Third-party code inclusion is rampant, potentially exposing sensitive data to attackers. Protected Web components can keep private data safe from opportunistic attacks by hiding static data in the Document Object Model (DOM) and isolating sensitive interactive elements within a Web component.
techniques support code isolation,\textsuperscript{1,2} but don’t offer isolation of data in the Document Object Model (DOM).\textsuperscript{3} Finally, the recent Web Components specification lets developers instantiate custom HTML tags for use within the page.\textsuperscript{4} A major feature of such custom elements is the support for a hidden DOM, known as the Shadow DOM.\textsuperscript{5} Unfortunately, the Web components specification focuses on functional separation of the DOM and doesn’t offer security features or code isolation.

Here, we motivate the need for a flexible mechanism that supports the isolation of the user’s private data in the DOM, as well as the isolation of sensitive elements, such as input elements of a login form. Furthermore, we investigate the properties of the Web components specification, and show that there’s a potential for offering the desired level of isolation without compromising the much needed flexibility of modern Web applications.

**Use Cases and Existing Technologies**

Integrating third-party components using remote scripts is common on the Web. Examples include programming APIs and development frameworks (such as jQuery and Bootstrap), advertising services (such as DoubleClick and AdSense), Web analytics tools (such as Google Analytics), and social media plug-ins (such as Facebook’s “like” button). A 2012 study of remote JavaScript inclusions on the Alexa top 10,000 sites showed that 88.45 percent include at least one remote script, and one site even included scripts from 295 remote hosts.\textsuperscript{6} Furthermore, 68.37 percent of sites included the Google Analytics library, and 79.74 percent included at least one Google library. Finally, the study applied a set of metrics to show that 12 percent of sites that were deemed security conscious included scripts from sites that deployed weak security measures.

Including remote scripts not only creates a vector for attacks targeting a specific Web application, but it also presents an attack vector for opportunistic attackers, who aim to execute low-profile attacks on a large number of Web applications. Such attacks can yield large quantities of sensitive information—for example, by scraping the webpage’s user-specific content, recording user-provided input in form fields, and extracting security tokens and session identifiers. Even when developers carefully select only trusted third parties for remote script inclusion, a certain risk persists, because third-party providers can be compromised as well. The dangers of third-party script inclusions are best illustrated by real-world examples, such as on-screen keyboard scraping malware\textsuperscript{7}, malware spread through advertisements,\textsuperscript{8} or actual compromises of third-party providers.\textsuperscript{9,10}

An opportunistic attacker can gain access to the Web application’s client-side context through several attack vectors—for example, by compromising a remotely included script or advertisement, or through a cross-site scripting attack (XSS). Because of the wide variety of sites that can be compromised through a malicious script or advertisement, opportunistic attackers carry out nontargeted attacks, such as looking for input elements of the type password, or scraping any user-specific displayed content, such as email messages, health records, and bank statements.

**Use Cases**

In light of the opportunistic attacker model, we propose three general use cases that benefit from effectively isolating data or HTML elements within the browser.

**Displaying sensitive information.** Many Web applications process and display user-specific information, which is often considered private and sensitive. Common examples of such private data are email messages, chat conversations, bank statements, and security challenges. Opportunistic attackers can easily inspect and collect such sensitive information because it isn’t isolated from the rest of the page, which includes third-party scripts.

An effective isolation mechanism for in-application content could prevent inspection or collection by an opportunistic attacker.
Protecting security tokens. A variant of displayed private information are application-related, hidden security tokens, often associated with a user’s session. For example, the security tokens protecting against cross-site request forgery (CSRF) attacks are embedded as hidden form elements.11

Hiding such security tokens from opportunistic attackers raises the security level of the applied countermeasures, thereby eliminating alternative attack vectors.

Protecting sensitive input elements. A third use case focuses on protecting client-side input elements, in contrast to hiding server-delivered content. Most Web applications contain sensitive input elements, such as HTML password elements and on-screen keyboards. Opportunistic attackers can easily gather sensitive user-provided data by using generally applicable selectors for sensitive input elements.

Isolating such sensitive input elements from opportunistic attackers ensures that user-provided input cannot easily be stolen with a nontargeted attack. Note that such an isolation mechanism must extend toward event handlers associated with isolated input elements.

