Architectures for Inlining Security Monitors in Web Applications

Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld

Chalmers University of Technology, Gothenburg, Sweden

Abstract. Securing JavaScript in the browser is an open and challenging problem. Code from pervasive third-party JavaScript libraries exacerbates the problem because it is executed with the same privileges as the code that uses the libraries. An additional complication is that the different stakeholders have different interests in the security policies to be enforced in web applications. This paper focuses on securing JavaScript code by *inlining* security checks in the code before it is executed. We achieve great flexibility in the deployment options by considering security monitors implemented as security-enhanced JavaScript interpreters. We propose architectures for inlining security monitors for JavaScript: via browser extension, via web proxy, via suffix proxy (web service), and via integrator. Being parametric in the monitor itself, the architectures provide freedom in the choice of where the monitor is injected, allowing to serve the interests of the different stake holders: the users, code developers, code integrators, as well as the system and network administrators. We report on experiments that demonstrate successful deployment of a JavaScript information-flow monitor with the different architectures.

1 Introduction

JavaScript is at the heart of what defines the modern browsing experience on the web. JavaScript enables dynamic and interactive web pages. Glued together, JavaScript code from different sources provides a rich execution platform. Reliance on third-party code is pervasive [32], with the included code ranging from format validation snippets, to helper libraries such as jQuery, to helper services such as Google Analytics, and to fully-fledged services such as Google Maps and Yahoo! Maps.

Securing JavaScript Securing JavaScript in the browser is an open and challenging problem. Third-party code inclusion exacerbates the problem. The *same-origin policy (SOP)*, enforced by the modern browsers, allows free communication to the Internet origin of a given web page, while it places restrictions on communication to Internet domains outside the origin. However, once third-party code is included in a web page, it is executed with the same privileges as the code that uses the libraries. This gives rise to a number of attack possibilities that include location hijacking, behavioral tracking, leaking cookies, and sniffing browsing history [21].

Security policy stakeholders An additional complication is that the different stakeholders have different interests in the security policies to be enforced in web applications. *Users* might demand stronger guarantees than those offered by SOP when it is not desired that sensitive information leaves the browser. This makes sense in popular web applications such as password-strength checkers and loan calculators. *Code developers* clearly have an interest in protecting the secrets associated with the web application. For example, they might allow access to the first-party cookie for code from third-party services, like Google (as needed for the proper functioning of such services as Google Analytics), but under the condition that no sensitive part of the cookie is leaked to the third party. *Code integrators* might have different levels of trust to the different integrated components, perhaps depending on the origin. It makes sense to invoke different protection mechanisms for different code that is integrated into the web application. For example, an e-commerce web site might include jQuery from a trusted web site without protection, while it might load advertisement scripts with protection turned on. Finally, *system and network administrators* also have a stake in the security goals. It is often desirable to configure the system and/or network so that certain users are protected to a larger extent or communication to certain web sites is restricted to a larger extent. For example, some Internet Service Providers, like Comcast, inject JavaScript into the users' web traffic but so far only to display browser notifications for sensitive alerts[1].

Secure inlining for JavaScript This paper proposes a novel approach to securing JavaScript in web applications in the presence of different stakeholders. We focus on securing JavaScript code by *inlining* security checks in the code before it is executed. A key feature of our approach is focusing on security monitors implemented, in JavaScript, as security-enhanced JavaScript interpreters. This, seemingly bold, approach achieves two-fold flexibility. First, having complete information about a given execution, security-enhanced JavaScript interpreters are able to enforce such fine-grained security policies as *information-flow security* [37]. Second, because the monitor/interpreter is itself written in JavaScript, we achieve great flexibility in the deployment options.

Architectures for inlining security monitors As our main contribution, we propose architectures for inlining security monitors for JavaScript: via browser extension, via web proxy, via suffix proxy (web service), and via integrator. While the code extension and proxy techniques themselves are well known, their application to security monitor deployment is novel. Being parametric in the monitor itself, the architectures provide freedom in the choice of where the monitor is injected, allowing to serve the interests of the different stake holders: users, code developers, code integrators, as well as system and network administrators.

We note that our approach is general: it applies to arbitrary security monitors, implemented as JavaScript interpreters. The Narcissus [13] project provides a baseline JavaScript interpreter written in JavaScript, an excellent starting point for supporting versatile security policies.

Our evaluation of the architectures explores the relative security considerations. When introducing reference monitoring, Anderson [3] identifies the following principles: (i) the monitor must be tamperproof *(monitor integrity)*, (ii) the monitor must be always invoked *(complete mediation)* [39], and (iii) the monitor must be small enough to be subject to correctness analysis *(small trusted computing base (TCB))* [39, 35]. Overall, the requirements often considered in the context of monitoring are that the monitor must enforce the desired security policy *(soundness)* and that the monitor is transparent to the applications *(transparency)*. Soundness is of higher priority than

[1] `https://gist.github.com/ryankearney/4146814`

transparency in our setting. Our methods of deployment do not rely on transparency to provide security guarantees. Even if the application is able to detect that it is running in the monitor, this knowledge cannot be used to circumvent the monitor given that the monitor is sound (if this was possible the monitor is by definition not sound). Of course, if the application is able to detect that it is monitored it might chose to only expose benign behavior in order to escape detection. In either case, the user is protected from attacks. Note the relation of the soundness to Anderson's principles: while the principles do not automatically imply soundness, they facilitate establishing soundness. Transparency requirements are often in place for reference monitors to ensure that no new behaviors are added by monitors for any programs, and no behaviors are removed by monitors when the original program is secure.

