site-specific. A base policy for a script is a generic policy that should cover shared behavior of most embedding sites, while a site-specific policy is developed by site administrators to capture customized behaviors. The primary contributions of this paper are the design and evaluation of three closely integrated tools (highlighted as rounded rectangles in Figure 1). These tools are available under open source licenses at ScriptInspector.org.

ScriptInspector. ScriptInspector (Section 3) is 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. Given a website URL and one or more script policies, ScriptInspector records accesses that violate the policy. When no policies are given, all resource accesses by scripts are recorded in the instrumented DOM. ScriptInspector is able to attribute accesses to responsible scripts, even when the call stack includes more than one party and when scripts are injected into the DOM.

Visualizer. The Visualizer (Section 4) is 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. When given a set of permission candidates, the Visualizer can also be used to draw the matching nodes on the page to help site administrators develop effective policies (Section 7.2). Section 5 reports on our experiences using Visualizer to understand frequently-embedded scripts, and Section 7.2 describes our experiences using it to develop policies for popular websites.

PolicyGenerator. Since web pages embed many scripts with complex behaviors, it would be tedious and error-prone to attempt to develop access control policies for each script manually. We developed PolicyGenerator to help site administrators develop effective policies with limited human intervention. PolicyGenerator uses information recorded by ScriptInspector to infer candidate permissions. Site administrators can review the generated policies using Visualizer. Section 6 explains how PolicyGenerator works and evaluates the quality of the policies it generates for scripts embedded in popular websites. For 72 of the 100 tested sites, PolicyGenerator finds effective, high-coverage policies for all embedded scripts with minimal human effort. We evaluate the robustness of developed policies in Section 8.

Threat model. The goal of our work is to provide site administrators with a way to ensure the integrity of their site and protect the privacy of their users from embedded scripts. We are concerned with both malicious scripts provided by deceptive script providers and corrupted scripts resulting from compromises of external sites hosting embedded scripts. We focus on detecting sensitive resource leaks (such as user email addresses and shopping cart contents) and unintended page modifications (such as injecting advertisements in unintended places), but consider attacks that exploit system vulnerabilities such as drive-by-downloads and heap sprays out-of-scope.

By default, we assume that scripts from different domains are not colluding (or simultaneously compromised by the same attacker). Access control policies for multiple domains need to be merged and re-evaluated when a possible collusion scenario is suspected.

We are concerned with *large-scale* compromises of website users, not targeted attacks on high-value individuals. Our focus is on a defense that is robust and capable of handling complex scripts (so must be a dynamic analysis), and that can be performed by a site administrator without needing any control over clients (so assumes the site administrator sees similar script behaviors in test browsers as clients will locally). In particular, our techniques are not designed to address the case where an adversary hosting a script serves a different script to targeted users based on their IP addresses or designs a script that only behaves maliciously after detecting a particular browser fingerprint. We discuss the possibilities and challenges for enforcing policies at runtime on the client-side in Section 10.

2 Policies

A policy is just 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 2.1 and node descriptors in Section 2.2. Section 2.3 addresses the problem of interference between permissions.

2.1 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 2.2) 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 [2] 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.

2.2 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 7.1. 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 2.3).

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

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

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.

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 1 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 1, 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 1, as shown here:

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

2.3 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 7.1).

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 7.2).

3 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 2. Next, we explain how ScriptInspector records third-party script behavior. Section 3.2 discusses how the records are checked against policies or output to logs for admin's inspection.

3.1 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 2.1. 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 3.2.

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 2.1), 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.

3.2 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 4) or as input to PolicyGenerator to use in automatic policy generation (Section 6). 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 1).

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.

4 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 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 5. 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 7.1).

Figure 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 2.1), the entry is displayed in green and

enclosed by a dashed border. In Figure 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.

5 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 [2], 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 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 content. 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 2. In addition, multiple tracking scripts insert