Motivating Empirical Evidence
The inclusion of potentially untrusted third-party code into a Web application is a common though potentially dangerous practice.8 Two important industry-driven surveys of the most critical software errors warn of this risk. The Open Web Application Security Project (OWASP) Top Ten Project, which lists the 10 most dangerous risks for Web applications, gives “using components with known vulnerabilities” ninth place.12 A similar initiative, the CWE/SANS Top 25 Most Dangerous Software Errors, puts “inclusion of functionality from untrusted control sphere” at the 16th spot.13

To support the high rankings in these industry surveys, and to establish the relevance of the aforementioned use cases, we conducted two relatively small-scale experiments. To support the use cases for hiding sensitive data in the DOM, we investigate popular online password managers, where the DOM holds all of the user’s passwords to every website. The second experiment supports the use case for protecting sensitive input elements by measuring the exposure of login forms to third-party script providers.

Password managers. Online password managers are used to store the multitude of authentication credentials required on the modern Web. This private and highly sensitive data is often even stored in an encrypted container, which is decrypted at the client side when the client provides the correct master key. One might expect that in such a sophisticated setup, the decrypted data is handled with care, preventing any risk of stolen or leaked data.

For seven online password managers, gathered from the top 20 results for the Google query “free online password manager,” we investigated whether they include scripts from a third-party on the page that hosts the passwords in the DOM, giving these scripts full access to the user’s credentials. As Table 1 shows, six of the seven (86 percent) include third-party scripts from at least one remote host on the page that displays the user’s passwords. The Ghostery browser extension (https://www.ghostery.com/en/) considers all scripts to be analytics. Additionally, two password managers include scripts from additional remote hosts on their main page, which is situated within the same origin as the sensitive page.

Login forms. Almost every webpage has a login form, which are a trivial target from which an opportunistic attacker can extract user credentials. We crawled the Alexa top 1,000 sites, looking for login forms situated on a page with third-party script inclusions, thereby giving the third party full access to the login form.

<table>
<thead>
<tr>
<th>Search ranking</th>
<th>Name</th>
<th>No. of remote scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PassPack</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>LastPass</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Norton Identity Safe</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Keeper</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Dashlane</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Clipperz</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>Mitto</td>
<td>1</td>
</tr>
</tbody>
</table>
We found that 52 percent of the websites included a login form, and all of them included at least one third-party script in the login page. Of the sites with a login form, 40 percent included scripts from more than five different third-party hosts. Figure 1 shows the right-skewed distribution of login pages including scripts from remote hosts, with an average number of 3.4 hosts on a login page, and an extreme of one login page including code from 36 different remote hosts. These numbers indicate that a scenario with an opportunistic attacker targeting login forms is, unfortunately, very plausible.

**Existing Technologies**

Several technologies are relevant when discussing third-party script inclusion and content separation.

**Document isolation.** Web developers can use frames or iframes to isolate content in separate documents to varying degrees, depending on the associated origins. Placing data in a document with a different origin from the main document effectively offers both DOM-based and script-based isolation, and further restrictions are available through the HTML5 sandbox attribute.

Document-based isolation offers strong security guarantees but has a rigid, block-level structure, making it less attractive for modern Web applications. Additionally, frames with different origins require a separate roundtrip to fetch the content, causing a delay in page load times.

**JavaScript sandboxing.** Driven by the rise in remote script inclusions, script-based sandboxing techniques are being developed and deployed. By isolating a remote script in a sandbox, developers gain fine-grained control over its capabilities, thereby preventing the script from misbehaving.

Although sandboxing techniques can effectively be used to contain remote scripts, they typically don’t provide a way to isolate parts of the DOM, making it difficult to secure the described use cases.

**DOM separation.** The Web Components specification combines a set of technologies allowing the creation of custom HTML elements. One interesting technology is the shadow DOM, which allows custom elements to hide their internal DOM structure from the outside world.

One currently deployed example is the HTML5 video element, which features a control bar with play/pause buttons. The internals of the video element are implemented using traditional HTML elements but are hidden from the webpage and the user via the shadow DOM. The shadow DOM is well suited to hiding content in the DOM but doesn’t prevent later access, nor does it offer script-based isolation properties.

**Protected Web Components**

Web components are the most viable starting point for creating a protection mechanism for private data and sensitive elements against opportunistic attackers. They offer the required flexibility to cope with the highly dynamic requirements of modern Web applications, as opposed to iframes, and already possess the capability to host a separate DOM tree using the shadow DOM, a property that is hard to achieve using JavaScript sandboxing technologies.