Since the architectures are parametric in the actual monitor, we can draw on the properties of the monitor to guarantee the above requirements. It is essential for soundness and transparency that the monitor itself supports them. In our consideration of soundness for security, we assume the underlying monitors are sound (as natural to expect of such monitors). This implies that dealing with such features as dynamic code evaluation in JavaScript is already covered by the monitors. We note that monitor integrity, and complete mediation are particularly important in our security considerations because they are crucially dependent on the choice of the architecture. Our security considerations for the architectures are of general nature because of the generality of the security policies we allow.

Roadmap We study the relative pros and cons of the architectures. The goal of the study is not to identify a one-fits-all solution but to highlight the benefits and drawbacks for the different stakeholders. With this goal, we arrive at a roadmap to be used by the stakeholders when deciding on what architecture to deploy.

Instantiation To illustrate the usefulness of the approach, we present an instantiation of the architectures to enforce secure information flow in JavaScript. Information-flow control for JavaScript allows tracking fine-grained security policies for web applications. Typically, information sources and sinks are given sensitivity labels, for example, corresponding to the different Internet origins. Information-flow control prevents *explicit flows*, via direct leaks by assignment commands, as well as *implicit flows* via the control flow in the program.

Our focus on information flow is justified by the nature of the JavaScript attacks from the empirical studies [21, 32] that demonstrate the current security practices fail to prevent such attacks as location hijacking, behavioral tracking, leaking cookies, and sniffing browsing history. Jang et al. [21] report on both explicit and implicit flows exploited in the empirical studies. Further, inlining by security-enhanced interpreting is a particularly suitable choice for tracking information flow in JavaScript, because alternative approaches to inlining suffer from scalability problems, as discussed in Section 5.

Our instantiation shows how to deploy *JSFlow* [19, 18], an information-flow monitor for JavaScript by Hedin et al., via browser extension, via web proxy, and via suffix proxy (web service). We report on security and performance experiments that illustrate successful deployment of a JavaScript information-flow monitor with the different architectures.

2 Architectures

This section presents the architectures for inlining security monitors. We describe four different architectures and report on security considerations, pros and cons, including how the architectures reflect the demands of the different stakeholders. In the following we contrast the needs of the private user and the corporate user; the latter representing the network and system administrators as well.

2.1 Browser extension

Modern browsers allow for the functionality of the browser to be enriched via *extensions*. By deploying the security monitor via a browser extension it is possible to enforce properties not normally enforced by browsers. A browser extension is a program that is installed into the browser in order to change or enrich the functionality. By employing a method pioneered by Zaphod [31] it is possible to use the monitor as JavaScript engine. The basic idea is to turn off JavaScript and have the extension traverse the page once loaded using the monitor to execute the scripts. This method leverages that the implementation language for extensions and the monitor is JavaScript.

It is illuminating to contrast deployment via browser extension with directly instrumenting the browser (e.g., [40, 22, 17]). While the latter may provide performance benefits it is also monitor specific. Each monitor leads to a different instrumented browser, which is a significant undertaking. In this sense, browser instrumentation is comparable to implementation of a new monitor. The browser extension, on the other hand, is parametric over the monitor allowing different monitors to be used with the same extension. While potentially slower than an instrumented browser it also offers greater flexibility.

Security considerations From a security perspective, one of the main benefits of this deployment method is strong security guarantees. Since the JavaScript engine is turned off, no code is executed unless explicitly done by the extension. During execution the scripts are passed as data to the monitor, and are only able to influence the execution environment implemented by the monitor and not the general execution environment. This ensures the integrity of the monitor and complete mediation. In addition, this also guarantees that the deployment method is sound given that the monitor is sound.

However, by running the monitor as an extension, the monitor is run with the same privileges as the browser. Compared to the other methods of deployment this means that a faulty monitor not only jeopardizes the property enforced by the monitor, but might jeopardize the integrity of the entire browser.

Pros and cons Regardless of whether the user is private or corporate, browser extensions provide a simple install-once deployment method. From the corporate perspective, central management of the extension and its policies can easily be incorporated into standard system maintenance procedures. Important for the private user, the fact that the extension is installed locally in the browser of the user makes it possible to give the user direct control over what security policies to enforce on the browsed pages without relying on and trusting other parties.

A general limitation of this approach is that browser extensions are browser specific. This is less of an issue for corporate users than for private users. In the former case it is common that browser restrictions are already in place, and corporations have the assets to make sure that extensions are available for the used platform. In the latter case, a private user may be discouraged by restrictions imposed by the extension.

2.2 Web proxy

Deployment via browser extension entails being browser dependent and running the security monitor with elevated privileges. The web proxy approach addresses these concerns by including to monitor in the page, modifying any scripts on the page to ensure they are run by the monitor. All modern browsers support relaying all requests through a proxy. A proxy specific to relaying HTTP requests is referred to as a web proxy. The web proxy acts as a man-in-the-middle, making requests on behalf of the client. In the process, the proxy can modify both the request and the response, making it a convenient way to rewrite the response to include the monitor in each page. Doing so makes the method more intrusive to the HTML content, but less intrusive to the browser.

Security considerations For the monitor to guarantee security, all scripts bundled with the page must be executed by the monitor. The scripts can either be inline, i.e., included as part of the HTML page, or external, i.e., referenced in the HTML page to be downloaded from an external source. Inline scripts appear both in the form of script-tags as well as inline event handlers, e.g., onclick or onload. Apart from including the monitor in all browsed pages, all scripts, whether inline or external, must be rewritten by the web proxy to be executed by the monitor.

External scripts are rewritten in their entirety, whereas inline scripts must be identified within the page and rewritten them individually. As opposed to a browser extension that replaces the JavaScript engine, the monitor is executed by the engine of the browser in the context of the page. This is the same context in which all scripts bundled with the page are normally executed.

