Thesis for the Degree of Doctor of Philosophy

Securing the mashed up web

Jonas Magazinius

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2013

Securing the mashed up web
Jonas Magazinius
ISBN 978-91-7385-917-2

Doktorsavhandlingar vid Chalmers tekniska hgskola
Ny serie nr 3598
ISSN 0346-718X

Technical Report 102D
Department of Computer Science and Engineering
Division of Software Engineering and Technology

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2013

Abstract

The Internet is no longer a web of linked pages, but a flourishing swarm of connected sites sharing resources and data. Modern web sites are increasingly interconnected, and a majority rely on content maintained by a 3^{rd} party. Web mashups are at the very extreme of this evolution, built almost entirely around external content. In that sense the web is becoming mashed up. This decentralized setting implies complex trust relationships among involved parties, since each party must trust all others not to compromise data. This poses a question:

How can we secure the mashed up web?

From a language-based perspective, this thesis approaches the question from two directions: attacking and securing the languages of the web. The first perspective explores new challenging scenarios and weaknesses in the modern web, identifying novel attack vectors, such as polyglot and mutation-based attacks, and their mitigations. The second perspective investigates new methods for tracking information in the browser, providing frameworks for expressing and enforcing decentralized information-flow policies using dynamic run-time monitors, as well as architectures for deploying such monitors.

Acknowledgements

During the five years that I have been working on this thesis some of the most significant events of my life has occurred; I married my wonderful wife Ana (well, to be honest that was two weeks before I started, but close enough I'd say); we became pregnant, an event in itself; not long after, we had our son and my hero, Dante. There have been other events as well, equally significant, not as positive. None of these events, positive or negative, have related to my work, but they have all had an impact on it in one way or another.

To my supervisor, Andrei Sabelfeld I am very grateful for having had you, as my supervisor. Besides our very successful professional relationship, you have been there for me through all events. Thank you, Andrei, for discussion, patience, encouragement and insight.

To my co-authors Aslan Askarov, Alejandro Russo, Phu Phung, David Sands, Mario Heiderich, Daniel Hedin and Andrei Sabelfeld, thank you for delightful co-operation and discussion.

To friends and colleagues Thank all of you!

Arnar, I already miss you! See you in Croatia?

Filippo, I can't express in words... Wait, I've already done that! W!

Jonas, you are too far away, it's been too long!

To my family You have been a part of every significant event and moment. This would not have been possible without your support.

Branko and Lidija, I love you! Ana could not have had better parents.

Rafael, I've told you that I'm proud of you, but you don't know how much you impress me! You're next!

Cornelia, it was so much better having you nearby! Come back?

Linnea, you are wonderful! Thank you!

Elias, soon we'll be celebrating you!

Gunnar, thanks for teaching me how to keep things in check! E2-E4

Anders, you are the one who is självklar!

To my mother Clearly I would not have been where I am today without you, that is sort of natural, but you I'm forever thankful for all that you have done for me over the years. I love you.

To my son I'm so proud of you. Already you have proven yourself stronger than most. You inspire me, always, to do better. I love you.

To the love of my life What a journey we have had so far. Sometimes apart, but always together. So many exiting things ahead, and I can't wait to share them with you. Together. I love you most of all.

In concluding this chapter of my life, I can't help but wonder if life will ever be as full of memorable moments.

Jonas Magazinius,
October 2013

Contents

CHAPTER 1

Securing the mashed up web

Securing the mashed up web

Jonas Magazinius

1 Introduction

The Internet is no longer a web of linked pages, but a flourishing swarm of connected sites sharing resources and data. Modern web sites are increasingly interconnected, and a majority rely on content maintained by a 3^{rd} party. Web mashups are at the very extreme of this evolution, built almost entirely around external content. In that sense the web is becoming mashed up. This decentralized setting implies complex trust relationships among involved parties, since each party must trust all others not to compromise data. This poses a question:

How can we secure the mashed up web?

From a language-based perspective, this thesis approaches the question from two directions: attacking and securing the languages of the web. The first perspective explores new challenging scenarios and weaknesses in the modern web. Identifying and mitigating new attack vectors proactively prevents them from being exploited. The second perspective investigates new methods of enforcing decentralized policies for monitoring information flow in the browser. Providing tools for enhancing security helps in mitigating existing problems.

2 Background

In the early days of the Internet static web pages were served from single web servers using a stateless protocol. Thereby the security considerations were confined within these boundaries. Since then the Internet has evolved to a complex patchwork of technologies and applications. Functionality has generally been the driving force behind this evolution and security has often been a secondary concern.