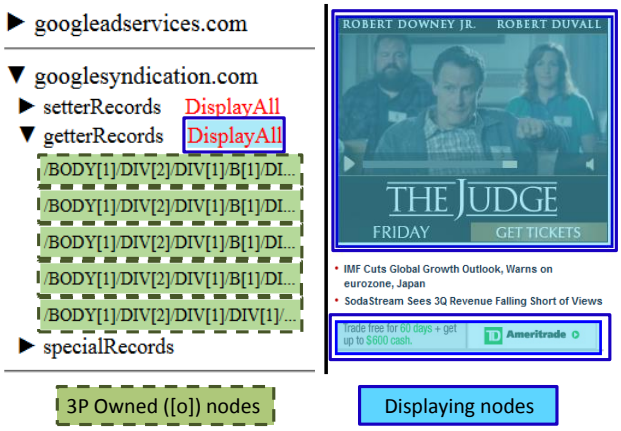

Figure 2: Visualizer interface

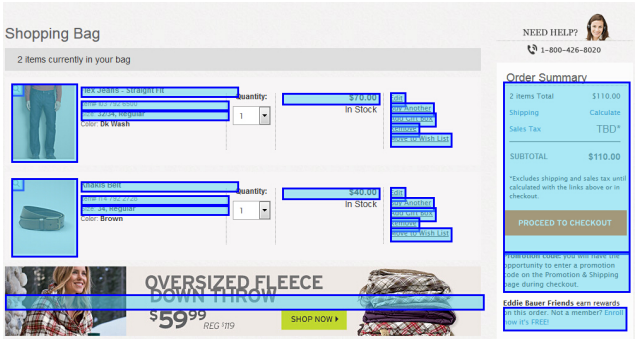

Figure 3: Script reading EddieBauer's shopping cart

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 trans-

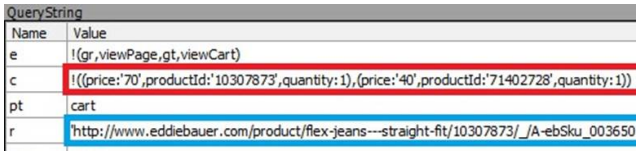

Figure 4: Script sending out EddieBauer's shopping cart information

mits user's shopping cart information (Figure 3). Figure 4 shows a captured form submission request in Fiddler [16] that 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 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.

6.1 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 5 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.[2]

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 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.

6.2 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.

[2]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>

Policy 2 Google Adsense Base Policy Excerpt
(permissions similar to those in Policy 1 are omitted)

//: GetAttribute:data−ad−container; //: GetId

//DIV[@id='div−gpt−ad−.∗']:!

/HTML[1]/BODY[1]:document.write:

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* 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 7).

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 7, 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.

7 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 7.1 introduces the PolicyGenerator tool and Section 7.2 reports on our experience using it.

7.1 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 selectorsdescribing 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 5, 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 1 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 2.3). 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 gen-

erator 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 4, 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.

7.2 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 8.

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 8.2).

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'].

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.

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
```

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
```

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.

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

8 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 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.

8.1 Policy size

Of the 100 sites tested, 72 sites needed at least one site-specific permission. Table I 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.

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 I: Scripts needing 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 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 5.

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). 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.

8.2 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 8.3. Section 8.4 considers several violation scenarios due to major updates in host sites or third-party scripts.

Figure 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 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.

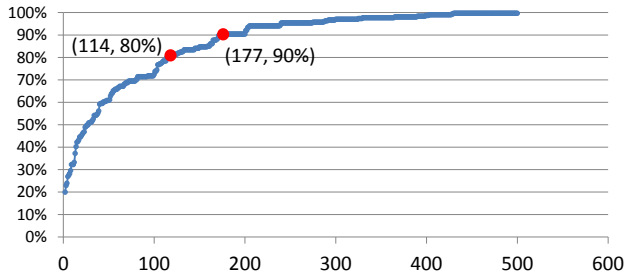

Figure 6: Training phase duration vs. Alarm rates

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.

8.3 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.

8.4 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

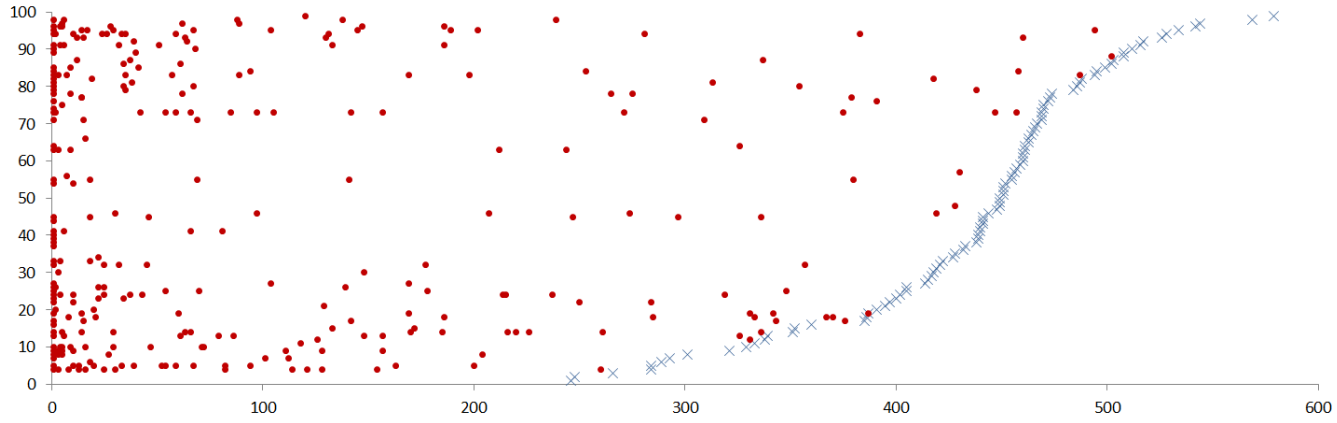

Figure 5: Policy convergence vs. number of revisions

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.

9 Related Work

The risks of embedded web scripts have been clear for some time, and many different solutions have been proposed. We highlight the most relevant previous work here.

Client-side script protections. Many solutions have been proposed that involve client-side protections that limit what embedded scripts can do and change how they interact with page resources. Browser-Enforced Embedded Policies (BEEP) [15] and Content Security Policy (CSP 1.0, 1.1) [29] use a whitelist approach to restrict the source of the dynamically loaded scripts. Compared to these works, our work supports fine-grained access control policies at the level of actions on individual DOM nodes.

