# PhD research proposal

Enrico Steffinlongo

May 30, 2014

## Background and Motivations

In the last few years there has been an increment in the usage of applications that have to interact both with user sensitive data and with the untrusted outer world. This increment is led by browser applications (HTML5 apps, browser extensions,...) and by smartphone apps, where there is unfortunately little control on the quality and the safety of software. To mitigate these risks, typical proposals are built on a privilege separated architecture that divides the system in different components, giving to each component only limited permissions; to preserve interaction between isolated components, the latter can communicate using a tight Message Passing Interface. To avoid privilege escalation attacks where a component that is compromised by an attacker obtains permissions that it did not originally have, permissions are usually given statically, before the execution of the application. According to the permission system, there is a strict punctual mapping between components and permissions in order to reduce the possible exploits of the attacker.

The message passing interface between components is often regulated by a centralized monitor and it can deliver different kinds of messages: from simple strings to complex object, with pointers and methods. The message passing interfaces constitutes a relevant attack surface against privilege-separated applications, because it may lead to privilege escalation attacks, when a compromised, or malicious component that does not have a permission can send messages asking other components to trigger security-sensitive operations.

An important example of real systems with the features above is Android. As discussed in [5, 8, 9, 7, 10, 3] this architecture suffers various attacks like privilege escalation, since applications written by programmers that are not security experts can expose a large attack surface, being over-privileged or lacking proper checks on the incoming messages.

Another case of privilege separated architecture which uses a simple message passing interface is the Google Chrome Extension framework. In this architecture, that extends the functionality of the browser, as shown in [2, 4], the choice is to separate every app in two sets of components: Backgrounds and Content Scripts. The former is the part of the application that interacts with sensible data, such as password and cookies, and with the browser core; as such, it is isolated from the page on which the extension is running, in order to prevent a malicious script in the page from being executed with high privileges. Content Scripts, instead, have no permission except the one used to send messages to the Background, and they have access to the page on which they are injected.

To avoid direct delegation of privileges from the background page to some content script, valid messages include only strings: objects, functions and pointers cannot be exchanged. This choice restricts the attack surface, since an attacker cannot send functions to be executed in the privileged background page. Unfortunately, it also reduces the expressiveness of the language, since any object is marshaled using a JSON serializer to a string that contains only values or pair field-value. This serialization fails in presence of recursive objects (e.g., DOM elements) and breaks the typical prototype-based inheritance of Javascript. This reduces severely the expressiveness of the framework, since developers are often forced to adopt complex workaround just to perform simple programming tasks.

We argue that current privilege-separated architectures suffer limitations that affect both functional and non-functional requirements. For example, even though sometimes permissions can be requested and revoked dynamically (e.g., in Chrome Extensions [1]), the programmer often gives to each component the maximum of the permissions required throughout its execution, even when a certain privilege is only rarely used. The study of disciplined programming patterns for the dynamic acquisition and revocation of privileges would be important to make applications more robust.

Another crucial aspect in the design of an architecture for developing security critical software is the choice of the adopted languages. Scripting languages tend to be easier to deal with and are a natural choice for developing simple applications, but they suffer of their weak typing discipline, which makes it hard to statically check software written with them and guarantee its safety. This is the reason why privilege-separated architectures implement several tight restrictions discussed above, but recent research[11, 14, 12, 15, 16] shows promising results in analyzing also weakly-typed scripting languages, hence one may wonder how existing privilege-separated architectures could be redesigned to improve their expressiveness, without sacrificing security (by demanding it to programming tools, like code analyzers or type checkers).

# Proposal

In this scenario we want to enhance the potentiality of privilege-separated architectures without sacrificing their security. The study will start from a formal analysis of the actual solutions proposed so far, understanding their goals and intrinsic limitations, with particular attention to the safety properties and to the expressiveness that these solutions grant to the programmer; we want to identify further improvements over them and to develop tools for checking existing software.

Examples of enhancements can regard various aspects of the framework, such as:

- potentialities enabled by a more structured security policy on the message passing interface, which can allow exchange of pointers and functions, without compromising the reliability of the system;
- impact of a more fine-grained global security policy that allows controlled delegation patterns and more expressive mechanisms for asking for and revoking privileges.

The security analysis of our proposals will be done using both static and dynamic techniques, which allow for ensuring their formal correctness. We plan to adopt well-established verification techniques, like typing [12], flow logic [18, 13, 17], abstract interpretation [6, 14, 16, 15] and model checking.

At the time of writing we already found that the Chrome Extensions architecture suffers a lack of a sound centralized security discipline on incoming messages. This may be a source of security flaws when combined with the typical programmers practice that essentially trusts the incoming messages and trigger security sensitive operations based on them (this is a consequence of the so-called permission bundling). To understand better this point, let us consider three components A, B, C. Component A runs with high privileges and receives messages from both B and C; B and C have low privileges and they ask A to perform distinct security sensitive operations O1 and O2 using different types of message M1 and M2. Assume that B only sends M1, while C only sends M2. If A trusts its incoming messages and bases the choice to perform O1 or O2 only based on them, then a compromised B can send M2 rather than M1 to A, thus triggering O2. Hence, the attack surface against this application is unreasonably large. To mitigate this risk, it is better to base the choice of A on a reliable information and not just on messages, that could be altered by compromised components. Our proposal is to write a tool based on a formal analysis that checks if an extension suffer flaws like the one described and furthermore automatically fixes them by inserting in the code stronger security checks.

# References


[1] Chrome extension overview
https://developer.chrome.com/extensions/overview, May 2014.

[2] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. Technical Report UCB/EECS-2009-185, EECS Department, University of California, Berkeley, Dec 2009.

[3] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Lintent: Towards security type-checking of android applications. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems*, volume 7892 of *Lecture Notes in Computer Science*, pages 289–304. Springer Berlin Heidelberg, 2013.

[4] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[6] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *Proceedings of the 13th International Conference on For-*

*mal Methods and Software Engineering*, ICFEM'11, pages 505–521, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[10] Adrienne Porter Felt, Steven Hanna, Erika Chin, Helen J. Wang, and Er Moshchuk. Permission re-delegation: Attacks and defenses. In *In 20th Usenix Security Symposium*, 2011.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 256–275, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Rene Rydhof Hansen. Flow logic for carmel. Technical report, 2002.

[14] David Van Horn and Matthew Might. An analytic framework for javascript. *CoRR*, abs/1109.4467, 2011.

[15] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *SIGSOFT FSE*, pages 59–69, 2011.

[16] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[18] Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation*, pages 223–244, 2002.