The most significant technological milestone in the history of the web is the introduction of JavaScript [15]. Developed in 1995, this highly dynamic language greatly boosted the dynamics and responsiveness of web pages. At the same time JavaScript's powerful capabilities to access and manipulate data in the browser opened up for a new era of vulnerabilities. To address the security concerns introduced by JavaScript, a set of rules were set up to confine its power. At the very core stands the same-origin policy (SOP) [19]; the fundament of web security designed to separate and isolate web pages from each other.

2.1 The same-origin policy

The same-origin policy classifies documents based on their origins. The notion of an origin is simple; the combination of the protocol, domain and port of the URL of the document. According to the SOP, documents from the same origin may freely access each other's content, while such access is disallowed for documents of different origins. However, the SOP does not prevent from including content of a different origin within a document. The SOP still applies; it is not possible to read the contents of a node, e.g., an image or a script, that was loaded from a different origin, but scripts are executed in the same scope, regardless of origin, and can access each other's resources. This allows for including libraries, developed and maintained by 3^{rd}-parties, that add to the functionality of the page. Libraries like jQuery [4] and Google Analytics [14] are included in roughly two thirds of the top 10,000 most popular web pages according to services like builtwith.com [10, 9]. Including libraries for, e.g., accessing information from social networks or gathering statistics, has become commonplace.

Retrieving or disclosing such information requires relaxing the SOP to establish a channel for communicating across origins.

2.2 Cross-origin communication

There are multiple methods of relaxing, or even circumventing, the SOP. Some are by design, some leverage unintended features of the browser. JavaScript object notation (JSON) [5] and cross-origin resource sharing (CORS) [3] are two good examples of such features.

JSON was originally a convenient hack, but has become an established format for delivering data. In web pages the XMLHttpRequest-function is used by scripts to request new content. The content returned is simply a serialized JavaScript object, which is then either parsed or evaluated to access the values.

It is possible to use JSON to communicate across origins, then referred to as JSON with Padding (JSONP). Unlike JSON, JSONP uses dynamic script inclusion, i.e., dynamically adding a script node to the document, for communicationand the padding part of JSONP is a call to a predefined callback function. It is considered to be insecure because using dynamic script inclusion implies that any JavaScript code in the response will be executed, not just the callback function. The callback function could also be overridden by another script, known as JSON Hijacking [16, 11], which would then receive the data. If the data contain sensitive information, a script could, intentionally or not, leak that information.

CORS allows an origin to list a set of trusted origins that are allowed to request certain data despite being of different origin, thereby intentionally relaxing the SOP. When using XMLHttpRequest to access data

across origins, it will first request the CORS policy file, before proceeding with the actual request. CORS enables using, e.g., JSON or XML, in communication across origins, without compromising the security of the page. One issue that has been brought up is that CORS policies are very coarse grained. As an example, there is no way of specifying how the data may be used once it has been delivered.

Cross-origin attacks As mentioned in Section 2.1, scripts in a document are executed in the same scope and have access to each other's resources. Also sensitive information, such as cookies, personal user information, and session tokens, live in this scope. Therefore it is a constant target for attackers to try to get code executing in the scope of an origin.

There are several classes of cross-origin attacks that circumvent SOP. A classical XSS attack exploit vulnerabilities to injects a malicious script into the code of a document. A more recent class of attacks abuse common languages and formats on the web other than JavaScript. Two noteworthy examples are GIFAR [8] and cross-origin CSS attacks [13]. In a GIFAR attack, the GIF and JAR (Java archive) formats are combined to create a file that appears to be a benign image but can be interpreted as a malicious JAR file that can bypass SOP. In a Cross-origin CSS attack fragments of CSS code are injected into an existing web page to extract information from the existing web page.

Despite being well known and thoroughly researched, cross-origin attacks continues to plague the web. The OWASP top 10 list [17] places both *injection* and *cross-site scripting* (XSS) attacks among the three top security risks for web applications.

2.3 Web mashups

According to the directory services programmableweb.com [1] there are, at the time of writing, more than 7,000 mashups. A mashup consist of a hosting page, usually called the integrator, and a number of 3^{rd}-party components, often of different origins. Mashups are interesting to study because they are decentralized in the sense that the only responsibility of the integrator is to coordinate the components. The integrator does not control the interaction or flow of information between components, nor does it have leveraged privileges compared to the other components. As discussed, including libraries implies that the integrated components will all execute in the same context under the integrating origin. Mashups, by nature, involve interaction between components and executing in the same context opens up a platform to do so. However, this does not necessarily mean that the components are aware of each other's existence within a page and components have ample opportunity to covertly leak each other's data. To put this into context, we proceed with two scenarios.