MashupOS [27] proposed several new attributes and tags for isolation. It offers more flexible security policies than the same-origin policy (SOP). OMash [7] introduces private/public member functions for different scripts. ESCUDO [14] migrates the concept of OS protection rings to limit scripts' permissions, JCShadow [24] uses multiple execution contexts to isolate JavaScript execution. Treehouse [12] uses HTML5 web workers to isolate execution context and virtualizes host DOM via a hypervisor-like interface to enforce access control policy. JSand [4] isolates Secure ECMAScript (SES) execution by wrapping resource accesses using the new Harmony Proxy API. Adjail [19] places advertising scripts in a shadow page, and forwards the displaying content and user events back and forth to the main page. Our work has a different goal: we focus on understanding and monitoring of script behavior, rather than on isolating and enforcing security policies in the browser.

Script transformations. Several previous works provide mechanisms for incorporating policies into scripts. ConScript [21] uses aspect-oriented programming to weave generic policy checking advice with API calls of interest. Similar to ConScript, WebJail [3] also leverages aspect-oriented programming methods to enforce access control policy, but is designed specifically for mashup applications. Phung et al. [25] wrap JavaScript built-in functions with mediation code before executing third-party scripts. ADSafe [8] and Caja [22] mediate access by rewriting the third-party scripts. The appeal of these solutions is that no browser modifications are needed to enforce policies. Rewriting-based solutions, however, may fail to preserve the original program's semantics. It is also challenging to implement JavaScript rewriters in a way that cannot be circumvented [20]. We implemented ScriptInspector by modifying a browser instead of using rewriting because it provides higher confidence of complete mediation and makes it convenient to attribute dynamically introduced code. However, our policies are independent of our ScriptInspector implementation and could be enforced client-side using script rewriting.

Policy generation. Compared to access mediation and script isolation mechanisms, automated policy generation has not been deeply studied yet. ConScript [21] suggested auto-generating policies, but did not evaluate policy generation. Mash-IF [18] provides a GUI tool to let developers mark

sensitive information on the page, and uses information flow tracking techniques to restrict data leakage. Compared to Mash-IF, PolicyGenerator auto-suggests public elements instead of sensitive information, and shows that the inaccuracies of performing taint tracking on JavaScript can be avoided by using simple yet robust policies focused on resource access. Zhou et al. developed a method for automatically identifying nodes with sensitive content in the DOM, but found it could not accurately distinguish sensitive nodes for many sites [33]. This paper is the first to demonstrate that automated tools can help generate robust policies for popular third-party scripts.

Script behavior visualization. Several tools present script behaviors in a user-understandable way. Wang et al. [28] use a browser-based interface to explore relationships between requests and discover vulnerabilities. Popular browser extensions like Ghostery [10] and Abine [1] help users and site administrators understand what third-party services exist on the current page. A recent Chrome developer tool [9] 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 Visualizer and PolicyGenerator can be of great value to web developers in understanding scripts and developing policies.

10 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 [11] 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.

Availability

ScriptInspector, Visualizer, and PolicyGenerator, as well as all of the policies we developed, are available under an open source license from http://ScriptInspector.org.

Acknowledgements

This work was partially supported by grants from the National Science Foundation and Air Force Office of Scientific Research, and a gift from Google. The authors thank Ivan Alagenchev, Longze Chen, Haina Li, and Weilin Xu for valuable contributions to this work.

References

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

[2] 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.

[3] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[4] 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 *28th Annual Computer Security Applications Conference*, 2012.

[5] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vetting Browser Extensions for Security Vulnerabilities with VEX. In *19th USENIX Security Symposium*, 2010.

[6] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *19th International Conference on World Wide Web*, 2010.

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

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

[9] 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.

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

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

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

[13] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *17th ACM Conference on Computer and Communications Security*, 2010.

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

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

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

[17] Zhou Li, Sumayah Alrwais, XiaoFeng Wang, and Eihal Alowaisheq. Hunting the Red Fox Online: Understanding and Detection of Mass Redirect-Script Injections. In *35th IEEE Symposium on Security and Privacy*, 2014.

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

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

[20] 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.

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

[22] 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.

[23] 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 *19th ACM Conference on Computer and Communications Security*, 2012.

[24] 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.

[25] Phu H. Phung, Davis Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *4th International Symposium on Information, Computer, and Communications Security*, 2009.

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

[27] 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.

[28] 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 *33rd IEEE Symposium on Security and Privacy*, 2012.

[29] Wikipedia. Content Security Policy. http://en.wikipedia.org/wiki/Content_Security_Policy.

[30] Wikipedia. Google Analytics Popularity. http://en.wikipedia.org/wiki/Google_Analytics#Popularity.

[31] Lenny Zeltser. Malvertising: The Use of Malicious Ads to Install Malware. http://www.infosecisland.com/blogview/14371.

[32] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *9th Joint Meeting on Foundations of Software Engineering*, 2013.

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