Unlike deployment via extension, omissions in this process breaks complete mediation, which risks undermining the integrity of the monitor; any script not subjected to the rewriting process is run in the same execution environment as the monitor. This assumes that there are no exploitable differences between the HTML parser used for rewriting and the parser of the browser. While this might be an issue in current browsers, with the introduction of standardized parsing in HTML5 we believe that this is a transient problem.

Under the assumption that all scripts are rewritten appropriately complete mediation is achieved. Complete mediation is required for both integrity and soundness, while the two latter are strongly related. Soundness is guaranteed by the soundness of the underlying monitor and complete mediation, given that the integrity of the monitor is guaranteed. This must, however, be the case, since the soundness of the monitor guarantees that no scripts executed by the monitor are able to jeopardize the integrity of the monitor. Thus, threats against the integrity of the monitor must come from scripts not run by the monitor, contradicting the assumption of complete mediation.

Unlike deployment via extension, special consideration is required for HTTPS connections, as HTTPS is designed to prevent the connection from being eavesdropped or modified in transit. To solve this the web proxy must establish two separate HTTPS connections, one with the client and one with the target. The client's request is passed on to the connection with the target and the rewritten response to the client. This puts considerable trust in the proxy, since the proxy has accesses to all information going to and from the user, including potentially sensitive or secret data. In addition, access to the unencrypted data significantly simplifies tampering unless additional measures are deployed. Whether including the proxy into the trusted computing base is acceptable or not depends on the situation.

Pros and cons In the corporate setting deployment via web proxy is appealing; it is common to use corporate proxies for filtering, which means that the infrastructure is already in place and trusted. Additionally, the use of proxies allows for easy central administration of security policies.

For the private user, however, the situation is different. Even though important considerations of extensions are addressed, e.g., browser dependency, and monitor privilege increase, adding the proxy to the trusted computing base might be a significant issue. Unless the private user runs and administers the proxy himself he might have little reason to trust the proxy with the ability to access all communicated information. This is especially true when the user visits web sites that he trusts more than the proxy. In such cases it could make sense to turn off the proxy, which, while possible, requires reconfiguring the browser.

2.3 Suffix proxy (service)

The extension and the web proxy deployment methods unconditionally applies the monitor to all visited pages. Suffix proxies can be used to provide selective monitoring, i.e., where the user can select when to use the monitor. Suffix proxies can be thought of as a service that allows the user to select which pages to proxy on demand — only pages visited using the suffix proxy will be subjected to proxying.

A *suffix proxy* is a specialized web proxy, with a different approach to relaying the request. The suffix proxy takes advantage of the *domain name system (DNS)* to redirect the request to it. Wildcard domain names allow all requests to any subdomain of the domain name to resolve to a single domain name, i.e., in DNS terms **.proxy.domain* $\Rightarrow$ *proxy.domain*.

Typically, the user navigates to a web application associated with the proxy and enters the target URL, e.g., `http://google.com/search?q=sunrise`, in an input field. To redirect the request to the proxy, the target domain name is altered by appending the domain name of the proxy, making the target domain a subdomain of the proxy domain, e.g., `http://google.com.proxy.domain/search?q=sunrise`. The suffix proxy is set up so that all requests to any subdomain are directed to the proxy domain. A web application on the proxy domain is set up to listen for such subdomain requests. When a request for a subdomain is registered, it is intercepted by the web application. The web application strips the proxy domain from the URL, leaving the original target URL, and makes the request on behalf of the client. As with the web proxy, relaying the request to the target URL gives the suffix proxy an opportunity to modify and include the monitor in the response.

Security considerations In the suffix proxy, not only the content is rewritten but also the headers of the incoming request and the returned response. Certain headers, like the *Host* and *Referrer* headers of the request, include the modified domain name and need to be rewritten to make the proxy transparent. Similarly, in the response, some headers, for instance *Location*, contain the unmodified target URL and need to be rewritten to include the monitor domain.

As the web proxy, the suffix proxy must ensure that all scripts bundled with a page are executed by the monitor. The procedure to rewrite scripts is much the same as for the web proxy. However, in order to guarantee complete mediation, the suffix proxy must also rewrite the URLs to external scripts to include the proxy domain, in addition to rewriting inline scripts in a page. Otherwise the script will not be requested through the monitor which will prevent it from being rewritten and thereby it will execute alongside the monitor.

Identical to the web proxy, soundness and integrity is guaranteed by complete mediation together with soundness of the monitor. See Section 2.2 above for a longer discussion.

A consequence of modifying the domain name is that the domain of the target URL no longer matches the modified URL, making them two separate origins as per the same-origin policy. This implies that all information in the browser specific to the target origin, e.g., cookies and local storage, are no longer associated with the modified origin, and vice versa. This results in a clean separation between the proxied and unproxied content.

Altering the domain name has another interesting effect on the same-origin policy. Modern web browsers allow relaxing of the same-origin policy for subdomains. Documents from different subdomains of the same domain can relax their domains by setting the `document.domain` attribute to their common domain. In doing so, they set aside the restrictions of the same-origin policy and can freely access each others resources across subdomains. This means that two pages of separate origins loaded via the proxy, each relaxing their domain attribute to the domain of the proxy, can access each others resources across domains. This is problematic for monitors that rely on the same-origin policy to enforce separation between origins. However, the flexibility of disabling the same-origin policy opens up for monitors aimed at replacing the same-origin policy with policies that are more appropriate for the given scenario. For example, given established trust between a number of sites it is possible for a monitor to disable the same-origin policy between these sites, while leaving it enabled, or even strengthened (via, e.g., information flow tracking), for any sites outside the set of mutually trusting sites. Note that a monitor is always able to refuse relaxation by preventing scripts from changing the domain attribute; what the suffix proxy enables is the ability for monitors to modify and even disable the same-origin policy by allowing JavaScript to relax the domain. Clearly, if this is done, care must be taken not to introduce any security breaches.