To leverage Web components to create protected Web components, we must be able to hide static data in the DOM tree, without it being accessible to opportunistic attackers. Second, protected Web components should be able to host interactive elements, without being vulnerable to script-based compromises—for example, through function-overriding or prototype-poisoning.
attacks. In this section, we explain how shadow DOM trees can be permanently hidden by taking advantage of ECMAScript 5 getters, and elaborate on techniques that can be used to isolate script code within a hidden tree. Figure 2 illustrates the use of protected Web components in a password manager.

Hiding Static Data
The goal of the first and second use cases was to embed private, user-specific data into the DOM tree, without exposing it to an opportunistic attacker, who uses DOM manipulation techniques to extract potentially sensitive information. Such techniques include the use of JavaScript DOM APIs, stylesheet operations, and custom selectors.

The shadow DOM supports the creation of separate DOM trees, which are attached to traditional HTML elements using the `shadowRoot` property, and composed into a single DOM tree during the rendering process. The main document and any embedded shadow DOM trees are functionally separated, limiting the propagation of Cascading Style Sheets (CSS) or selectors between the main document and the subtrees, in both directions. Shadow DOM trees are already used to implement browser controls, such as the playback bar for the video element, and can also be used by a developer through a JavaScript API. Note that the browser’s internal shadow DOM trees are not accessible through the `shadowRoot` property, whereas developer-created shadow DOM trees remain accessible from JavaScript.

Unfortunately, the latter property of script-defined shadow DOM trees conflicts with the goal of hiding static data in the DOM. However, by redefining the getter of the `shadowRoot` property, developers can make their script-defined DOM trees inaccessible to JavaScript. Figure 3 shows the creation and population of a shadow DOM, and the overriding of the getter to return `null` instead of a reference to the shadow DOM.

After redefining the getter and wiping all existing references to the shadow DOM, it’s no longer possible to directly access the data stored in the shadow DOM. Therefore, instantiating an inaccessible shadow DOM tree with sensitive data before loading untrusted code ensures that the private data will never be exposed to opportunistic attackers.

Isolating Interactive Scripts
The third use case aims to protect sensitive input elements from untrusted scripts. Sensitive input elements are usually part of a form, and they typically depend on JavaScript handlers for interactive input processing and validation.

Although the shadow DOM is ideally suited to isolating elements from the rest of the page, a problem arises when these elements use JavaScript handlers for processing input events. The shadow DOM offers functional separation but doesn’t instantiate a separate JavaScript context, leaving the JavaScript code defined

---

**Figure 2.** Protected Web components for data security: (a) a password manager page containing private data and sensitive elements, together with a third-party advertisement, without any isolation or protection; (b) the effect of using protected Web components.

**Figure 3.** Data can be hidden in the shadow DOM by clearing existing references and redefining the only access point.

```
var protected = document.createElement('div');
var root = protected.createShadowRoot();
//Append data to the root
root = null;
Object.defineProperty(protected, "shadowRoot", 
{ get: function() { return null; }});
```
in the shadow DOM vulnerable to several attacks, such as function overriding and prototype poisoning.

To obtain protected Web components, the shadow DOM’s script code needs to be effectively isolated, not only to prevent JavaScript functions and variables from leaking into the global namespace, but also to prevent the use of potentially contaminated functions defined in the global namespace or Object prototypes. Obtaining this isolation in the current shadow DOM requires two separate steps. First, any code within the shadow DOM should be encapsulated in a separate namespace, which is possible in JavaScript through the correct use of closures. Second, the use of potentially contaminated functions can be prevented by storing and using known good versions of the required functions, a technique often used in JavaScript sandboxing and policy enforcement mechanisms.14,15 Figure 4 is a brief code snippet using closures and known good functions.

Isolating the shadow DOM’s JavaScript code, in combination with overriding the shadowRoot getter, effectively supports HTML elements containing sensitive data, while maintaining script-based interaction.

Motivating Examples Revisited

Protected Web components offer a strong mechanism to isolate data and sensitive elements within the DOM tree, without sacrificing the flexibility to place this data anywhere within the page, like iframes do. These properties ensure that protected Web components are well suited to meet the three use cases described earlier.

Displaying sensitive information. By embedding sensitive data in a secure Web component, using the shadow DOM to hide static data, we effectively prevent an opportunistic attacker from extracting the data in an automated way.

Protecting security tokens. Because security tokens are often embedded in interactive elements such as forms, they can be protected by placing the element inside a secure Web component. Security tokens, such as CSRF tokens, are part of the DOM, and the secure component will prevent an opportunistic attacker from extracting them.

