Poster Proposal for *IEEE Security and Privacy*, May 2013

RedactDOM: Preventing Sensitive Data Leaking through Embedded Scripts

Longze Chen and Yuchen Zhou and David Evans
University of Virginia
[lc8dp, yz8ra, evans]@virginia.edu

***Abstract*—Third-party embedded scripts provide a powerful way to build rich web applications, but raise important security risks. These untrusted scripts can access everything on the embedding web pages including sensitive data, which can then be leaked to malicious third-party servers. We propose RedactDOM, an egress-based approach that prevents untrusted scripts from leaking sensitive information without disrupting the scripts' functionality. For each page, a projection page is created with a redacted DOM that has the structure and scripts from the original page but removes all potentially sensitive data. RedactDOM blocks outgoing requests generated by untrusted scripts on the real page and replaces them with a safe substitutes from the redacted page.**

I. Motivation

Modern websites favor third-party scripts for the variety of services they can provide. For example, embedded widgets can make the page more interactive while powerful JavaScript libraries can provide web developers more functionality with less effort. Embedded scripts run as the same principal as the host scripts: they can access any (sensitive) web content and exfiltrate it to an external server. Nikiforakis et al.'s recent large scale study on JavaScript inclusions found that almost 90% of the Alexa top 10K websites include remote inclusions and the total number of inclusions increased by 40% each year for the past three years [1]. Their results show that scripts provided by low-maintenance websites are risky to include. These remote sites have security vulnerabilities and each embedded script is a potential vector for attacking the containing site.

To prevent private information leakage from third-party scripts, one general solution is to restrict what page content the third-party scripts may access. For example, AdJail [2] puts third-party advertising scripts into a shadow iframe with a different domain, using the browser's same origin policy to isolate that frame and setting up a restricted communication channel between the shadow iframe and host page. However, AdJail and many similar approaches often rely on developer-defined policies and precisely identified private information.

Instead, we propose an alternative solution based on egress control that filters the outgoing traffic from untrusted scripts. Our approach duplicates responses into two pages, a real page which is displayed to the user, and a hidden redacted page which only contains insensitive page content. We compare outgoing traffic to untrusted servers between real page and redacted page, discarding message that are only seen from the real page but not from the redacted page. Our approach is similar to the idea of multi-execution to enhance application privacy used by Capizzi et al. [3] to prevent information leakage in Windows applications and FlowFox [4], a variant of Firefox that can enforce information flow control in web applications. However, these works did not address possible divergence issues in multi-execution process and their evaluations were limited to trivial policies that do not provide protection for any sensitive information in the DOM.

II. Design

We consider a script *safe* if it does not leak any sensitive information to untrusted servers. Our goal is to prevent untrusted scripts from leaking private data back to third-party servers. Our approach focuses on filtering the outgoing traffic rather than restricting third-party scripts' access to page resources. Scripts which have full access to the page can only initiate safe requests that do not leak any private information. A clear benefit this offers is that it allows library scripts such as jQuery to access sensitive information on the page while preventing them from giving those information away to untrusted destinations.

RedactDOM may be implemented as a browser extension or a proxy, as illustrated in Figure 1. Every web page, which we refer to as the *real page*, has a corresponding *redacted page*. The redacted page is created (1) in a way that preserves