Similar to the web proxy, HTTPS requires special consideration. For the suffix proxy, however, the situation is slightly simpler. Given that the suffix proxy builds on DNS wildcards, it is sufficient to issue a certificate for all subdomains of the proxy domain, e.g., `*.proxy.domain`. Such a wildcard certificate is valid for all target URLs relayed through the proxy.

Pros and cons In the corporate setting the suffix proxy does not offer any advantages over a standard web proxy. Giving corporate users control over the decision whether or not to use the proxy service opens up for mistakes.

From the perspective of a private user suffix proxies can be very appealing. Given that the suffix proxy is hosted by a trusted party, e.g., the user's ISP, the proxy can provide additional security for any web page. At the same time the user retains simple control over which pages are proxied. At any point, the user can opt out from the proxy service by not using it.

On the technical side, while sharing a common foundation, there are several differences between a suffix proxy compared to a traditional web proxy. The differences lie, not in how the monitor is included in the page, but in the way the proxy is addressed. A consequence of the use of wildcard domain names is that the suffix proxy requires somewhat more rewriting than the web proxy in order to capture all requests.

Additionally, the suffix proxy can ensure that only resources relevant to security are relayed via the proxy, whereas a traditional web proxy must cover all requests. This both reduces the load on the proxy service, as well as the overhead for the end user, thus benefiting both the user and the service provider. This is not possible in a web proxy, that must relay all requests, but a suffix proxy provides the means to do so.

2.4 Integrator

Modern web pages make extensive use of third-party code to add features and functionality to the page. The code is retrieved from external resources in the form of JavaScript libraries. The third-party code is considered to be part of the document and is executed in the same context as any other script included in the document. Executing the code in the context of the page gives the code full access to all the information of the page, including sensitive information such as form data and cookies. Granting such access requires that the code integrator must trust the library not to abuse this privilege. To a developer, an appealing alternative is to run untrusted code in the monitored context, while running trusted code outside of the monitor.

Integrator-driven monitor inclusion is suitable for web pages that make use of third-party code. The security of the information contained on the web page relies not only on the web page itself, but also on the security of all included libraries. To protect against malicious or compromised libraries, an integrator can execute part of, or all of the code in the monitor. Unlike the other deployment alternatives, that consider all code as untrusted, this approach requires a line to be drawn between trusted and untrusted code. The code executing outside of the monitor is trusted with full access to the page and the monitor state, and the untrusted code will be executed by the monitor, restricted from accessing either. This can be achieved by manually including the monitor in the page and loading the third-party code either through the suffix proxy, or from cached rewritten versions of the code. This approach allows for a well defined, site-specific policy specification. The monitor is set up and configured with policies best suiting the need of the site.

An important aspect of integrator-driven monitor inclusion is the interaction between trusted and untrusted code. The trusted code executing outside the monitor can interact with the code executed in the monitor. This way, the trusted code can share specific information with the library, that the library requires to execute. There are different means of introducing this information to the monitor. The most rudimentary solution is to evaluate expressions in the monitor, containing the information in a serialized form. The monitor can also provide an API for reading and writing variables, or calling functions in the monitor. This simplifies the process and makes it less error-prone. A more advanced solution is a set of shared variables that are bidirectionally reflected from one context to the other when their values are updated.

Security considerations One security consideration that arises is the implication of sharing information between the trusted and untrusted code. It might be appealing to simplify sharing of information between the two by reflecting a set of shared variables of one into the other. However, automatically reflecting information from one context to the other, will have severe security implications in terms of confidentiality as well as monitor integrity. If the execution of trusted code depends on a shared variable, the untrusted code can manipulate the value to control the execution. Thus, for security reasons, any sharing of information with the untrusted code must be done manually by selectively and carefully introducing the information in the monitored context.

It should be noted that since the trusted code is running along side the monitor, it can access and manipulate the state of the monitor and thereby the state of the untrusted code. It is impossible for the monitored code to protect against such manipulation. The integrator approach allows web developers to make use of untrusted code in trusted pages in a secure way. Thus, regardless of manually or automatically wrapping the untrusted code it is the responsibility of the integrator to ensure complete mediation. In the former case by making sure to wrap all untrusted code, and in the latter case by ensuring that eventual parser differences do not compromise mediation. Since the integrator approach provides complete mediation for untrusted code only, it is important that the trusted code does not break the integrity of the monitor. If this would occur soundness cannot be guaranteed, since the integrity breach could potentially allow for the untrusted code to break out of the monitor. However, if the trusted code does not break the integrity of the monitor, the integrator approach guarantees soundness and integrity with respect to the untrusted code in a similar manner to the proxy approach, see Section 2.2.

Pros and cons This developer-centric approach gives the integrator full control over the configuration of the monitor and the policies to enforce. From the perspective of a user this approach is not intrusive to the browser, requires no setup or configuration, and provides additional security for the user's sensitive information. However, it also limits the user's control over which policies are applied to user information.

A benefit of the integrator-driven approach over the proxies is potential performance gains. While the proxies for all code on the page to run monitored, the integrator-driven approach lets the integrator select what is monitored and what is not.

As previously stated, sharing information between trusted and untrusted code in a secure manner requires manual interaction. This implies that the developer must to some degree understand the inner workings of the monitor and the implications of interacting with the monitor.

2.5 Summary of architectures

We have discussed four architectures for deployment. The differences between the architectures decides which architecture is better suited for different stakeholders and situations. The first three architectures were targeted to end users and were distinguished by their appeal to corporate and private users, whereas the last architecture was targeted on code developers.