Mashup scenarios The following two scenarios illustrate two security issues that may arise when building a mashup.

Secure cash transportation Consider a company specializing on cash transportation services. The company require tracking of their armored trucks at all times in the event that a truck is hijacked. To minimize the risk of a truck being hijacked, the company randomizes the order of the pick-up points to vary the route of each truck.

The company intends to create a private web application to keep track of their armored trucks. However, visualizing the location of a truck on a map is beyond their capabilities and therefore they have decided to build a mashup where this service is provided by a 3^{rd}-party.

At this point a problem arises, because of the coarse-grained nature of mashup composition they have to fully trust the mapping service with their confidential information. The route of a truck is highly confidential, and so is the exact location of the truck. The mapping service could either unknowingly or deliberately leak the route or location, or intentionally misrepresent the location to the operator. Even if the mapping service itself has no malicious intentions, a result of this integration is that the security of the web application, as a whole, now also rely on the security of the mapping service.

TweetFace.com In this scenario, a mashup is combining the APIs of Twitter and Facebook to aggregate the users social network information in one place. The integrator is responsible for initializing the libraries, load the network feeds, and aggregate the data. However, the integrator has no idea about the inner workings of the libraries, and there is nothing preventing either one of the libraries from doing the same aggregation. Either one of the libraries could be aware that it will be used in this context and covertly abuse the privileges of the integrator to extract as much users information as possible from the other social network. The malicious library will piggyback on the privileges that the integrating application has in the non-malicious network.

Mashup security Due to the apparent limitations of mashups, mashup security has been discussed intensively in recent years. A number of approaches have been proposed and a recent survey by De Ryck *et al* [20] summarizes the state-of-the-art in mashup security. The survey identifies key challenges in mashup security and categorizes the proposals based their contribution to each challenge. Rather than summarizing the categories identified by De Ryck *et al*, we will list the general techniques used in the proposed approaches.

JavaScript subsets A number of approaches involve using well-behaved subsets of JavaScript. By syntactically filtering the language, problem-

atic language features can be disallowed. This technique is unnecessarily coarse-grained, as it must reject programs that uses, potentially harmful, restricted language features, even when used in a benign way.

Code rewriting Code rewriting is another popular technique, where program code is parsed and rewritten to a semantically equivalent, but restricted program. Apart from ensuring that the code adheres to a subset, this technique supports making decisions in run-time, making it more fine-grained. However, it lacks support for writing dynamic policies, i.e. for controlled release of information.

Relaxing the SOP Other techniques rely on relaxing the SOP to allow partial cross-domain interaction, either by existing ways of bypassing the SOP, or by modifications to the browser. When information is communicated in this manner, the principal sharing the information looses all control over how it is used on the receiving end.

What these techniques generally lack is being able to define and enforce policies for fine-grained control over the flow of information within the application. In other words, To be able to specify and control where information is allowed to flow within a program.

2.4 Information flow security

Language-based information-flow security [21] considers programs that manipulate pieces of data at different sensitivity levels.

As an example, the first sample mashup from Section 2.3 operate on sensitive (secret) data such as the routes and locations of armored trucks and at the same time on insensitive (public) data such as 3^{rd}-party mapping services.

A key challenge is to secure *information flow* in such programs, i.e., to ensure that information does not flow from secret inputs to public outputs. Or to relate to the samples, that routes and locations is not leaked via map coordinates sent to the map service.

Policies for information flow allows us to define, in a precise manner, the security and integrity level of data, as well as how it can be shared with other principals. Declassification policies describe the intentional release of information, i.e., from a more secret level to a less secret or even public level.

Information flow policies can be enforced either statically or dynamically. Static enforcement analyses the program, identifying information sources and sinks, and determines from the structure of the program whether it will respect the policy or not. Dynamic enforcement instead monitor the execution of the program, making runtime decisions whether to proceed with an action or not.

The driving force for using the dynamic techniques is expressiveness: as more information is available at runtime, it is possible to use it and accept secure runs of programs that might be otherwise rejected by static analysis. Dynamic techniques are particularly appropriate to handle the dynamics of web applications, where scripts can utilize dynamic code evaluation to parse strings to code at runtime.