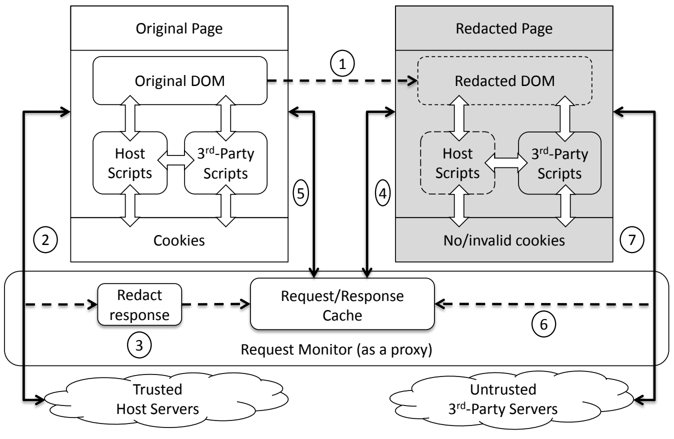

Fig. 1: RedactDOM Design

all trusted and untrusted scripts and DOM structure from the real page, but removes all potentially sensitive data including text content, attributes, and cookies associated with the original domain. Hence, requests from the redacted page are known to be safe. The *request monitor* observes outgoing requests from both pages. Since our focus is on preventing leaks to untrusted servers, all requests from the real page to the trusted server are allowed to proceed normally (2), but in addition their responses are redacted and then cached in the request monitor (3). As the redacted page makes the same request, its response will be directly drawn from the cache (4), thus preserving the number of requests sent from the browser. Vice versa, when the real page makes a request to an untrusted server (5), its response is directly drawn from cache if a matched request is made earlier (6), or is put on hold until the same request is made by the redacted page, in which case the request proceeds as normal (7) and responses are returned to both pages. The requests will be dropped if no matching request are found after a fixed period. Hence, all external servers see the same number of requests as seen from an unmodified client.

III. Open Questions

Our design raises a number of open questions, which we will explore by conducting a variety of synthetic and user-driven experiments using our implementation. The main questions concern whether it is possible to maintain page functionality with our redacted requests, and whether our security assumptions hold for most web applications.

A. Preserving Functionality

Our design relies on there being a clear correspondence between external requests (that do not leak private information) made by the redacted page and the real page. This could fail if the behavior of scripts on the redacted page diverge because of DOM differences in two pages. Since we are not able to precisely identify private information on the page, the redacted page is also missing all public information. If scripts' behavior on the redacted page depends on that content it will diverge from the scripts on the real page. It may also diverge because of nondeterminism in the execution.

Another concern is how to handle user inputs to the real page in the redacted page. The user inputs may be sensitive and should not be visible to the redacted page, but may lead to further divergences between the pages. We propose to conduct experiments to test if our system yields reasonable compatibility if user inputs are completely ignored in the redacted page. If this turns out to be unacceptable, we may need to replace the redacted page with a new copy that reflects any changes to the page structure that resulted from user input, while keeping the content redacted.

B. Security Assumptions

Our design assumes that private information can be easily redacted from the DOM. This depends on being able to identify and strip out the private information. We expect that all the private information is in the form of text content or attributes, but there is no guarantee that this is the case. For example, a site could easily embed private information directly in scripts. We do not consider the case where the server is malicious, though, since our goal is to prevent private information that the server already has from leaking to other parties. We plan to test this assumption by comparing the responses from a set of test servers on requests issued with different credentials.

We also assume the redacted page and real page can be sufficiently isolated to prevent scripts running on the redacted page from learning about user actions or redacted content on the real page. There could be side channels (such as timing of events) that would leak such information, and the adversary has a powerful resource in controlling scripts that run on the real page. Our design will need to limit the interaction between the two pages, and carefully reason about how much information may be leaked by any interactions.

IV. Conclusion

RedactDOM is an egress-based approach to prevent untrusted scripts from leaking sensitive data. By limiting external requests to those generated by a redacted page containing no sensitive information, we can provide high assurance that no sensitive information is leaking through these requests.

Our work is focused on privacy, but does not place any limits on what adversarial scripts could do to the real page, other than the external network traffic. This prevents some clear and important threats, but does not prevent annoying page modifications, or more seriously, forged requests to the origin site (which will have the client's credentials). We will explore variations on our redaction policy to include also requests to the origin site, but this will require a careful balance between compatibility and security.

References

[1] 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 *ACM Conference on Computer and Communications Security*, 2012.

[2] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan, "Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements," in *19th USENIX Security Symposium*, 2010.

[3] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla, "Preventing Information Leaks through Shadow Executions," in *Proceedings of the 24th Annual Computer Security Applications Conference*, 2008.

[4] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "FlowFox: A Web Browser with Flexible and Precise Information Flow Control," in *Proceedings of the 19th ACM conference on Computer and Communications Security*, 2012.