Deployment via extension offers potentially stronger security guarantees at the price of running the monitor with the privilege of the browser, while the web proxy and suffix proxy approached were more susceptible to mistakes in the rewriting procedure. In the extension failing to identify a script leads to the script not executing, while in the proxies unidentified scripts would execute alongside the monitor, potentially jeopardizing its integrity. However, deployment via proxies requires someone to run and administer the proxies. For the corporate user the corporation is a natural host for such services, while the private user might lack such a trusted 3rd party. See Table 1 for a summary of how the architectures best suit each stakeholder. The results of the table are not firm. Rather they are recommendations based on the properties of the deployment methods and the needs of the different stakeholders in general.

For the corporate user we argue that deployment via web proxy may be the most natural method: it allows for simple centralized administration and, since it is common to use corporate proxies, the infrastructure might already be in place. As a runner up, deployment via extension is a good alternative deployment method, while the suffix proxy is the least attractive solution from a corporate perspective. The latter allows the user to select when to use the service, which opens up for security issues in case the user forgets to use the service.

For the private user we argue that deployment via extension is the most appealing method: after an initial installation it allows for local administration without the need to run additional services or rely on trusted 3rd parties. An interesting alternative for the private user is to use the suffix proxy. For web sites the private user trusts less than the provider of the suffix proxy service, the suffix proxy allows for increased security on a per web site basis. The web proxy is the least attractive means of deployment for the private user. From the private user's perspective the web proxy offers essentially the same guarantees as the extension, while either encumbering her to run her own proxy or rely on a 3rd party.

Finally, for the developer using the monitor as a library provides the possibility to include untrusted code safely using the monitor, while allowing trusted code to run normally. This allows for security, while lowering the performance impact by only monitoring potentially malicious parts of the program.

	Browser extension	Web proxy	Suffix proxy	Integrator driven
Corporate user		✓		
Private user	✓		✓	
Developer				✓

Table 1: Suitability of architectures with respect to different stakeholders.

3 Implementation

This section details our implementations of the architectures from Section 2. The code is readily available and can be obtained from the authors upon request.

3.1 Browser extension

The browser extension is a Firefox extension based on Zaphod [31], a Firefox extension that allows for the use of experimental Narcissus [13] engine as JavaScript engine. When loaded, the extension turns off the standard JavaScript engine by disallowing JavaScript and listens for the `DOMContentLoaded` event. `DOMContentLoaded` is fired as soon as the DOM tree construction is finished. On this event the DOM tree is traversed twice. The first traversal checks every node for event handlers, e.g., onclick, and registers the monitor to handle them. The second traversal looks for JavaScript script nodes. Each found script node is pushed onto an execution list, which is then processed in order. For each script on the execution list, the source is downloaded and the monitor is used to execute the script; any dynamically added scripts are injected into the appropriate place on the execution list.

The downside of this way of implementing the extension is that the order in which scripts are executed is important. When web pages are loaded, the scripts of the pages are executed as they are encountered while parsing the web page. This means that the DOM tree of the page might not have been fully constructed when the scripts execute. Differences in the state of the DOM tree can be detected by scripts at execution time. Hence, to guarantee transparency the execution of scripts must occur at the same times in the DOM tree construction as they would have in the unmodified browser. This can be achieved using DOM `MutationEvent` [41] instead of the `DOMContentLoaded` event. The idea is to listen to any addition of script nodes to the DOM tree under the construction, and execute the script on addition. However, due to performance reasons the DOM `MutationEvents` are deprecated, and are being replaced with DOM `MutationObserver` [42]. It is unclear whether the `MutationObserver` can be used to provide transparency, since events are grouped together, i.e., the mutation observer will not necessarily get an event each time a script is added — to improve performance single events may bundle several modifications together.

However, the exact order of loading is not standardized and differs between browsers. This forces scripts to be independent of such differences. Thus, using the method of executing scripts on the `DOMContentLoaded` event is not necessarily a problem in practice.

Further, since extension run with the same privileges as the browser certain protection mechanism are in place to protect the browser from misbehaving extensions. Those restrictions may potentially clash with the selected monitor. One example of this is `document.write`. The effect of `document.write` is [23] to write a string into the current position of the document. For security reasons, extensions are prohibited from calling `document.write`. Intuitively, `document.write` writes into the character stream that is fed to the HTML parser, which can have drastic effects on the parsing of the page. In a monitor it is natural to implement `document.write` by at some point calling the `document.write` of the browser. The alternative is to fully implement `document.write`, which would entail taking the interaction between the content written by `document.write`, the already parsed parts of the page and the remaining page into account. The inability to provide full functionality of `document.write` does not jeopardize the security, as argued in the introduction. Rather, it may prevent certain pages to execute properly. The consensus in the community is that `document.write` has few valid use cases, all pertaining to the inclusion of various entities during page load (calling `document.write` on a fully loaded page overwrites the entire page). One arguably reasonable use of `document.write` to include style sheets that only work when JavaScript is enabled. Another common use of `document.write`, that is broadly considered bad style, is to include scripts *synchronously* onto the page. Both approaches work by executing `document.write` with very specific strings as parameters, e.g.,

```
document.write(
    '<script src="http://somesite.com/script.js"></' + 'script>')
```

In such cases it is a simple matter to identify the attempt at inclusion, and mimic the appropriate behavior.

The extension consists of 1200 lines of JavaScript and XUL code.

3.2 Web proxy

The web proxy is implemented as an HTTP-server. When the proxy receives a request it extracts the target URL and in turn requests the content from the target. Before the response is delivered to the client, the content is rewritten to ensure that all JavaScript is executed by the monitor.

In HTML, where JavaScript is embedded in the code, the web proxy must first identify the inline code in order to rewrite it. Identifying inline JavaScript in HTML files is a complex task. Simple search and replace is not satisfactory due to the browser's error tolerant parsing of HTML-code, meaning that the browser will make a best-effort attempt to make sense of malformed fragments of HTML. It would require the search algorithm to account for all parser quirks in regard to malformed HTML, a task which is at least as complex as actually parsing the document. Consider the example below:

```
<script>0</script/> HTML </script>
<script>0</script./> alert(JavaScript) </script>
<p>a<sCript/"=/ src=//t.co/abcde a= >b</p></script c<p>d
```

The first line will be interpreted as a script followed by the text *HTML*, the second line as a script that alerts the string *JavaScript*, and the third will display *ac* and *d* in two separate paragraphs and load a script from an external domain.