Already in 1996 the browser vendor Netscape were thinking along these lines when they considered replacing the same-origin policy by introducing taint tracking in their flagship browser Netscape Navigator 3 [2]. Taint tracking taints secret variables and propagates this taint through any operation that does computation on a tainted variable. The project did not succeed seeing that taint tracking fails to take into account implicit information-flows, and the method turned out to be less secure than the existing policy.

Mozilla's ongoing project FlowSafe [12] aims at empowering Firefox with runtime information-flow tracking, where dynamic information-flow reference monitoring [6, 7] lies at its core.

Looking at the second sample mashup from Section 2.3, there are multiple principals, e.g., Facebook and Twitter, each with their own sensitive information. Each of them also has a security policy that determines what data is secret or not. Using language-based tools for tracking information flow in the browser, it is possible to track how sensitive inputs propagate as scripts are executed. Initially we build a security lattice over all involved origins, and use this inferred lattice for labeling inputs to their respective origins. Whenever secret information is used in an operation, the result is labeled with a label that is at least as restrictive. When this secret information, or information derived from it, reaches a public output, such as to be communicated to a different origin, the monitor inspects the label and decides how to proceed, i.e., if the label is less-or-equal than the target origin, it is allowed, otherwise execution is stopped. This allows for secure collaboration on secret data without the risk of compromising data. In addition, declassification allow for controlled release of information and can be used to share the minimum of information a collaborator requires.

3 Thesis contributions

As stated in the introduction this thesis explores two perspectives on the modern web, hence, the contributions can be structured in two main categories; attacking and securing languages. The first category contributes new attack vectors and approaches for mitigating these attacks. The second category can be further decomposed into the work on the definition of decentralized policies, and the work on enforcing such policies. It contributes formal definitions for declassification in a decentralized setting,

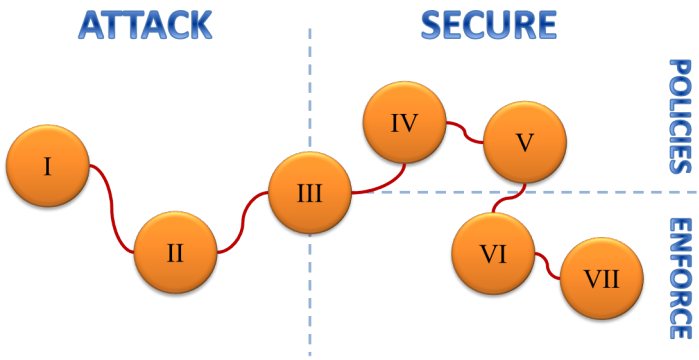

Fig. 1. Contribution overview

new techniques for enforcing policies and architectures for deploying security monitors.

Taking a bird's-eye view on the thesis, Figure 1 visualizes how the papers relate to each other and the aforementioned categories. Paper I introduces the section on exploitation and connects to Paper II, as they both identify new attack scenarios. Paper III spans all categories as it involves both attacks on security monitors, their mitigations, and work on defining "sane" policies. From there on the focus is on policies and monitoring, which is introduced in Paper IV, and further elaborated in the two following papers Paper VI and Paper V. Here, Paper VI explore run-time monitoring, while Paper V expands the work on policies from Paper IV. Finally, Paper VII ties the knot on the thesis, evaluating architectures for distributing monitors to the end user.

3.1 Exploiting languages

In our efforts to secure the web, we tend to focus solely on JavaScript and forget about the full complexity of the browser and all the languages and formats it is built to handle. This opens up for new attack vectors, outside the scope of current research. In Paper I and Paper II two such attack vectors are identified, and concrete suggestions for mitigation measures are presented. In Paper III attacks against a class of monitors, i.e., monitors that execute in the same scope as the code they are monitoring, are identified and proceeds to contribute practical solutions to these attacks.

Polyglot attacks Paper I paper is the first to present a generalized description of polyglot attacks. It also provides characteristics of what constitutes a dangerous format in the web setting and identify particularly dangerous formats, with PDF as the prime example. An in-depth study

of PDF-based injection and content smuggling attacks, unfortunately, it shows that several major web sites do not protect against polyglot attacks. Several mitigation approaches for polyglot attacks are suggested, both general and specific to particular components and formats. The foremost candidate among these approaches is an expected type declaration, where the browser informs the web server what type of content it expects to retrieve based on the context it will be interpreted in. The web server can then interrupt the request if there is a mismatch in the expected and actual content type.