Although protected Web components offer significant security benefits against a realistic, ubiquitous opportunistic attacker, they also have a limited impact. First, by embedding sensitive elements in a secure Web component, they are effectively separated from the rest of the page, preventing any interactions, even from legitimate code within the page. Therefore, all code interacting with a sensitive element must be loaded in the secure component. Typically, this code is closely tied to the element anyway, with validation handlers and autocompletion code as an example. Continuing on these handlers, we regret that the full implementation burden rests once again with the developer. Therefore, we envision the Web components specification endorsing two configurable extensions to the current model:

- hiding a shadow DOM, where the shadowRoot attribute doesn’t return a reference to the shadow DOM, similar to the current behavior of user-agent-created shadow DOMs, and

Protecting sensitive input elements. Sensitive input elements capture user input and can be a target for opportunistic attackers. These elements can be placed in a secure Web component as well, preventing direct querying by an attacker. If these input elements depend on script-based handlers for validation, autocompletion, and so on, the handler code must be part of the secure Web component as well.

The protected Web components not only fit the three proposed use cases but also protect against opportunistic attackers in the two examples presented earlier. First, the online password managers can use protected Web components to prevent deliberate or inadvertent extraction of the user’s credentials from the DOM, while preserving the possibility of including third-party scripts. In the second scenario, the login forms and associated handlers can be embedded in a protected Web component, preventing a curious or malicious script from stealing the user’s credentials through input events.

Figure 4. By using closures and known good copies of functions, scripts can be isolated within the shadow DOM.

```javascript
(function() {
  var getElement = document.getElementById;
  var data = getElement("shadowinput").textContent;
  //...
})();
```
instantiating a new script context within the shadow DOM, ensuring that all scripts imported by the shadow DOM are separate from the hosting page.

The latter extension is comparable to how Web workers also run in a separate context, enabling messaging through a predefined interface. The possibility of instantiating new script contexts in a shadow DOM also benefits the deployment of Web components, because it prevents naming and scoping conflicts between the different imported components and the host page. The downside of instantiating a new script context is the lack of shared global variables, requiring any libraries to be loaded in each context.

Hiding private content and sensitive elements through Web components can help mitigate opportunistic, nontargeted attacks. We consider this approach to be part of the recent trend in client-side security mechanisms, which significantly improve the security of client-side aspects of Web applications, often by applying the defense-in-depth principle. Previously adopted examples are the HttpOnly flag for cookies, which prevents several common session attacks; and the Content Security Policy, which significantly raises the bar for typical cross-site scripting attacks.

Acknowledgments

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT), the Research Fund KU Leuven, the IWT-SBO project SPION, and by the EU FP7 project STREWS. The Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE) also provided financial support.

References


Philippe De Ryck is a post-doctoral researcher in the Computer Science Department at the Katholieke Universiteit Leuven, Belgium. He has recently finished his PhD on Web application security, with a specific focus on cross-site request forgery (CSRF), session management, and JavaScript sandboxing techniques. He is the lead author of Primer on Client-side Web Security, which gives a
broad overview of the current state of client-side security in the Web. Contact him at philippe.deryck@cs.kuleuven.be.

Nick Nikiforakis is an assistant professor in the Computer Science Department at Stony Brook University. His research interests include Web application security and privacy, which he usually approaches by looking at the Web as a series of inter-connected ecosystems. Contact him at nick@cs.stonybrook.edu.

Lieven Desmet is a research manager of secure software within the iMinds-DistriNet research group at the Katholieke Universiteit Leuven, Belgium. His research interests include software security, and in particular, Web application security. Lieven received a PhD in computer science from the University of Leuven. He’s a board member of the Open Web Application Security Project’s Belgium chapter, and program director of the yearly SecAppDev training courses on secure application development. Contact him at lieven.desmet@cs.kuleuven.be.

Frank Piessens is a professor in the Department of Computer Science at the Katholieke Universiteit Leuven, Belgium. His research interests include software security, and in particular the development of high-assurance techniques to deal with implementation-level software vulnerabilities and bugs, including techniques such as software verification, runtime monitoring, type systems, and programming language design. Contact him at frank.piessens@cs.kuleuven.be.

Wouter Joosen is a professor in the Computer Science Department at the Katholieke Universiteit Leuven, Belgium. His research interests are aspect-oriented software development, middleware, and software security. Joosen received a PhD in computer science from KU Leuven. Contact him at wouter.joosen@cs.kuleuven.be.

Selected CS articles and columns are available for free at http://ComputingNow.computer.org.