In the web proxy, Mozilla's JavaScript-based HTML-parser *dom.js* [16], is used to parse the page. HTML parsing is standardized in the HTML5 specification. The dom.js parser is HTML5 compliant and parses the HTML the same as any HTML5 compliant browser. In the case that the browser is not HTML5 compliant, or if there are implementation flaws, there may be ways to circumvent the rewriting in the proxy by exploiting such parsing mismatches. However, as browser vendors are implementing according to the specification to an increasing degree, these types of attacks are to be less likely.

After the page has been parsed, the DOM-tree can be traversed to properly localize all inline script code. All occurrences of JavaScript code are rewritten as outlined in the code snipped below, wrapped in a call to the monitor. Because all instances of the modified script code will reference the monitor, the monitor must be added as the first script to be executed.

Rewriting JavaScript requires converting the source code to a string that can be fed to the monitor. The method `JSON.stringify()` provides this functionality and will properly escape the string to ensure that it is semantically equivalent when interpreted by the monitor. The code string is then enclosed in a call to the monitors interpreter, as shown below:

```
code = 'Monitor.eval(' + JSON.stringify(code) + ')';
```

The implementation consists of 256 lines of JavaScript code.

3.3 Suffix proxy (service)

The web proxy serves as a foundation for the suffix proxy. The suffix proxy adds with an additional step of rewriting to deal with external resources. Since the suffix proxy is referenced by altering the domain name of the target, the proxy must ensure that relevant resources, e.g., scripts, associated with the target page are also retrieved through the proxy. Resources with relative URLs requires no processing, as they are relative to the proxy domain and will by definition be loaded through the monitor. However, the URLs of resources targeting external domains must be rewritten to include the proxy domain. Similarly, links to external pages must include the domain of the proxy for the monitor. The external references are identified in the same manner as inline JavaScript, by parsing the HTML to a DOM-tree and traversing the tree. When found, the URL is substituted using a regular expression.

Another difference to the web proxy relates to the use of non-standard ports. The web proxy will receive all requests regardless of the target port. The suffix proxy, on the other hand, only listens to the standard ports for HTTP and HTTPS, port 80 and 443 respectively. The port in a URL is specified in conjunction to, but not included in the domain. Hence any URLs specifying non-standard ports would attempt to connect to closed ports on the proxy server. A solution to this problem is to include the port as part of the modified domain name. To prevent clashing with the target domain or the proxy domain, the port number is included between the two, e.g., *http://target.domain.com.8080.proxy.domain/*. This does not clash with the target domain because the top domain of the target domain cannot be numeric. Neither does it clash with the proxy domain because it is still a subdomain of the proxy domain. The implementation consists of 276 lines of JavaScript code.

4 Instantiation

This section presents practical experiments made by instantiating the deployment architectures with the *JSFlow* [19, 18] information-flow monitor. JSFlow is a tool that extends the formalization of a dynamic information flow tracker [20] to the full JavaScript language and its APIs. We briefly describe the monitor and discuss security and performance experiments.

Since the deployment approaches are parametric in the choice of monitor, we limit our interest to properties that relate to the approaches rather than the monitors. In particular, for the performance experiments we measure the time from issuing the request until the response is fully received, since the performance after that point depends entirely on the monitor, and for the security experiments we focus on results that are more generally interesting and not depending on the specific choice of monitor.

Monitor JSFlow is a dynamic information-flow monitor that tags values with runtime security labels. The security labels default to the origin of the data, e.g., user input is tagged *user*, but the labels can be controlled by the use of custom data attributes in the HTML. The default security policy is a strict version of the same-origin policy, where implicit flows, and flows via, e.g., image source attributes, are taken into account. Whenever a potential security violation has been encountered the monitor stops the execution with a security error. Implemented in JavaScript, the monitor supports full ECMA-262 (v5) [12] including the standard API and large parts of the browser-specific APIs such as the DOM APIs. JSFlow supports a wide variety of information-flow policies, including tracking of user input and preventing it from leaving the browser, as used in the security experiments below.

Security experiments Our experiments focus on password-strength checkers. After the user inputs a password, the strength of the password is computed according to some metric, and the result is displayed to the user, typically on a scale from *weak* to *strong*.

This type of service is ubiquitous on the web, with service providers ranging from private web sites to web sites of national telecommunication authorities.

Clearly, the password needs to stay confidential; The strength of the password is irrelevant if the password is leaked. We have investigated a number of password-strength services. Our experiments identify services that enforce two types of policies: (i) allow the password to be sent back to the origin web site, but not to any other site (suitable for server-side checkers); and (ii) disallow the password to leave the browser (suitable for client-side checkers). The first type places trust on the service provider not to abuse the password, while the second type does not require such trust, in line with the *principle of least privilege* [39]. Note that these policies are indistinguishable from SOP's point of view because it is not powerful enough to express the second type.

One seemingly reasonable way to enforce the second type of policy is to isolate the service, i.e., prevent it from performing any communication. While effective, such a stern approach risks breaking the functionality of the service. It is common that pages employ usage statistics tracking such as Google Analytics. Google Analytics requires that usage information is allowed to be gathered and sent to Google for aggregation. Using information-flow tracking, we can allow communication to Google Analytics but with the guarantee that the password will not be leaked to it.