Mutation-based XSS attacks Paper II contributes several attacks based on mutation of HTML in the browser, and evaluates of the prevalence of mutation patterns. The pattern, serializing and re-interpreting HTML content, turns out to be common and can be found in multiple JavaScript libraries as well as in web pages around the web. Paper II also provides mitigations against mutation-based attacks and proposes *trueHTML* – HTML free from mutations.

Monitor attacks Another consideration is how to make monitors tamper resistant, when executing in an environment under attacker influence. Paper III highlights weaknesses in *Lightweight Self-Protecting JavaScript*, developed by Phung *et al* [18]. It identifies several attacks that affect all monitors that execute alongside the code they are monitoring. Each attack is mitigated and a "safe" wrapper is presented, able to withstand all identified attacks.

3.2 Securing languages

This part of the thesis focuses on dynamic decentralized web applications, such as web mashups, and the problems encountered when several principals collaborate and share resources. We consider the problems of defining policies (Paper , and) and enforcing them (Paper , and) in this decentralized setting.

Policies Paper IV proposes a lattice-based approach to mashup security. The security lattice is built from the origins of the mashup components so that each level in the lattice corresponds to a set of origins. The key element in the approach is that the security lattice is inferred directly from the mashup itself. The classified information is given a label that correspond to the origin from which it was loaded. The labels are used to track information flow within the browser and to prevent information flowing from a higher level in the lattice to a lower, i.e., from a secure context to a less secure context.

Relating this to the first example in Section 2.3, this would prevent the mapping service from being able to leak the route or current location of

the trucks. Client-side interaction within the boundaries of the browser, e.g., plotting the route on the map or displaying the current position, is still possible, but cannot be communicated back to the service provider. Herein lies a problem, unless the entire database of images is included in the library, the map service must know at the very least the corner points of the map in order to provide the map associated with that area. This requires a mechanism to release certain specific, less secret, data to the map service; it requires *declassification*.

Paper IV brings up declassification and how individual release policies of each origin can be combined in a composite policy. It extends previous work on delimited release to a setting with multiple origins and provides a formal definition of *composite delimited release*.

The ideas of composite delimited release is further extended in Paper V which focuses on *declassification* policies, i.e., policies for intended information release. Here *decentralized delimited release* is introduced, a decentralized language-independent framework for expressing what information can be released. The framework enables combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it. While composite delimited release requires that the principals *syntactically* agree on escape hatches, Paper V removes this limitation, allowing principals to instead agree *semantically*. This is a great advantage as principals is no longer required to know the exact syntax of collaborating components.

Again relating this to the sample mashup, declassification allows the transportation company to define a fine-grained policy to, e.g. reveal the corner points of the required map to the map service but not reveal sensitive information about the route.

The issue of writing "sane" declarative policies is discussed in Paper III. Sane in the sense that the inner workings of the policy should not be influenced by the input it inspects, and declarative in the sense that the policy declares what it expects to inspect via inspection types and the monitor ensures that the input adheres to the declared inspection types.

Enforcement Paper VI presents a framework for *on-the-fly inlining* of dynamic information-flow monitors. Inlining means transforming the source code, adding monitor checks at critical points in the program. We consider a source language that includes dynamic code evaluation of strings whose content might not be known until runtime. To secure this construct, our inlining is done on-the-fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment. This is done by inlining a call to the transformation function where ever code evaluation can occur, rewriting

the argument on-the-fly. Paper VI also discusses practical considerations, experimental results based on both manual and automatic code rewriting. Paper V provides a reference implementation for a subset of JavaScript, that builds upon the inlining techniques described in Paper VI, which is then applied to two sample scenarios.

Phung *et al* [18] describe a method for wrapping built-in methods of JavaScript programs in order to enforce security policies. The method is appealing because it requires neither deep transformation of the code nor browser modification. Unfortunately the implementation outlined suffers from a range of vulnerabilities. Paper III deals with a number of problems encountered in the approach, but that could be generalized to other enforcement techniques. The techniques applies whenever the code of the monitor is required to be executed in the same context as the code it is monitoring. In this setting, the monitored code has ample opportunity to influence and subvert the monitor and thereby execute unmonitored. The paper enumerates a number of attack patterns and how they can be mitigated.

Paper VII brings up monitors that enforce policies by replacing the ordinary execution environment, e.g., by substituting the JavaScript engine of the browser. Replacing the ordinary execution environment with one that enforces policies that are otherwise not supported, effectively protects against a wider range of attacks.

As the main contribution, Paper VII proposes architectures for deploying security monitors for JavaScript: via browser extension, via web proxy, via suffix proxy (web service), and via integrator. Our evaluation of the architectures explores the relative security considerations.