We have investigated a selection of sites[2] that fall into the first category, and a selection of sites[3] that fall into the second category. Of these, it is worth commenting on two sites, one from each category. Interestingly, the first site, `https://testalosenord.pts.se/`, is provided by the Swedish Post and Telecom Authority. The site contains a count of how many passwords have been submitted to the service, with over 1,000,000 tried passwords so far. We are in contact with the authority to help improve the security and usability of the service. The second, `http://www.getsecurepassword.com/CheckPassword.aspx`, is an example of a web site that uses Google Analytics. The monitor rightfully allows communication to Google while ensuring the password cannot be leaked anywhere outside the browser.

The benefit of the architectures for the scenario of password-strength checking is that users can get strong security guarantees either by installing an extension, using a web proxy, or a suffix proxy. In the latter two cases, the system and network administrators have a stake in deciding what policies to enforce. Further, the integrator architecture is an excellent fit for including a third-party password-strength checker into web pages of a service, say a social web site, with no information leaked to the third party.

Performance experiments Since the approaches are parametric in the choice of monitor, we are interested in evaluating the performance of each approach rather than the performance of the employed monitor. We measure the time from issuing the request until the response is fully received, as the performance after that point depends entirely on the monitor. We measure the average overhead introduced by architectures compared to a reference sample of unmodified requests. The overhead is measured against two of the

[2] `https://testalosenord.pts.se/`, `http://www.lbw-soft.de/`, `http://www.inutile.ens.fr/estatis/password-security-checker/`, `https://passfault.appspot.com/password_strength.html`, and `http://geodsoft.com/cgi-bin/pwcheck.pl`.

[3] `http://www.getsecurepassword.com/CheckPassword.aspx`, `http://www.passwordmeter.com/`, `http://howsecureismypassword.net`, `https://www.microsoft.com/en-gb/security/pc-security/password-checker.aspx`, `https://www.grc.com/haystack.htm`, and `https://www.my1login.com/password-strength-meter.php`.

password-strength checkers listed previously, namely `passwordmeter.com`, over HTTP, and `testalosenord.pts.se`, over HTTPS. This measures the additional overhead introduced by the deployment method. Three out of the four architectures are evaluated; the browser extension, the web proxy, and the suffix proxy. The overhead of the integrator architecture is specific to the page that implements it, and is therefore not comparable to the other three. The browser extension does not begin executing until the browser has received the page and has begun parsing it, therefore its response time is the same as with the extension disabled. Due to the rewriting mechanism being closely related, the web proxy and the suffix proxy show similar results.

	Measurements (ms)	Average (ms)	Delta (ms)	Overhead (%)
Reference	434, 420, 445, 443	435	0	0%
Browser extension	434, 420, 445, 443	435	0	0%
Web proxy	638, 690, 681, 781	697	+262	+60.2%
Suffix proxy	663, 775, 689, 694	705	+270	+62.0%

Table 2: Architecture overhead passwordmeter.com

Proxy	Measurements (ms)	Average (ms)	Delta (ms)	Overhead (%)
Reference	114, 240, 103, 104	140	0	0%
Browser extension	114, 240, 103, 104	140	0	0%
Web proxy	372, 308, 311, 305	324	+184	+131.4%
Suffix proxy	316, 314, 324, 333	321	+181	+129.2%

Table 3: Architecture overhead testalosenord.pts.se

The tests were performed in the Firefox browser on the Windows 7 64 bit SP1 operating system on a machine with an Intel Core i7-3250M 2.9 GHz CPU, and 8 GB of memory.

5 Related work

We first discuss the original work on reference monitors and their inlining, then inlining for information flow, and, finally, inlining security checks in the context of JavaScript.

Inlined reference monitors Anderson [3] introduces reference monitors and outlines the basic principles, recounted in Section 1. Erlingsson and Schneider [15, 14] instigate the area of inlining reference monitors. This work studies both enforcement mechanisms and the policies that they are capable of enforcing, with the focus on safety properties. Inlined reference monitors have been proposed in a variety of languages and settings: from assembly code [15] to Java [10, 11, 9].

Ligatti et al. [25] present a framework for enforcing security policies by monitoring and modifying programs at runtime. They introduce *edit automata* that enable monitors to stop, suppress, and modify the behavior of programs.

Inlining for secure information flow Language-based information-flow security [37] features work on inlining for secure information flow. Secure information flow is not a safety property [29], but can be approximated by safety properties (e.g., [6, 38, 4]).

Chudnov and Naumann [7] have investigated an inlining approach to monitoring information flow in a simple imperative language. They inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [36]. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor.

Magazinius et al. [28] cope with dynamic code evaluation instructions by inlining on-the-fly. Dynamic code evaluation instructions are rewritten to make use of auxiliary functions that, when invoked at runtime, inject security checks into the available string. The inlined code manipulates shadow variables to keep track of the security labels of the program's variables. In similar vein, Bello and Bonelli [5] investigate on-the-fly inlining for a dynamic dependency analysis. However, there are fundamental limits in the scalability of the shadow-variable approach. The execution of a vast majority of the JavaScript operations (with the prime example being the + operation) is dependent on the types of their parameters. This might lead to coercions of the parameters that, in turn, may invoke such operations as `toString` and `valueOf`. In order to take any side effects of these methods into account, any operation that may case coercions must be wrapped. The end result of this is that the inlined code ends up emulating the interpreter, leaving no advantages to the shadow-variable approach.

Inlining for secure JavaScript Inlining has been explored for JavaScript, although focusing on simple properties or preventing against fixed classes of vulnerabilities. A prominent example in the context of the web is BrowserShield [34] by Reis et al. to instrument scripts with checks for known vulnerabilities.

Yu et al. [44] and Kikuchi et al. [24] present an instrumentation approach for JavaScript in the browser. Their framework allows instrumented code to encode edit automata-based policies.

Phung et al. [33] and Magazinius et al. [27] develop secure wrapping for self-protecting JavaScript. This approach is based on wrapping built-in JavaScript methods with secure wrappers that regulate access to the original built-ins.

Agten et al. [2] present JSand, a server-driven client-side sandboxing framework. The framework mediates attempts of untrusted code to access resources in the browser. In contrast to its predecessors such as ConScript [30], WebJail [1], and Contego [26], the sandboxing is done purely at JavaScript level, requiring no browser modification.

Despite the above progress on inlining security checks in JavaScript, achieving information-flow security for client-side JavaScript by inlining has been out of reach for the current methods [40, 8, 43, 21, 17] that either modify the browser or perform the analysis out-of-the-browser.

6 Conclusions

Different stakeholders have different interests in the security of web applications. We have presented architectures for inlining security monitors, to take into account the security goals of the users, system and network administrators, and service providers and integrators. We achieve great flexibility in the deployment options by considering security monitors implemented as security-enhanced JavaScript interpreters. The architectures allow deploying such a monitor in a browser extension, web proxy, or web service. We have reported on the security considerations and on the relative pros and cons for each architecture. We have applied the architectures to inline an information-flow security monitor for JavaScript. The security experiments show the flexibility in supporting the different policies on the sensitive information from the user. The performance experiments show reasonable overhead imposed by the architectures.

Future work is focused on three promising directions. First, JavaScript may occur outside script elements, e.g., as part of css, SVG or Flash. Ignoring JavaScript outside script elements potentially opens up for bypassing the security policies. One possible solution to this is to disallow JavaScript from occurring outside normal script tags, either by removing it of turning off the enabling features (e.g., Flash). Even though such a method might seem drastic it is conceivable due to the limited proliferation of JavaScript outside normal scripts. Nevertheless, to provide a more complete solution, we aim to investigate extending the approaches presented in this paper to handle such JavaScript. Second, recall that the integrator architecture relies on the developer to establish communication between the monitored and unmonitored code. With the goal to relieve the integrator from manual efforts, we develop a framework for secure communication that provides explicit support for integrating and monitored and unmonitored code. Third, we pursue instantiating the architectures with a monitor for controlling network communication bandwidth.

Acknowledgments Thanks are due to Christian Hammer for useful feedback. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish agencies SSF and VR.

References

1. S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen. Webjail: least-privilege integration of third-party components in web mashups. In *Proc. of ACSAC'11*, 2011.
2. P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In R. H. Zakon, editor, *ACSAC'12*, pages 1–10. ACM, 2012.
3. J. P. Anderson. Computer security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.
4. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
5. L. Bello and E. Bonelli. On-the-fly inlining of dynamic dependency monitors for secure information flow. In *Proc. of FAST'11*, pages 55–69, 2011.
6. G. Boudol. Secure information flow as a safety property. In *Proc. of FAST'08*.
7. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. of CSF'10*.
8. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavasCript. In M. Hind and A. Diwan, editors, *PLDI*, pages 50–62. ACM, 2009.
9. M. Dam, G. L. Guernic, and A. Lundblad. Treedroid: a tree automaton based approach to enforcing data processing policies. In *Proc. of ACM CCS'12*, pages 894–905, 2012.
10. M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded java. In *Proc. of ECOOP'09*, pages 546–569, 2009.
11. M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably correct inline monitoring for multithreaded java-like programs. *Journal of Computer Security*, 18(1):37–59, 2010.
12. ECMA International. ECMAScript Language Specification, 2009. Version 5.
13. B. Eich. Narcissus—JS implemented in JS. `http://mxr.mozilla.org/mozilla/source/js/narcissus/`, 2011.
14. U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004.
15. U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proc. of NSPW'99*, pages 87–95, 1999.
16. A. Gal. dom.js. `https://github.com/andreasgal/dom.js`.
17. W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proc. of ACM CCS'12*, Oct. 2012.
18. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow. Software release. Located at `http://chalmerslbs.bitbucket.org/jsflow`, Sept. 2013.

19. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*. ACM, Mar. 2014.
20. D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–18, June 2012.
21. D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proc. of ACM CCS'10*, Oct. 2010.
22. S. Just, A. Cleary, B. Shirley, and C. Hammer. Information Flow Analysis for JavaScript. In *Proc. of PLASTIC '11*.
23. J. Kesselman. Document Object Model (DOM) Level 2 Core Specification, 2000.
24. H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. Javascript instrumentation in practice. In *Proc. of APLAS'08*, pages 326–341, 2008.
25. J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.
26. T. Luo and W. Du. Contego: Capability-based access control for web browsers - (short paper). In *Proc. of TRUST'11*, pages 231–238, 2011.
27. J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *Proc. of NordSec'10*, pages 239–255, 2010.
28. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
29. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.
30. L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proc. of IEEE S&P'10*, 2010.
31. Mozilla Labs. Zaphod add-on for the Firefox browser. `http://mozillalabs.com/zaphod`, 2011.
32. N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *Proc. of ACM CCS'12*, pages 736–747, Oct. 2012.
33. P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proc. of ASIACCS'09*, pages 47–60, 2009.
34. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.
35. J. M. Rushby. Design and verification of secure systems. In *Proc. SOSP'81*, 1981.
36. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, pages 186–199, July 2010.
37. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
38. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. PSI'09*, LNCS. Springer-Verlag, June 2009.
39. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, Sept. 1975.
40. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. of NDSS'07*, Feb. 2007.
41. W3C. Document Object Model (DOM) Level 3 Events Specification. `http://www.w3.org/TR/DOM-Level-3-Events/`.
42. W3C. DOM4 W3C Working Draft 6. `http://www.w3.org/TR/dom/`.
43. A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys'09*, pages 233–246, New York, NY, USA, 2009. ACM.
44. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 237–249. ACM, 2007.