4 Summary

In a web of constant evolution, the problem of keeping user data confined within the boundaries of the browser is a moving target. Web pages of today thrive on dynamicity and are built around 3^{rd}-party content, with mashups being the prime example. A side-effect is that each origin in turn depends on the security of all 3^{rd}-party code it includes. The result is a web of trust, where security depends on the weakest <a>.

In the beginning of this chapter we posed the question: how can we secure the mashed up web?

As is shown in this thesis, such dynamicity requires fine-grained control over how information flows within the browser. By utilizing language-based tools such as information-flow control and declassification, it is possible to enforce fine-grained policies on individual data elements to prevent information leakages. Dynamic information-flow monitors allows

us to track secret information as it is being manipulated by an executing program. Declassification allows for controlled release of information, at the exact time, but not earlier, it is required to be released. Combined they provide the necessary means to confine secret information in the browser, while at the same time allowing the full dynamicity of the mashed up web.

5 Statement of contribution

Paper I – Polyglots: Crossing Origins by Crossing Formats

Accepted for publication in Proceedings of ACM Conference on Computer and Communications Security

My co-authors and I contributed equally to the technical material and the writing of this paper. The technical material was independently discovered and later refined together with my co-author Billy K. Rios. My co-author Andrei Sabelfeld and I were the main contributors to the writing of the paper.

Paper II –mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations

Accepted for publication in Proceedings of ACM Conference on Computer and Communications Security

My co-authors and I contributed equally to the technical material and the writing of this paper. My main contribution was in the evaluation of the prevalence of this type of vulnerabilities. My contribution to the writing of the paper was focused on the section on evaluation, but extended into the other sections as well.

Paper III – Safe Wrappers and Sane Policies for Self Protecting JavaScript

Presented at OWASP AppSec Research 2010 and published in the joint Proceedings of the 15th Nordic Conference in Secure IT Systems (Nordsec'10)

My co-authors and I contributed equally to the technical material and the writing of this paper. I provided the technical knowledge behind some of the attacks described and together with my co-author Phu H. Phung, I extended his original prototype implementation.

Paper IV – A Lattice-based Approach to Mashup Security

Published in Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)

My co-authors and I contributed equally to the technical material and the writing of this paper.

Paper VI – On-the-fly Inlining of Dynamic Security Monitors

Published in Journal of Computers & Security

My co-authors and I contributed equally to the technical material and the writing of this paper. I wrote the formal proofs and provided a prototype implementation.

Paper V – Decentralized Delimited Release

Published in Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'11)

My co-authors and I contributed equally to the technical material and the writing of this paper. The formal definitions and proofs was mainly done by my co-author Aslan Askarov, while I implemented a prototype implementation and performed the experiments.

Paper VII – Architectures for Inlining Security Monitors in Web Applications

My co-authors and I contributed equally to the technical material and the writing of this paper. My contribution in terms of technical material was the design and implementation of three of the four distribution architectures.

References

1. http://www.programmableweb.com/.
2. http://docstore.mik.ua/orelly/web/jscript/ch20_04.html.
3. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.
4. jQuery. http://jquery.com/.
5. JSON.org. http:/json.org/.
6. 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.
7. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.
8. R. Brandis. Exploring Below the Surface of the GIFAR Iceberg. An EWA Australia Information Security Whitepaper. Electronic Warfare Associates-Australia, Feb. 2009.
9. BuiltWith.com. Google Analytics Usage Statistics. http://trends.builtwith.com/analytics/Google-Analytics.
10. BuiltWith.com. jQuery Usage Statistics. http://trends.builtwith.com/javascript/jQuery.

11. B. Chess, Y. T. O'Neil, and J. West. JavaScript Hijacking. http://tr.im/jshijack. Accessed in January 2010.
12. B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, Oct. 2009.
13. L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, pages 619–629, Oct. 2010.
14. G. Inc. Google Analytics. http://www.google.com/analytics/.
15. Mozilla.org. Mozilla Developer Center: JavaScript. https://developer.mozilla.org/en/JavaScript.
16. Open Ajax Alliance. Ajax and Mashup Security. http://tr.im/ajaxmashupsec. Accessed in January 2010.
17. Open Web Application Security Project (OWASP). OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013, 2013.
18. P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-Protecting JavaScript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.
19. J. Ruderman. Mozilla Developer Center: JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript.
20. P. D. Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *Proceedings of the 15th Nordic Conference in Secure IT Systems (Nordsec)*, Oct. 2010.
21. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.